Checking Just Pairs of Threads for Efficient and Scalable Incremental Verification of Multithreaded Programs

Pavel Parízek
Charles University
Prague, Czech Republic
parizek@d3s.mff.cuni.cz

Filip Kliber
Charles University
Prague, Czech Republic
kliber@d3s.mff.cuni.cz

ABSTRACT

Many techniques of automated verification target multithreaded programs, because subtle interactions between threads may trigger concurrency errors such as deadlocks and data races. However, techniques and tools involving systematic exploration of the whole space of possible thread interleavings do not scale to large software systems, despite various clever algorithmic optimizations. A viable approach is to use incremental verification techniques that, in each run, focus just on the recently modified code and the relatively small number of affected execution traces, and therefore can provide results (bug reports) very quickly.

In this paper we present a new algorithm for incremental verification of multithreaded programs based on the *pairwise approach*, whose key idea is systematic exploration of all possible thread interleavings just for specific relevant pairs of threads.

We implemented the algorithm with Java Pathfinder as the backend verification tool, and evaluated it on several multithreaded Java programs. Results show that our incremental algorithm (1) can find errors very fast, (2) greatly reduces time needed for complete safety verification, and (3) it can find the same errors as full verification of the whole state space.

1. INTRODUCTION

An important subject of automated verification and bug detection techniques are multithreaded programs, especially because subtle interactions between threads may trigger concurrency errors such as deadlocks and data races during the program execution. Techniques most suitable for precise detection of possible concurrency errors involve systematic exploration of the whole space of all possible thread interleavings that may occur at runtime. However, exhaustive systematic verification does not scale to large real-world multithreaded software systems due to state space explosion [2]. The main cause of limited scalability is the huge number of thread interleavings that have to be analyzed even for rather small multithreaded programs. State-of-the-art techniques and tools partially address this challenge by using many clever algorithmic improvements, optimizations and heuristics, but scalability of verification to large and complex software systems remains to be an issue.

One viable approach is to use incremental verification techniques that, in each run, analyze just the recently modified source code and the corresponding small subset of affected thread interleavings. The main practical benefit of such incremental procedures is that they can be run very often — for example, after each commit to a source code repository — and they provide the list of detected bugs very quickly, allowing developers to fix the reported bugs while they have the respective code in fresh memory. Several research groups already published some work in this direction [1, 7, 11], targeting also efficient incremental search for concurrency errors.

Overview. In this paper, we propose an algorithm for incremental verification of multithreaded programs that is based on systematic exploration of possible thread interleavings just for specific pairs of threads. We call it the *pairwise approach*. The key idea behind our pairwise approach is that, when developers edit the code of a specific program thread, it is sufficient to check all possible interleavings just for the modified code and in a pairwise manner with respect to other threads — that means checking the interleavings for each relevant pair of threads separately. Every run of our verification algorithm focuses just on the most recent source code modification, that means on detecting possible concurrency errors that involve the modified code, assuming the previous version of the input program was already verified and is free of concurrency errors.

For the purpose of explaining our approach, we consider only those source code modifications that involve program statements that represent possible interaction between threads — this includes, for example, reading or writing a field of a shared heap object, lock acquire statement, lock release, and so on. We use the phrase *thread interaction statements* to denote such statements.

A run of the incremental verification procedure based on our pairwise approach would be triggered when the developer inserts or deletes a thread interaction statement to/from the program source code. First, the verification procedure identifies all static program threads that may execute the modified code fragment. We denote this set of threads by the term *modified threads*. Subsequently, the procedure checks all possible interleavings for every pair of threads, where at least one element of a pair belongs to the set of modified threads. In this way, our verification procedure checks all possible interactions of the modified source code fragment (affected block of program statements) in the respective threads with the current version of the source code in all other threads, and performs the checks in a pairwise manner.

For illustration, consider the program in Figure 1, which involves three static threads (T1, T2, and T3). Two dynamic instances of T2 are created at runtime, while just one instance is created for each of the other threads T1 and T3. When the developer adds a call of print(o.f) into the code of T2, which is underlined in the source code listing, then our incremental verification algorithm checks the following pairs of threads: (T2 T1), (T2 T2), (T2 T3). The pair (T2 T2) has to be analyzed because the program involves two dynamic instances of thread T2.

```
T1: o.f = x ; t2a = start T2 ; t2b = start T2
T2: evaluate(o) ; print(o.f)
T3: y = p.g ; z = r.h; w = y + z
```

Figure 1: Example program

We also want to emphasize that, even though our verification algorithm checks all interleavings just for each pair of threads separately, it reports all concurrency errors that involve the modified code — and, in particular, even all errors in synchronization patterns that involve more than two concurrent threads. Details are provided in Section 3.2.

