Static and Precise Detection of Concurrency Errors in Systems Code Using SMT Solvers*

Shuvendu K. Lahiri[1], Shaz Qadeer[1], and Zvonimir Rakamarić[2]

[1] Microsoft Research, Redmond, WA, USA
{shuvendu,qadeer}@microsoft.com
[2] Department of Computer Science, University of British Columbia, Canada
zrakamar@cs.ubc.ca

Abstract. Context-bounded analysis is an attractive approach to verification of concurrent programs. Bounding the number of contexts executed per thread not only reduces the asymptotic complexity, but also the complexity increases gradually from checking a purely sequential program.
Lal and Reps [14] provided a method for reducing the context-bounded verification of a concurrent boolean program to the verification of a sequential boolean program, thereby allowing sequential reasoning to be employed for verifying concurrent programs. In this work, we adapt the encoding to work for systems programs written in C with the heap and accompanying low-level operations such as pointer arithmetic and casts. Our approach is completely automatic: we use a verification condition generator and SMT solvers, instead of a boolean model checker, in order to avoid manual extraction of boolean programs and false alarms introduced by the abstraction. We demonstrate the use of *field slicing* for improving the scalability and (in some cases) coverage of our checking. We evaluate our tool STORM on a set of real-world Windows device drivers, and has discovered a bug that could not be detected by extensive application of previous tools.

1 Introduction

Context-bounded analysis is an attractive approach to verification of concurrent programs. This approach advocates analyzing all executions of a concurrent program in which the number of contexts executed per thread is bounded by a given constant K. Bounding the number of contexts executed per thread reduces the asymptotic complexity of checking concurrent programs: while reachability analysis of concurrent boolean programs is undecidable, the same analysis under a context-bound is NP-complete [18, 15]. Moreover, there is ample empirical evidence that synchronization errors, such as data races and atomicity violations, are manifested in concurrent executions with small number of context switches [19, 16]. These two properties together make context-bounded analysis an effective approach for finding concurrency errors. At the same time, context-bounding provides for a useful trade-off between the cost and coverage of verification.

In this work, we apply context-bounded verification to concurrent C programs such as those found in low-level systems code. In order to deal with the complexity of low-level concurrent C programs, we take a three-step approach. First, we eliminate all the

* This work was supported by a Microsoft Research Graduate Fellowship.

complexities of C, such as dynamic memory allocation, pointer arithmetic, casts, etc. by compiling into the Boogie programming language (BoogiePL) [9], a simple procedural language with scalar and map data types. Thus, we obtain a concurrent BoogiePL program from a concurrent C program. Second, we eliminate the complexity of concurrency by appealing to the recent method of Lal and Reps [14] for reducing context-bounded verification of a concurrent boolean program to the verification of a sequential boolean program. By adapting this method to the setting of concurrent BoogiePL programs, we are able to construct a sequential BoogiePL program that captures all behaviors of the concurrent BoogiePL program (and therefore of the original C program as well) up to the context-bound. Third, we generate a verification condition from the sequential BoogiePL program and check it using a Satisfiability Modulo Theories (SMT) solver [8].

In order to scale our verification to realistic C programs, we introduce the idea of *field slicing*. The main insight is that the verification of a given property typically depends only on a small number of fields in the data structures of the program. Our algorithm partitions the set of fields into *tracked* and *untracked* fields; we only track accesses to the tracked fields and abstract away accesses to the untracked fields. This approach not only reduces the complexity of sequential code being checked, but also allows us to soundly drop context-switches from the program points where only untracked fields are accessed. Our approach is similar to localization-reduction [13], but adapted to work with arrays. We present an algorithm for refining the set of tracked fields based on the counterexample-guided-abstraction-refinement (CEGAR) loop, starting with the fields in the property of interest. Our refinement algorithm is effective; on a number of examples it discovered the field abstraction that was carefully picked by a manual inspection of the program.

We implemented our ideas in a prototype tool called STORM. We applied STORM on several real-life Windows device drivers that operate in a highly concurrent setting, and we clearly demonstrate its usability and scalability. Furthermore, we assess the effect of code size, number of contexts, and number of places where a context-switch could happen on STORM's performance. In the process, we found a bug in one of the drivers that could not be detected by extensive application of previous tools. The bug was confirmed and fixed by the driver developers.

2 Translation

In earlier work, Lal and Reps [14] presented a mechanism for transforming a multithreaded program operating on scalar variables into a sequential program, with a fixed context-bound. In this section, we show the main steps to transform a multithreaded program written in C into a sequential program, using Lal and Reps method. The input C programs support pointers, dynamic memory allocation, unbounded arrays, and low-level operations such as casts and pointer arithmetic that are prevalent in systems software. Our translation is performed in two steps:

1. Translate a multithreaded C program into a multithreaded BoogiePL program using the HAVOC tool [3]. The resultant BoogiePL program contains scalars and maps, and operations on them. The translation compiles away the complexities of C

$$
\begin{array}{ll}
\textit{Locs} & l ::= *e \mid e \rightarrow f \\
\textit{Expr} & e ::= x \mid n \mid l \mid \&l \mid e_1 \ \mathtt{op}\ e_2 \mid e_1 \oplus_n e_2
\end{array}
$$

$$
\textit{Command}\ c ::= \mathtt{skip} \mid c_1; c_2 \mid x := e \mid l := e \mid \mathtt{if}\ e\ \mathtt{then}\ c \mid \mathtt{while}\ e\ \mathtt{do}\ c
$$

Fig. 1. A simplified subset of C.

$$
\begin{array}{ll}
E(x) & = x \\
E(n) & = n \\
E(e \rightarrow f) & = \mathsf{Mem}^f[E(e) + \mathit{Offset}(f)] \\
E(*(e:\tau)) & = \mathsf{Mem}^\tau[E(e)] \\
E(\&e \rightarrow f) & = E(e) + \mathit{Offset}(f) \\
E(\&*e) & = E(e) \\
E(e_1\ \mathtt{op}\ e_2) & = E(e_1)\ \mathsf{op}\ E(e_2) \\
E(e_1 \oplus_n e_2) & = E(e_1) + n * E(e_2)
\end{array}
\qquad
\begin{array}{ll}
C(\mathtt{skip}) & = \mathsf{skip} \\
C(c_1; c_2) & = C(c_1); C(c_2) \\
C(x := e) & = x := E(e); \\
C(l := e) & = E(l) := E(e); \\
C(\mathtt{if}\ e\ \mathtt{then}\ c) & = \mathsf{if}\ E(e)\ \mathsf{then}\ C(c) \\
C(\mathtt{while}\ e\ \mathtt{do}\ c) & = \mathsf{while}\ E(e)\ \mathsf{do}\ C(c)
\end{array}
$$

Fig. 2. Translation from C into BoogiePL.

programs related to pointers, dynamic memory allocation, casts, and pointer arithmetic.

2. Translate the multithreaded BoogiePL program into a sequential BoogiePL program, for a fixed context-bound. We show how to extend Lal and Reps method to deal with programs with maps or arrays.

In the next two subsections, we describe these two steps in details.

2.1 Translating from C into BoogiePL

We present a translation of a simplified subset of C into BoogiePL programs. The translation is similar to the one presented earlier [6]; the main difference lies in splitting the C heap into multiple maps corresponding to different fields and types in the program, by assuming a *field-safe* C program — the field-safety can be justified formally in HAVOC and we explain it in this section.

Figure 1 shows a simplified subset of C for illustrating the translation from C into BoogiePL. We assume that the input program is well-typed in the original C type system. Furthermore, all structures, global variables, and local variables whose address can be taken are allocated on the heap. The field names are assumed to be unique and $\mathit{Offset}(f)$ provides the offset of a field f in its enclosing structure. For this presentation, we do not show how we handle nested structures and unions. The tool supports all the features of C programming language and details of the translation can be found in earlier work [6]. In the figure, *Locs* denotes the set of heap expressions that can be used or assigned to, and *Expr* denotes the set of C expressions. The expressions include variables (x), constants (n), *Locs* and their addresses, binary operations (such as $\leq$), and pointer arithmetic $\oplus_n$ over n-byte pointers. The language contains `skip`, sequential composition, assignments, conditional statements, and loops.

Figure 2 shows our translation from C into BoogiePL. Initially, ignore the superscript to Mem and assume there is a single Mem map. We represent the C heap using the map $\mathsf{Mem} : \mathsf{int} \rightarrow \mathsf{int}$ that maps an address to a value. The operator $E(e)$ describes the translation of a C expression e. We use $e : \tau$ to denote that τ is the static type of

e. Addresses of fields and pointer arithmetic are compiled away in terms of arithmetic operations. Finally, a dereference is translated as a lookup into the Mem map. The operator $C(c)$ translates a C statement into BoogiePL and is self-explanatory. Assignments to heap expressions result in updates to the Mem map.

The benefit of the translation with a single map Mem is that it does not rely on the types and the type-safety of a C program. However, the lack of types can make disambiguating locations in the heap difficult. For example, the following assertion cannot be proved without knowledge about the layout of the pointers `x` and `y`:

```
x->f = 1; y->g = 0; assert(x->f == 1);
```

To disambiguate heap locations, we had earlier proposed the use of a map $\mathsf{Type} : \mathsf{int} \rightarrow \mathsf{type}$ that maintains a "type" with each location in the heap [6]. A global quantified *type-safety invariant* relating the contents of Mem and Type is asserted after every update to the heap; the assertion ensures that the runtime type of a pointer corresponds to its static type. The type safety invariant helps disambiguate pointers and fields of different types, such as the pointers `&x->f` and `&y->g` in the example above.

Although the scheme described above provides an accurate memory model for C, using the type invariant while checking other properties is computationally expensive as this invariant makes use of quantifiers. Therefore, we have adopted the following strategy that provides a way for a separation of concerns. We split the Mem map into a set of maps where there is a map Mem^f for each (word-valued) field f and Mem^τ for each pointer type τ, and use the translation shown in Figure 2. We then add assertions for each memory dereference as follows: for a dereference $e \rightarrow f$ in a statement, we assert $\mathsf{Type}[E(e) + \mathit{Offset}(f)] = f$, and for a dereference $*e$, we assert $\mathsf{Type}[E(e)] = \tau$. These assertions are checked during the type-checking phase. If the assertions can be proved by the type-safety checker in HAVOC or other orthogonal techniques [21], we say that the resultant program is *field-safe* with respect to our choice of memory splits. This allows us to have a high-level (Java-like) view of the C heap while proving the concurrency related properties, without sacrificing soundness. Besides, as we show in the next section, the ability to split the C heap into independent maps allows us to perform scalable bug detection using SMT solvers.

The type-safety checker may fail to prove the introduced assertions in programs that take the address of fields in structures and dereference them directly, as in the following example:

```
x->f = 1; int *y = &x->f ; *y = 0; assert(x->f == 0);
```

In this case, the pointers `y` and `&x->f` are aliased and the type-safety checker would complain. To get around this problem, the user can specify that the maps for field f and type $\mathsf{int}*$ should be unified into a single map.[3]

2.2 Eliminating Concurrency Under a Context-Bound

The previous section showed how to convert a concurrent C program into a concurrent BoogiePL program. In this section, we show how to reduce a concurrent BoogiePL

[3] In our examples from Section 4.1, we only had to unify three fields in the `serial` driver. HAVOC automatically issued field-safety warnings, and we introduced three annotations to merge the fields (no code changes are required).

program into a sequential BoogiePL program while capturing all behaviors within a context-bound, i.e. within a certain number of contexts per thread [14].

For the rest of this section, we fix the number of threads in the input program to a positive number n and the context-bound to a positive number K. Note that the number of possible context-switches in that case is $n * K - 1$. Without loss of generality, we assume that the input concurrent program is provided as a collection of procedures containing $n+1$ distinguished procedures *Init*, T_1, ..., T_n, each of which takes no parameters and returns no value. The concurrent program is then given by $P \triangleq Init(); (T_1() || \cdots || T_n())$. Our goal is to create a sequential program Q that captures all behaviors of P up to the context-bound K. More precisely, Q will capture all round-robin schedules of P starting from thread T_1 in which each thread can execute at most K times. Each thread is allowed to stutter in each turn, thereby enabling Q to model even those schedules that are not round-robin.

The global store of the concurrent C program is captured in the BoogiePL program as a collection of global maps from integers to integers, as described in the previous section. We assume that the program has been transformed so that every statement either reads (into a local variable) or writes (from a local variable) a global map at a single index, and that the condition for every branch depends entirely on local variables. We will also assume that each such read or write to the global memory executes atomically. To model synchronization constructs, the grain of atomicity can be explicitly increased by encapsulating statements inside an atomic block. For example, the acquire operation on a lock stored at the address a is modeled using a global map variable *Lock* and a local scalar variable *tmp* as follows:

$$\mathsf{atomic}\ \{\ tmp := Lock[a];\ \mathsf{assume}\ tmp = 0;\ Lock[a] := 1;\ \}$$

Finally, we assume that assertions in the program are modeled using a special global boolean variable *error* that is set to true whenever the condition in the assert statement evaluates to false.

To convert the concurrent program P into the semantically-equivalent sequential program Q, we introduce several extra global variables. First, we introduce a global variable k to keep track of the number of contexts executed by each thread. Second, for each global map G, we introduce $K-1$ new symbolic map constants named V_2^G to V_K^G. Finally, we replace each global map G with K new global maps named G_1 to G_K. Intuitively, the sequential program Q mimics a concurrent execution of P as follows. First, each map G_i is initialized to the arbitrary symbolic constant V_i^G for all $2 \leq i \leq K$. The initialization procedure *Init* runs using the global map G_1 (with an arbitrary initial value) and initializes it. Then, the procedure T_1 starts executing using the global map G_1. Context switches in T_1 are simulated by a sequence of $K-1$ nondeterministic choices. The i-th such choice enforces that the program stops using the map G_i and starts using the map G_{i+1}. Then, each of T_2 to T_n is executed sequentially one after another under the same policy. Note that when T_{j+1} starts executing on the map G_i, the value of this map is not arbitrary; rather, its value is left there by T_j when it made its i-th context switch. Finally, when T_n has finished executing, we ensure that the final value of map G_i is equated to V_{i+1}^G, which was the arbitrary initial value of the map G_{i+1} at the beginning of the $i+1$-th context of T_1.