Contribution. The main contributions presented in this paper include:

- Algorithm for incremental verification of multithreaded programs based on systematic exploration of all possible interleavings just for specific pairs of threads.
- Experimental evaluation of the prototype implementation of our algorithm in Java Pathfinder [20] on 9 multithreaded Java programs.

Results of experiments show that runs of the incremental verification procedure finish very quickly for every subject program and every source code modification that we considered in our evaluation. For the most complex program in our benchmark set, the verification procedure finished on average in 16 seconds (when considering just experiments focused on fast search for concurrency errors), respectively in 18 seconds (experiments focused on checking safety). In addition, our incremental verification algorithm can find the same errors as if the full verification of the whole program state space is run every time. Since these results are promising, in the future we plan to perform additional experiments on even larger programs. Besides that, we will make the definitions of the core verification algorithm and proofs of soundness more formal.

In the next section, we define important terminology and notation used within the rest of this paper. We describe the proposed incremental verification algorithm based on checking interleavings for specific pairs of threads, and briefly discuss important properties of the algorithm, in Section 3. Then we show the results of our experiments in Section 4 and provide overview of related work in Section 5. Additional information that does not fit into this paper (regarding properties of the algorithm, evaluation, and related work) is available in a technical report [15].

2. NOTATION AND TERMINOLOGY

We define *thread interaction statements* more precisely as follows. It is any program statement that accesses (reads or updates) a memory location reachable from multiple threads. The memory location can be an object field, an array element, or a lock status variable, for example.

An atomic *transition* in the state space is defined as a sequence of statements that (i) begins with a thread interaction statement and (ii) then contains any number of thread-local statements, with effects not visible to other threads. All statements in a transition are executed by the same thread. A program code location is a *scheduling-relevant point*, if the next instruction to be executed represents a thread-interaction statement.

We use the term *modified code fragment* to denote a piece of source code that includes at least one added or deleted thread interaction statement, together with the affected nearby code within the same procedure. More precisely, we define the modified code fragment as a list l_{mod} of adjacent program statements within a single procedure. Note also that the whole sequence l_{mod} always belongs to the code of single program thread T. To simplify presentation, here we assume that the developer edited (added or removed) just a single thread interaction statement *st* in each step of iterative program development, because every modified thread interaction statement must be processed separately.

3. PAIRWISE APPROACH TO EXPLORATION OF THREAD INTERLEAVINGS

For the purpose of explaining our approach, we assume that the input for a run of the verification procedure consists of a modified code fragment, which is specified by a pair of program code locations that represent boundaries of the respective sequence l_{mod} of program statements. We use the symbol T_{mod} to denote the static modified thread (in the program source code) that contains the modified code fragment l_{mod}.

If the given input program involves multiple dynamic instances of the static thread T_{mod}, our verification algorithm considers every two dynamic instances of T_{mod} as distinct dynamic threads for the purpose of creating pairs of threads subject to verification.

3.1 Main Algorithm

When given the modified code fragment l_{mod} and the static modified thread T_{mod} as input, our core verification algorithm explores all possible interleavings that involve l_{mod} for all pairs (T_i, T_j) of dynamic threads, where T_i is a dynamic instance of T_{mod} and T_j is a dynamic instance of some static thread from the set $\mathcal{T}$ of all program threads. The key feature of the verification algorithm is that it processes each pair of threads separately, one at a time. Figure 2 captures the most important aspects of our verification algorithm, which we explain below in this section.

```
for $T_i \in$ getDynamicInstances($T_{mod}$) do
  for $T_j \in$ getDynamicInstances($\mathcal{T}$) do
    exploreInterleavingsForThreadPair($\mathcal{T}, T_i, T_j$)
  end for
end for

procedure onThreadSchedulingChoice($T_{cur}, T_i, T_j$)
  $p_i$ = getThreadCurrentPC($T_i$)
  if $T_{cur} == T_j$ then
    if $p_i == l_{mod}$.entry then return $\{T_i, T_j\}$
  end if
  if $T_{cur} == T_i$ then
    if $p_i \in l_{mod}$ then return $\{T_i, T_j\}$
  end if
  return $\{T_{cur}\}$
end proc
```