We capture the intuition described above by performing the following transformations in sequence:

1. Replace each statement of the form $tmp := G[a]$ with

$$\begin{array}{l}\mathsf{atomic}\ \{\\ \quad \mathsf{if}\ (k=1)\ tmp := G_1[a]\\ \quad \mathsf{elsif}\ (k=2)\ tmp := G_2[a]\\ \quad \ldots\\ \quad \mathsf{else}\ tmp := G_K[a]\\ \}\end{array}$$

   and each statement of the form $G[a] := tmp$ with

$$\begin{array}{l}\mathsf{atomic}\ \{\\ \quad \mathsf{if}\ (k=1)\ G_1[a] := tmp\\ \quad \mathsf{elsif}\ (k=2)\ G_2[a] := tmp\\ \quad \ldots\\ \quad \mathsf{else}\ G_K[a] := tmp\\ \}\end{array}$$

2. After each atomic statement that is not within the lexical scope of another atomic statement, insert a call to procedure *Schedule* with the following specification:

$$\begin{array}{l}\mathsf{modifies}\ k\\ \mathsf{ensures}\ \mathsf{old}(k) \le k \wedge k \le K\\ \mathsf{exsures}\ \mathsf{true}\\ \mathsf{void}\ \mathit{Schedule}(\mathsf{void});\end{array}$$

   Here, exsures true means that *Schedule* may terminate either normally or exceptionally; under normal termination, k is incremented by an arbitrary amount but remains within the context-bound K. The possibility of incrementing k by more than one allows the introduction of stuttering into the round-robin schedules. The possibility of exceptional termination allows a thread to stop executing at any point. The raised exception is caught by handlers (as shown below) that wrap the invocation of each T_i. We assume that *Init* does not share any code with the threads and we do not add a call to *Schedule* to any of the procedures called from *Init*.

For each procedure f, let the procedure obtained by the transformation above be denoted by f'. Let us assume that there is a single map variable G in the original program. The sequential program Q is then defined to be as follows:

$$\begin{array}{l}G_2 := V_2^G;\ \ldots;\ G_K := V_K^G;\\ \mathit{Init}();\\ \mathit{error} := \mathit{false};\ k := 1;\\ \mathsf{try}\ \{\ \mathit{Schedule}();\ T_1'()\ \}\ \mathsf{finally}\ k := 1;\\ \ldots\\ \mathsf{try}\ \{\ \mathit{Schedule}();\ T_n'()\ \}\ \mathsf{finally}\ k := 1;\\ \mathsf{assume}\ G_1 = V_2^G;\ \ldots;\ \mathsf{assume}\ G_{K-1} = V_K^G;\\ \mathsf{assert}\ \neg \mathit{error}\end{array}$$

Note that all constraints involving the symbolic map constants are *assumed* equalities. These equalities can be handled by the select-update theory of arrays without requiring the axiom of extensionality. Consequently, these constraints do not present any impediment to the use of an off-the-shelf SMT solver. The transformed program contains control flow due to exceptions which can be easily compiled away if the underlying verification-condition generator does not understand it. Furthermore, since the transformed program is sequential, the verification-condition generator can ignore the atomic annotations in the code.

3 Field Slicing

Once we have the sequential BoogiePL program generated from the multithreaded C program, the next step is to try to verify the program using BOOGIE. BOOGIE performs precise reasoning across loop-free and call-free code, but needs loop invariants and procedure contracts to deal with loops and procedure calls modularly. In order to have an automatic tool, we inline procedures and unroll loops (with some exception discussed later).[4] Since recursion is rare in system programs, inlining procedures is acceptable; however, the size of inlined procedures can be very large. Our initial attempt at verifying these inlined programs did not succeed. On the other hand, we may lose coverage when we unroll loops a fixed number of times. In this section, we illustrate the use of a simple *field slicing* technique to achieve scalability when checking large inlined call-free programs without sacrificing precision; in some cases, our method enables us to avoid unrolling loops and therefore obtain greater coverage.

3.1 Abstraction with Tracked Fields

The high-level idea of this section is fairly simple: our translation of C programs described in Section 2.1 uses a map Mem^f for dereferencing a field f, and a map Mem^τ for dereferencing pointers of type τ. We assume that the input C program has been proven *field-safe* for this split, i.e. the type checker has verified the assertions about the Type map as described earlier. We guess a subset of these fields and types as *relevant* and slice the program with respect to these fields. If the abstracted program can be proved correct, then we have proved the correctness of the sequential BoogiePL program. Otherwise, we have to *refine* the set of relevant fields and try again. While proving the abstracted program, we can skip loops (without the need to unroll them) that do not modify any of the relevant fields.

In this section, we formalize how we perform the slicing with respect to a set of fields, while in the next section we show how to refine the set of fields we track. Let us define the operation $\texttt{Abstract}(P, F)$ that takes a BoogiePL program P generated in the last section and a set of fields F, and performs the following operations:

1. For any field $g \notin F$, translate the writes $\mathsf{Mem}_i^g[e] := tmp$ for all $1 \le i \le K$ as skip.
2. For any field $g \notin F$, translate the reads $tmp := \mathsf{Mem}_i^g[e]$ for all $1 \le i \le K$ as havoc tmp, which scrambles the value of tmp.

[4] Inference of loop invariants and procedure contracts is an important area of future work.

```
Input: Program P
Output: Program P checked or error trace
 1: allFields ← all fields in P
 2: trackedFields ← ∅
 3: loop
 4:    A ← Abstract(P, trackedFields)
 5:    (checked, absErrTrace) ← Check(A)
 6:    if checked = true then
 7:       return CHECKED
 8:    else
 9:       concTrace ← Concretize(P, absErrTrace)
10:       checked ← Check(concTrace)
11:       if checked = true then
12:          F ← allFields
13:          for all f ∈ allFields do
14:             absTrace ← Abstract(concTrace, trackedFields ∪ F \ {f})
15:             checked ← Check(absTrace)
16:             if checked = true then
17:                F ← F \ {f}
18:             else
19:                trackedFields ← trackedFields ∪ {f}
20:             end if
21:          end for
22:       else
23:          return BUG(concTrace)
24:       end if
25:    end if
26: end loop
```

Fig. 3. Algorithm for tracked fields refinement based on the CEGAR loop.

3. Finally, remove the call to *Schedule* that was inserted after the atomic section for a read or write from a field $g \notin F$.

It is easy to see that the first two steps are property-preserving, i.e. they do not result in missed bugs. Since statements such as havoc *tmp* and skip do not access any global state, context switches after them will not introduce any extra behavior. Consequently, the trailing calls to *Schedule* can be removed, thereby eliminating a significant number of redundant context switches.

In addition to reducing code size and eliminating context switches, checking the abstraction $\texttt{Abstract}(P, F)$ has another benefit. It enables us to create simple summaries for loops whose body does not contain any reads or writes from F. The summary leaves the memory maps unchanged and puts nondeterministic values into the local variables modified by the loop. This simple heuristic for creating loop summaries is precise enough for our examples: it worked for 5 out of a total of 15 loops in our benchmarks from Section 4.1.

Both of these factors improve the scalability of our approach and improve coverage by not requiring every loop to be unrolled. In particular, we can avoid the problem with

unrolling loops whose exit condition does not depend on any input values (e.g. a loop that goes from 1 to 100) — for such loops any unrolling less than 100 times would block the execution after the loop.

3.2 Refining Tracked Fields

In this section, we provide an algorithm for inferring the set of relevant fields that affect the property being checked. Our inference algorithm is a variant of the counterexample-guided abstraction refinement (CEGAR) framework [5, 13]. Figure 3 gives the pseudo-code for the algorithm. The algorithm takes a program *P* and checks if the assertion in the program holds. We start with initializing *trackedFields* with an empty set, and then we add fields to the set based on the analysis of counterexamples. The outer loop in lines 3 to 26 refines *trackedFields* from a single abstract counterexample *absErrTrace* obtained by checking the abstract program *A*. If the abstract program *A* is not correct, we concretize the abstract counterexample trace *absErrTrace* and check if the trace is spurious. If the trace is not spurious, then we have a true error in line 23. The operation `Concretize` simply restores the reads and writes of fields that were abstracted away (we do not add the context switches back, although adding them would not break the algorithm). The inner loop in lines 13 to 21 greedily finds a minimal set of fields from *allFields* such that abstracting them would result in a spurious counterexample. Those fields are added to *trackedFields* and the outer loop is iterated again. Since each iteration of the inner loop increases the size of *trackedFields* and the total number of fields is finite, the algorithm terminates.

4 Implementation and Results

In this section, we describe our prototype implementation STORM, and our experience with applying the tool on several real-life benchmarks. As described earlier, STORM first uses HAVOC to translate a multithreaded C program along with a set of relevant fields into a multithreaded BoogiePL program (Section 2.1), then reduces it to a sequential BoogiePL program (Section 2.2), and finally uses BOOGIE to check the sequential program. The BOOGIE verifier [2] generates a verification condition from the BoogiePL description, and uses the SMT solver Z3 [8] to check the resulting verification condition.

4.1 Benchmarks

We evaluated STORM on a set of real-world Windows device driver benchmarks. Table 1 lists the device drivers used in our experiments and the corresponding driver dispatch routines we checked. It also provides their size, total number of fields, number of threads, and the scenario in which they are checked. STORM found a bug in the `usbsamp` driver (see Section 4.3) and `usbsamp_fix` is the fixed version of the example.

We implemented a common harness for putting device drivers through different concurrent scenarios. Each driver is checked in a scenario possibly involving concurrently executing driver dispatch routines, driver request cancellation and completion routines,

Driver	LOC	Routine	#F	#T	Scenario
`daytona`	105	`ioctl`	53	2	D | CA

`mqueue`	494	`read`
		`write`
		`ioctl`
			72	4	D | CA | CP | DPC

`usbsamp`	644	`read`
		`write`
		`ioctl`
			113	3	D | CA | CP

`usbsamp_fix`	643	`read`
		`write`
		`ioctl`
			113	3	D | CA | CP

`serial`	1089	`read` `write`	214	3	D | CA | DPC

Table 1. Windows device drivers used in the experiments. "LOC" is the bare number of lines of code in the scenarios we check, excluding whitespaces, comments, variable and function declarations, etc.; "Routine" lists the dispatch routines we checked; "#F" gives the total number of fields; "#T" is the number of threads in the checked scenario; "Scenario" shows the concurrent scenario being checked, i.e. which driver routines are executed concurrently as threads (D – dispatch routine, CA – cancel routine, CP – completion routine, DPC – deferred procedure call).

and deferred procedure calls (column "Scenario" in Table 1). The number of threads and the complexity of a scenario depend on the given driver's capabilities. For example, for the `usbsamp` driver, the harness executes a dispatch, cancel, and completion routine in three threads. Apart from providing a particular scenario, our harness also models synchronization provided by the device driver framework, as well as synchronization primitives, such as locks, that are used for driver-specific synchronization.