Figure 2: Important parts of the main algorithm that explores all interleavings for a pair of threads

The set $I_{i,j}$ of thread interleavings explored for a given pair of dynamic threads, (T_i, T_j), where T_i is a dynamic instance of T_{mod}, is defined as follows. For every scheduling-relevant point p_j in the code of T_j, the set $I_{i,j}$ contains an interleaving in which the first statement in the sequence l_{mod} is scheduled to be executed at p_j. Thread T_i executes the code fragment l_{mod} in that particular interleaving. In addition, for each specific point p_j and for the specific thread interleaving, in which execution of statements in l_{mod} immediately follows the point p_j, the set $I_{i,j}$ also contains a thread interleaving for every scheduling-relevant point p_i in l_{mod} where thread T_j is scheduled to run at p_i within that particular interleaving. This definition of the set $I_{i,j}$ covers also the case when program execution begins with the statements in the sequence l_{mod}, because all thread start events are scheduling-relevant points too. All the possible interleavings of the modified code fragment l_{mod} with the existing code of thread T_j are captured in this way. The body of the event handler onThreadSchedulingChoice in Figure 2, invoked within the scope of the procedure exploreInterleavingsForThreadPair, captures the definition of $I_{i,j}$ in an imperative style. Here, the symbol T_{cur} represents the current (running) thread in the program state when a scheduling choice was reached on the currently explored state space path. This event handler procedure is called for each program state where a thread scheduling choice needs to be created; it returns the set of all threads to be explored from that state. The backend verification tool, which is actually used to systematically explore possible interleavings for the given pair of threads, has to create all the required thread scheduling choices in the program state space, in order to ensure that all thread interleavings in the set $I_{i,j}$ are truly explored.

Note that the set $I_{i,j}$, defined above, has to include just those interleavings where thread preemption is enabled from the state in which the next instruction to be executed by thread T_i is the first instruction of the sequence l_{mod}. Therefore, all preemptive non-deterministic thread scheduling choices may be disabled on the particular analyzed execution trace in the program state space, until T_i reaches the beginning of l_{mod}.

Exploring relevant interleavings of multiple threads. In order to enable sound detection of all error states (e.g., deadlocks) that involve $N > 2$ threads, we extended our core verification algorithm (described above) such that it uses a dynamic happens-before ordering relation [8] computed over the execution trace prefix up to the first instruction (exclusively) of the modified code fragment l_{mod}. Given an execution trace prefix w, our algorithm determines the list E of relevant events (such as lock acquire, lock release, and thread join) in the prefix w, where each event in the list satisfies these two conditions: (1) it is associated with a thread other than T_i or T_j, and (2) it is not guaranteed to happen strictly before the first instruction of l_{mod} is reached by T_i. For every event $e \in E$, the algorithm has to consider both cases with respect to thread scheduling; it explores (i) the interleaving in which the instruction corresponding to e is executed before the first instruction of l_{mod} and (ii) also the complementary interleaving where the event e occurs after the end of l_{mod}, all that while preserving the program code order for all events associated with the same dynamic thread instance. The set $I_{i,j}$ is expanded to contain the additional interleavings that involve events in the list E.

3.2 Properties of the Algorithm

In this section, we discuss important properties of our verification procedure, including soundness and coverage, all that especially with respect to the ability of detecting concurrency errors.

Even though our verification algorithm explores all possible interleavings just for pairs of threads, it detects also concurrency errors that involve the modified code fragment and $N > 2$ threads. We show that, for each error in a given program, our algorithm explores at least one execution trace (interleaving of threads) leading to the corresponding error state. Our discussion focuses on three main kinds of concurrency errors — deadlocks, atomicity violations and ordering violations. We begin with deadlocks.

Deadlocks over N threads. Here we leverage the assumption that the whole program state space did not contain any deadlock before the modifications represented by l_{mod} in the code of T_i were made by developers. Execution of the modified code fragment l_{mod} by T_i is a necessary precondition for reaching a deadlock introduced through statements in l_{mod}. Therefore, we consider an arbitrary thread interleaving that reaches a deadlock state involving the set $\mathcal{T}_D$ of N threads, where $\mathcal{T}_D$ contains at least the pair (T_i, T_j) of threads for some T_j. We use the symbol π_D to denote this execution trace. Next, we explain how it is guaranteed that our verification algorithm explores the trace π_D, discovering the respective deadlock state along the way.