STORM has the ability to check any user-specified safety property. In our experiments, we checked the *use-after-free* property for the `IRP` (*IO Request Packet*) data structure used by the device drivers. A driver may complete and free an `IRP` it receives by calling a request completion routine (e.g. `WdfRequestComplete` in Figure 4), and must not access an `IRP` object once it has been completed. To check this property, we introduced assertions via automatic instrumentation before each access to an `IRP` object; our examples have up to a hundred of such assertions. Typically, drivers access and may complete the same request in multiple routines executing concurrently. To satisfy our crucial *use-after-free* property, the code must follow the proper and often complex synchronization protocol. Bugs often manifest only in highly concurrent scenarios; consequently, this property is difficult to check with static analysis tools for sequential programs.

4.2 Evaluation

Our empirical evaluation of STORM consists of two sets of experiments. In the first one (Table 2 and Table 3), we run STORM on the benchmarks described in the previous section using manually provided fixed set of tracked fields. We assess the scalability of STORM with respect to code size, number of threads, number of contexts, and number of locations where a context switch could potentially happen. In the second set of ex-

Example	Routine	# of contexts per thread				
		1	2	3	4	5
daytona	ioctl	3.4	3.8	4.2	4.5	5.6
mqueue	read	62.1	161.5	236.2	173.0	212.4
	write	48.6	113.4	171.2	177.4	192.3
	ioctl	120.6	198.6	204.7	176.1	199.9
usbsamp	read	17.9	37.7	65.8	66.8	85.2
	write	17.8	48.8	52.3	74.3	109.7
	ioctl	4.4	5.0	5.1	5.3	5.4
usbsamp_fix	read	16.9	28.2	38.6	46.7	47.5
	write	18.1	32.2	46.9	52.5	63.6
	ioctl	4.8	4.7	5.1	5.1	5.2
serial	read	36.5	95.4	103.4	240.5	281.4
	write	37.3	164.3	100.8	233.0	649.8

Table 2. Varying the number of contexts per thread.

Example	Routine	#CS	% of switches removed			
			0	40	80	100
daytona	ioctl	26	3.9	3.7	3.6	3.5
mqueue	read	201	161.1	121.3	112.1	57.8
	write	198	112.7	101.5	100.6	25.2
	ioctl	202	197.7	192.8	168.5	73.1
usbsamp	read	90	37.7	42.2	*22.6	*17.9
	write	90	48.9	37.7	*22.7	*18.9
	ioctl	22	5.0	4.8	4.5	4.4
usbsamp_fix	read	89	28.2	25.9	22.6	17.0
	write	89	32.2	28.2	22.5	16.5
	ioctl	21	4.7	4.7	4.5	4.3
serial	read	307	95.4	92.7	66.3	47.6
	write	309	164.8	120.2	94.3	29.7

Table 3. Varying the number of locations where a context switch could happen. The number of contexts per thread is fixed to 2. "CS" represents the total number of places where a context switch could happen. The examples where we missed the usbsamp bug because of randomly (unsoundly) removing context switch locations are marked with *.

periments (Table 4), instead of using manually provided tracked fields, we determine the usability of our tracked fields refinement algorithm by using it to completely automatically check our benchmark drivers. All experiments were conducted on an Intel Pentium D at 2.8GHz running Windows XP, and all runtimes are in seconds.

Table 2 shows the result of varying the number of contexts per thread from 1 (sequential case) to 5. We managed to successfully check all of our benchmarks with up to 5 contexts per thread, which clearly demonstrates the scalability of our approach. In the process, our tool discovered a bug in the usbsamp driver (details can be found in Section 4.3).

Table 3 demonstrates how the runtimes vary with the number of places in the code where a context switch can be introduced. For the usbsamp example that has a bug, removing the context switches results in the bug not being discovered. The runtime decreases as the number of context-switch locations decreases. This observation justifies that removing context switches during field slicing is important for scalability.

Table 4. Results of the tracked fields refinement algorithm. "#F" gives the total number of fields; "#MF" is the number of manually provided tracked fields; "#AF" denotes the number of tracked fields generated by the refinement algorithm; "#IT" is the number of CEGAR loop iterations; "Time" is the total running time.

Example	Routine	#F	#MF	#AF	#IT	Time(s)
daytona	ioctl	53	3	3	3	244.3
mqueue	read	72	7	9	9	3446.3
	write			8	8	3010.0
	ioctl			9	9	3635.6
usbsamp_fix	read	113	1	3	3	4382.4
	write			4	4	2079.2
	ioctl			0	0	21.7
serial	read	214	5	5	5	3013.7
	write			4	3	1729.4

Table 4 describes the results of applying the abstraction-refinement algorithm from Section 3 to discover the set of relevant fields and completely automatically check the examples. Using the refinement algorithm, we were always able to obtain a set of relevant fields that is just a small fraction of the set of all fields and that closely matches the set of manual fields that we used previously. Most of the runtime is actually spent in scripts to perform the abstraction, and can be significantly reduced. Without the use of field slicing, STORM was unable to run on large examples. For example, even checking the `mqueue read` routine with only two contexts does not terminate in one hour if we do not use field slicing.

4.3 Bug Found

By applying STORM on the Windows device drivers listed in Table 1, we found a concurrency bug in the `usbsamp` driver. We reported the bug, and driver developers confirmed and fixed it. Figure 4 illustrates the bug with a simplified code excerpt from the driver. It contains two routines, the `UsbSamp_EvtIoRead` dispatch routine and the `UsbSamp_EvtRequestCancel` cancellation routine. The routines get executed by threads `T1` and `T2`, respectively. The example proceeds as follows:

1. Thread `T1` starts executing on a request `Request`, while thread `T2` is blocked since cancellation for `Request` has not been enabled.
2. `T1` enables cancellation and sets the cancellation routine with the call to the driver framework routine `WdfRequestMarkCancelable` on line 8. Then the context switch on line 10 occurs.
3. `T2` can now start executing `UsbSamp_EvtRequestCancel`, and another context switch happens on line 7 of `T2`.
4. `T1` completes `Request` on line 11 and context switches again on line 12.
5. On line 9, `T2` tries to access `Request` that has been completed in the previous step, which is an error.

It is important to note that although the scenario leading to this bug might seem simple, the bug has not been found before by extensively applying other software checkers on `usbsamp`. For instance, SLAM [1] failed to discover this bug since SLAM can check only sequential code. KISS [19], on the other hand, can check concurrent code, but only

```
1 // Thread T1
2 VOID UsbSamp_EvtIoRead(
3     WDFQUEUE   Queue,
4     WDFREQUEST Request,
5     size_t     Length
6   ) {
7   ...
8   WdfRequestMarkCancelable(
9     Request, UsbSamp_EvtRequestCancel);
10  ... // SWITCH 1: T1->T2
11  WdfRequestComplete(Request, status);
12  ... // SWITCH 3: T1->T2
13 }
```

```
1 // Thread T2
2 VOID
3 UsbSamp_EvtRequestCancel(
4     WDFREQUEST Request
5   ) {
6   PREQUEST_CONTEXT rwContext;
7   ... // SWITCH 2: T2->T1
8   rwContext =
9     GetRequestContext(Request);
10  ...
11 }
```

Fig. 4. Simplified version of the code illustrating the concurrency bug STORM found in the `usbsamp` example. Places where context switches happen when the bug occurs are marked with `SWITCH`.

up to 2 context switches, and would therefore also miss this bug since the bug occurs only after at least 3 context switches.

5 Related Work

We roughly divide the related work into two areas — bounded approaches to concurrency and other techniques for analysis of concurrent C programs.

Bounded approaches to concurrency. The idea of context-bounded analysis of concurrent programs was proposed by Qadeer and Wu [19], and later context-bounded reachability analysis for concurrent boolean programs was shown to be decidable [18]. Many subsequent approaches have relied on bounding the number of contexts to tackle the complexity and scalability issues of concurrent program analysis [18, 19, 16, 20, 14].

KISS [19] transforms a concurrent program with up to two context switches into a sequential one by mimicking context switches using procedure calls. However, restricting the number of context switches can be limiting, as evidenced by the bug in Section 4.3 that STORM discovered.

Rabinovitz and Grumberg [20] propose a context bounded verification technique for concurrent C programs based on bounded model checking and SAT solving. The algorithm applies traditional BMC on each thread separately and generates sets of constraints for each. The constraints are instrumented to account for concurrency, by introducing copies of global variables and additional constraints for context switches. The resulting formula is solved by a SAT solver. Our work offers several important advantages: we support memory maps to deal with a possibly unbounded heap; our source-to-source program transformation allows us to leverage any sequential verification technique, including annotation-based modular reasoning; our experiments are performed on real-world benchmarks, whereas the authors apply the technique to hand-crafted microbencmarks. Finally, it is unclear how to exploit techniques such as field slicing using their method.

Bounded model checking of concurrent programs was also explored by Ganai and Gupta [10], where concurrency constraints are added lazily and incrementally during

bounded unrolling of programs. The number of context switches is not bounded a priori, but heap and stack are, and the number of program steps the bounded model checker explores is limited by the available resources.

Suwimonteerabuth et al. [22] present a context-bounded analysis of multithreaded Java programs. Their approach is different from ours because it translates a multithreaded Java program to a concurrent pushdown system by bounding the size of the program heap and using finite bitvector encoding for integers.

CHESS [16] is a tool for testing multithreaded programs that dynamically explores thread interleavings by iteratively bounding the number of contexts. On the other hand, STORM is a static analysis tool and therefore does not have to execute the code using tests and offers more coverage since it explores all possible paths in a program up to a given context bound.