The trace π_D has the following three parts: (1) actions guaranteed to happen strictly before execution of the first action in the sequence l_{mod}, then (2) some interleaving of the sequence of all actions in l_{mod} executed by T_i with actions of T_j and threads in the set $\mathcal{T}_D \setminus \{T_i, T_j\}$, which can happen concurrently with l_{mod}, and finally (3) actions guaranteed to happen strictly after execution of the last action in the sequence l_{mod}. In order to detect the deadlock state reached by π_D, our verification algorithm needs to explore a thread interleaving π'_D that may differ from π_D only in the sub-sequence of independent actions within the middle part. We exploit the fact that l_{mod} contains just a single thread interaction statement st. Therefore, all other statements in l_{mod} can be moved next to st in π'_D, using the concept of left movers and right movers [10]. Given the set E of actions by threads in the set $\mathcal{T}_D \setminus \{T_i\}$, such that actions in E may happen concurrently with l_{mod}, the interleaving π'_D satisfies the property that each action in the set E is located either before the first statement of l_{mod} or after the last statement of l_{mod}. This trace π'_D is explored by our verification algorithm for some pair (T_i, T_j) of threads, and it reaches the same deadlock error state as the execution trace π_D.

Atomicity violations. In the case of atomicity violations, the problem is much simpler than for deadlocks. Given an atomicity violation (a data race) that involves a set $\mathcal{T}_{AV}$ of N threads, it can be observed when any two threads in $\mathcal{T}_{AV}$ access a shared memory location without proper synchronization. Our verification algorithm detects such data race by exploring all possible interleavings of every pair of threads in the set $\mathcal{T}_{AV}$.

Ordering violations. Finally, if some statements from the sequence l_{mod} representing the input modified code fragment are involved in an ordering violation error, then the wrong order of actions must be observed for some interleaving of the modified fragment l_{mod} executed by thread T_{mod} with some other thread T_j, i.e. for the pair (T_{mod}, T_j) of threads, and with concurrent actions by other program threads. We assume there was no ordering violation error before changes represented by l_{mod} were made.

Coverage of all thread interleavings. Another desired property of our incremental verification algorithm is the ability to cover, for the most recent version of an input program, the set of all thread interleavings that reach a concurrency error state and would be explored by full verification of the whole program state space at once.

We use an inductive approach over incremental modifications of program code to show that it is sufficient to perform systematic exploration of possible interleavings just for pairs of threads and incremental modifications during the software development process. More specifically, we show that every possibly relevant thread interleaving (i.e., that needs to be explored in order to find all errors) in the whole state space of the final (most recent) program version is covered by the pairwise approach to incremental verification for some modified code fragment at a particular step of the incremental program development.

As the base case, we assume that every relevant thread interleaving for the already existing code, that means before the modification represented by l_{mod}, is covered by our verification algorithm in some previous iteration.

Then we need to show that, for an input modified code fragment l_{mod}, our verification algorithm explores all thread interleavings that (1) contain statements from the sequence l_{mod} and (2) may reach an error state. Without loss of generality, we pick an arbitrary interleaving π that involves the modified code l_{mod}. The interleaving π can be decomposed into three segments: the prefix π_{pf}, the middle segment π_{mod} containing all of the modified code (l_{mod}) together with the set E of relevant actions by other threads (not guaranteed to be executed before the first instruction of l_{mod} or after the last instruction of l_{mod}), and the suffix π_{sf}.

From the base case assumption of inductive reasoning, we know that an execution trace created by concatenation of π_{pf} with π_E and π_{sf}, where π_E corresponds to a subsequence of π_{mod} that contains only elements of the set E, is covered by a previous run of our verification algorithm. The current run of the algorithm for the input fragment l_{mod} explores various interleavings of actions in l_{mod} executed by T_{mod} with actions in E. Note that, in each of those interleavings, some actions from the set E are executed by a thread T_j that makes a pair with T_{mod}. The middle segment π_{mod} of π corresponds to an interleaving π'_{mod} that is defined as a concatenation of three parts, specifically (i) a sequence of actions in the set E_1, (ii) actions in l_{mod}, and (iii) actions in the set E_2, where $E_1 \cup E_2 = E$. Since l_{mod} contains just a single thread interaction statement st, the interleaving π'_{mod} can be soundly transformed by changing the order of all other statements from l_{mod} in a way that matches π_{mod}. Consequently, also the interleaving π is covered by our algorithm.