Analysis of concurrent C programs. Kahlon et al. [12] focus their efforts on iteratively reducing the number of thread interleavings using invariants generated by abstract interpretation. The described techniques are complementary to our approach, since we could also use them to reduce the number of interleavings in our instrumented program. The authors then apply model checking, but only on program slices in order to resolve data-race warnings, and therefore fair comparison with our experiments would be hard.

Witkowski et al. [23] describe their experience with applying CEGAR-based predicate abstraction on concurrent Linux device drivers. Their results indicate that concurrency rapidly increases the number of predicates inferred by the refinement loop, which in turn causes a fast blow-up in the model checker. Before we derived our current technique based on SMT solvers, we attempted a similar approach where we used the Lal-Reps method to create a source-to-source transformation from a multithreaded to a sequential C program, which is then analyzed by the SLAM [1] verifier. Our experience was similar as we could not scale this approach beyond even simple microbenchmarks. Henzinger et al. [11] present a more scalable approach for CEGAR-based predicate abstraction of concurrent programs; their method checks each thread separately in an abstract stateful context that is iteratively constructed by a refinement loop.

Chugh et al. [4] introduce a framework for converting a sequential dataflow analysis into a concurrent one using a race detection engine. The race detection engine is used to ensure soundness of the sequential analysis by invalidating the dataflow facts influenced by concurrent writes. The analysis is scalable, but yields many false positives; our approach is much more precise, but not as scalable.

There also exists work that targets analysis of concurrent boolean program models [7, 17]. However, these approaches do not clarify how to obtain these models from real-world programs, while our approach can automatically analyze C programs.

References

1. T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *Conf. on Programming Language Design and Implementation (PLDI)*, pages 203–213, 2001.
2. M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Intl. Symp. on Formal Methods for Objects and Components (FMCO)*, pages 364–387, 2005.

3. S. Chatterjee, S. K. Lahiri, S. Qadeer, and Z. Rakamarić. A reachability predicate for analyzing low-level software. In *Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 19–33, 2007.
4. R. Chugh, J. W. Voung, R. Jhala, and S. Lerner. Dataflow analysis for concurrent programs using datarace detection. In *Conf. on Programming Language Design and Implementation (PLDI)*, pages 316–326, 2008.
5. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Conf. on Computer Aided Verification (CAV)*, pages 154–169, 2000.
6. J. Condit, B. Hackett, S. K. Lahiri, and S. Qadeer. Unifying type checking and property checking for low-level code. In *Symp. on Principles of Programming Languages (POPL)*, pages 302–314, 2009.
7. B. Cook, D. Kroening, and N. Sharygina. Symbolic model checking for asynchronous boolean programs. In *Intl. SPIN Workshop on Model Checking Software*, pages 75–90, 2005.
8. L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 337–340, 2008.
9. R. DeLine and K. R. M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research, 2005.
10. M. K. Ganai and A. Gupta. Efficient modeling of concurrent systems in BMC. In *Intl. SPIN Workshop on Model Checking Software*, pages 114–133, 2008.
11. T. A. Henzinger, R. Jhala, and R. Majumdar. Race checking by context inference. In *Conf. on Programming Language Design and Implementation (PLDI)*, pages 1–13, 2004.
12. V. Kahlon, S. Sankaranarayanan, and A. Gupta. Semantic reduction of thread interleavings in concurrent programs. In *Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 124–138, 2009.
13. R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton University Press, 1995.
14. A. Lal and T. W. Reps. Reducing concurrent analysis under a context bound to sequential analysis. In *Conf. on Computer Aided Verification (CAV)*, pages 37–51, 2008.
15. A. Lal, T. Touili, N. Kidd, and T. W. Reps. Interprocedural analysis of concurrent programs under a context bound. In *Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 282–298, 2008.
16. M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *Conf. on Programming Language Design and Implementation (PLDI)*, pages 446–455, 2007.
17. G. Patin, M. Sighireanu, and T. Touili. Spade: Verification of multithreaded dynamic and recursive programs. In *Conf. on Computer Aided Verification (CAV)*, pages 254–257, 2007.
18. S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. In *Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 93–107, 2005.
19. S. Qadeer and D. Wu. KISS: Keep it simple and sequential. In *Conf. on Programming Language Design and Implementation (PLDI)*, pages 14–24, 2004.
20. I. Rabinovitz and O. Grumberg. Bounded model checking of concurrent programs. In *Conf. on Computer-Aided Verification (CAV)*, pages 82–97, 2005.
21. Z. Rakamarić and A. J. Hu. A scalable memory model for low-level code. In *Conf. on Verification, Model Checking and Abstract Interpretation (VMCAI)*, pages 290–304, 2009.
22. D. Suwimonteerabuth, J. Esparza, and S. Schwoon. Symbolic context-bounded analysis of multithreaded Java programs. In *Intl. SPIN Workshop on Model Checking Software*, pages 270–287, 2008.
23. T. Witkowski, N. Blanc, G. Weissenbacher, and D. Kroening. Model checking concurrent Linux device drivers. In *Intl. Conf. on Automated Software Engineering (ASE)*, pages 501–504, 2007.