4. EXPERIMENTAL EVALUATION

We have implemented our verification algorithm based on the pairwise approach in the Java Pathfinder (JPF) framework [20]. In our prototype implementation, we replaced the default JPF module for creating thread scheduling choices with our own component that precisely follows the algorithm described in Section 3.1 (exploring all interleavings just for pairs of threads), including the extension that uses a dynamic happens-before ordering relation. We also created a listener module that observes the process of state space traversal to determine whether the current program location on the currently analyzed execution trace is within the sequence of statements that represents the modified code fragment.

The complete source code, together with benchmark programs, scripts and configuration files needed to run all the experiments, is available at `https://github.com/d3sformal/incverif-pairwise`.

Our main goal for the experimental evaluation was to show (1) that incremental verification based on the pairwise approach finds bugs in the modified code very quickly, and (2) that it detects exactly the same set of errors in subject programs as full verification of the whole state space. We also wanted to show that running the incremental verification procedure after every source code modification within the process of incremental software development is more efficient than running full verification each time — despite that possible interleavings are explored for many distinct pairs of threads in each run of the incremental procedure.

Benchmarks. The set of subject programs that we used for evaluation includes 9 multithreaded Java programs from widely known benchmark collections. More specifically, the set contains five programs from the CTC repository [19] (Alarm Clock, Prod-Cons, RAX Extended, Rep Workers and SOR), two programs from the pjbench suite [21] (Cache4j and Elevator), QSort MT from the Inspect suite [18], and plain Java version of the PapaBench real-time benchmark [14].

Methodology and Setup of Experiments. In order to evaluate the performance of our incremental verification algorithm in the setting where it is run for each modified source code fragment, we needed to simulate incremental development of subject programs. The methodology that we actually used was inspired by Conway et al. [3]. We describe the main steps of the procedure that we applied to each subject program.

As the first step, all thread interaction statements in the program code are identified by static analysis. Then, for every thread interaction statement, the respective modified code fragment (that includes the affected nearby code) is determined using traversal of the list of program statements. A set $\mathcal{CF}$ of code fragments for the program is created in this way.

For each code fragment F in the set $\mathcal{CF}$ collected for the program, we performed multiple experiments with different configurations. One group of experiments was configured to simulate the scenario in which the fragment F was added into the program source code by the developer, and the other group of experiments simulates the scenario where the fragment F was deleted. Then, for each of those groups, we performed experiments focused on fast search for concurrency errors (bug finding) and experiments focused on verifying safety. Our rationale behind this setup of experiments is the following. When some piece of code (the fragment F) is added or deleted by the developer, typically several iterations of fixing bugs take place, followed by the check for safety performed at the end, before the developer moves on to the next incremental change. Therefore, we cover both scenarios in our experimental evaluation. In order to compare incremental verification with full verification of the whole program state space, we also ran full verification with the default configuration of JPF for all scenarios. For the purpose of evaluating the ability to find concurrency errors quickly, we injected synthetic errors into Prod-Cons, Cache4j, Elevator and jPapaBench. All other benchmarks already contained some errors in their code.

We configured JPF to search for deadlocks, race conditions and uncaught exceptions (including violated assertions). In the case of incremental verification, we used the time limit of 60 seconds for each run of JPF within the scope of experiments focused on bug-finding, and the limit of 10 minutes for experiments focused on safety verification. In the case of full verification, we used the time limit of 1 hour as a practical upper bound. The memory limit was set to 16 GB.

Results and Discussion. Table 1 contains the aggregate results of all our experiments focused on fast search for concurrency errors. The second column, labeled as $|\mathcal{CF}|$, shows the number of code fragments considered for each program. For both pairwise incremental verification and full verification, we report the average running time and standard deviation over all code fragments. The running time for a particular code fragment is computed as the sum of the running times of JPF for all pairs of threads. Note that, when computing the overall running times for incremental verification, we ignored the runs of JPF that did not finish within the time limit — the percentage of timed-out JPF runs is reported separately. Similarly, we report the percentage of JPF runs in the full verification mode that failed (i.e., run out of the memory or time limit).

Table 2 contains the results of experiments focused on checking safety. The meaning of all columns and metrics is the same as for the other table.

Results of experiments focused on evaluating the ability to find errors fast, provided in Table 1, show that each run of our incremental verification procedure finishes very quickly for all benchmarks and for every modified code fragment. The improvement in running time is apparent especially in the case of large and complex benchmarks (Cache4j, jPapabench).

On the other hand, data in Table 1 also shows that the performance benefits of incremental verification, when focused on finding bugs quickly, are slim for small programs, since also full verification finds errors in such programs very quickly, especially when the errors are shallow. The overhead associated with incremental verification (exploring interleavings for multiple pairs of threads after each program code change) makes the approach useful especially for large programs.

Results of experiments focused on safety verification, presented in Table 2, show that usage of our pairwise approach to incremental verification greatly improves the speed compared to full verification. In particular, full verification of safety run out of the time limit for a great majority of modified code fragments in the case of some benchmarks (such as Elevator and jPapaBench), while the pairwise incremental verification succeeded in most cases within the limited time and memory resources.

Based on thorough inspection of log files and reports by JPF, we have validated that, for every subject program in our evaluation, our incremental verification algorithm can find the same errors as if the full verification of the whole program state space is run every time. Some errors reported by JPF are caused by removal of the respective program code fragment. This corresponds to the scenario of an incomplete program still under development, where the run of incremental verification is started at a time when the source code of the subject program is not yet complete.

5. RELATED WORK

We briefly present techniques and tools that have similar goals (motivation), address the same problems, or use closely related approaches. More thorough comparison to our work is provided in the technical report [15].

The relevant techniques and tools can be divided into these categories:

- Detecting possible concurrency errors using systematic exploration of the whole space of all possible thread interleavings (cf. [13, 2]).
- Algorithmic improvements, optimizations and heuristics for addressing the problem of state explosion in the context of efficient model checking of multithreaded programs, where huge number of thread interleavings has to be analyzed even for rather small multithreaded programs. This includes different variants of partial order

Table 1: Results of experiments focused on fast search for concurrency errors (bug-finding)

		pairwise incremental verification			full verification	
program	$\lvert\mathcal{CF}\rvert$	time: avg ± dev	timed out runs	found bugs	time: avg ± dev	failed runs
Alarm Clock	56	0.25 ± 0.22 s	0 %	yes	1.20 ± 2.60 s	0 %
Prod-Cons	38	0.14 ± 0.08 s	0 %	yes	0.16 ± 0.04 s	0 %
RAX Extended	44	0.17 ± 0.23 s	0 %	yes	0.15 ± 0.05 s	1.1 %
Rep Workers	126	4.67 ± 12.14 s	3.4 %	yes	5.31 ± 29.30 s	2.8 %
SOR	56	0.09 ± 0.17 s	0 %	yes	0.83 ± 0.92 s	0 %
Cache4j	110	8.14 ± 15.06 s	0.84 %	yes	32.46 ± 283.76 s	0 %
Elevator	106	7.92 ± 16.14 s	1.67 %	yes	12.19 ± 71.40 s	1.0 %
QSort MT	71	1.01 ± 2.27 s	1.25 %	yes	0.61 ± 0.41 s	0.7 %
jPapaBench	374	15.96 ± 25.93 s	0 %	yes	28.52 ± 178.98 s	0 %

Table 2: Results of experiments with focused on checking safety (full state space traversal)

		pairwise incremental verification		full verification	
program	$\lvert\mathcal{CF}\rvert$	time: avg ± dev	timed out runs	time: avg ± dev	failed runs
Alarm Clock	56	0.25 ± 0.24 s	0 %	249.54 ± 143.48 s	0 %
Prod-Cons	38	0.16 ± 0.11 s	0 %	6.33 ± 1.82 s	0 %
RAX Extended	44	0.17 ± 0.24 s	0 %	12.71 ± 35.15 s	1.1 %
Rep Workers	126	41.32 ± 128.76 s	4.6 %	4494.43 ± 2859.57 s	14.7 %
SOR	56	0.09 ± 0.16 s	0 %	352.02 ± 74.26 s	0.9 %
Cache4j	110	46.18 ± 128.60 s	2.1 %	4936.50 ± 1987.80 s	2.7 %
Elevator	106	27.38 ± 54.99 s	0.6 %	102.96 ± 394.04 s	88.2 %
QSort MT	71	23.72 ± 92.62 s	3.8 %	735.84 ± 360.06 s	5.6 %
jPapaBench	374	18.41 ± 32.01 s	0 %	8.19 ± 168.26 s	89.3 %

reduction [5], modular reasoning with abstraction of threads [4], bounding the number of thread preemptions on each state space path (e.g., [16]), and iterative context-bounded verification [12].

- Incremental state space exploration involving model checking [9, 17] and symbolic execution [7].

- Very fast detection of concurrency errors and the corresponding bugs in the program source code, on-the-fly while developers edits the source code in some IDE. This category includes techniques based on executing the program with random thread scheduling [1] and static analysis [11].

6. ACKNOWLEDGMENTS

This work was partially supported by the Czech Science Foundation project 20-07487S and partially supported by the Charles University institutional funding project SVV 260588.

7. REFERENCES

[1] L. Blaser. 2018. Practical Detection of Concurrency Issues at Coding Time. Proceedings of ISSTA 2018, ACM.

[2] E. Clarke, O. Grumberg, and D. Peled. 2000. Model Checking. MIT Press, 2000.

[3] C.L. Conway, K.S. Namjoshi, D. Dams, and S.A. Edwards. 2005. Incremental Algorithms for Inter-procedural Analysis of Safety Properties. Proceedings of CAV 2005, LNCS 3576.

[4] C. Flanagan and S. Qadeer. 2003. Thread-Modular Model Checking. Proceedings of SPIN 2003, LNCS 2648.

[5] C. Flanagan and P. Godefroid. 2005. Dynamic Partial-Order Reduction for Model Checking Software. POPL 2005, ACM.

[6] A. Groce and W. Visser. 2004. Heuristics for Model Checking Java Programs. International Journal on Software Tools for Technology Transfer, 6(4), Springer, 2004.

[7] S. Guo, M. Kusano, and C. Wang. 2016. Conc-iSE: Incremental Symbolic Execution of Concurrent Software. ASE 2016, ACM.

[8] L. Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. Communications of the ACM, 21(7), 1978.

[9] S. Lauterburg, A. Sobeih, D. Marinov, and M. Viswanathan. 2008. Incremental State-Space Exploration for Programs with Dynamically Allocated Data. Proceedings of ICSE 2008, ACM.

[10] R.J. Lipton. Reduction: A Method of Proving Properties of Parallel Programs. Communications of the ACM, 18(12), 1975, ACM.

[11] B. Liu and J. Huang. 2018. D4: Fast Concurrency Debugging with Parallel Differential Analysis. Proceedings of PLDI 2018, ACM.

[12] M. Musuvathi and S. Qadeer. 2007. Iterative Context Bounding for Systematic Testing of Multithreaded Programs. PLDI 2007, ACM.

[13] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P.A. Nainar, and I. Neamtiu. 2008. Finding and Reproducing Heisenbugs in Concurrent Programs. In Proceedings of OSDI 2008, USENIX.

[14] F. Nemer, H. Casse, P. Sainrat, J.P. Bahsoun, and M. De Michiel. 2006. PapaBench: A Free Real-Time Benchmark. Proceedings of WCET 2006, OASIcs, volume 4.

[15] P. Parizek and F. Kliber. Incremental Verification of Multithreaded Programs by Checking Interleavings for Pairs of Threads. Technical report no. D3S-TR-2022-01, Department of Distributed and Dependable Systems, Charles University, 2022.

[16] S. Qadeer and J. Rehof. 2005. Context-Bounded Model Checking of Concurrent Software. Proceedings of TACAS 2005, LNCS 3440.

[17] G. Yang, M.B. Dwyer, and G. Rothermel. Regression Model Checking. Proceedings of ICSM 2009, IEEE CS.

[18] Y. Yang, X. Chen, and G. Gopalakrishnan. 2008. Inspect: A Runtime Model Checker for Multithreaded C Programs. Technical Report UUCS-08-004, University of Utah, 2008.

[19] Concurrency Tool Comparison repository, `https://facwiki.cs.byu.edu/vv-lab/index.php/Concurrency\_Tool\_Comparison`

[20] Java Pathfinder verification framework (JPF), `https://github.com/javapathfinder/jpf-core/wiki`

[21] Parallel Java Benchmarks, `https://bitbucket.org/psl-lab/pjbench`