Memory Model Sensitive Bytecode Verification[*]

Thuan Quang Huynh[†]
Department of Computer Science
University of Maryland College Park
thuan@cs.umd.edu

Abhik Roychoudhury
Department of Computer Science
National University of Singapore
abhik@comp.nus.edu.sg

Abstract

Modern concurrent programming languages like C# and Java have a programming language level memory model, which captures the set of all allowed behaviors of programs on any implementation platform — uni- or multi-processor. Such a memory model is typically weaker than Sequential Consistency and allows reordering of operations within a program thread. Therefore, programs verified correct by assuming Sequential Consistency (that is, each thread proceeds in program order) may not behave correctly on certain platforms! The solution to this problem is to develop program checkers which are memory model sensitive. In this paper, we develop a bytecode level invariant checker for the programming language C#. Our checker identifies program states which are reached only because the C# memory model is more relaxed than Sequential Consistency. It employs partial order reduction strategies to speed up the search. These strategies are different from standard partial order reduction methods since our search also considers execution traces containing bytecode re-orderings. Furthermore, our checker identifies (a) operation re-orderings which cause such undesirable states to be reached, and (b) simple program modifications — by inserting memory barrier operations — which prevent such undesirable re-orderings.

1 Introduction

Modern mainstream programming languages like C# and Java support multi-threading as an essential feature of the language. In these languages multiple threads can access shared objects. Moreover, synchronization mechanisms exist for controlling access to shared objects by threads. If every access to a shared object by any thread requires prior acquisition of a common

[*]A preliminary version of this paper appeared as [13]. The conference paper is available from `http://www.comp.nus.edu.sg/∼abhik/pdf/fm06.pdf`

[†]This work was done when the first author was a Research Assistant at National University of Singapore.

lock, then the program is *guaranteed* to be "properly synchronized". On the other hand, if there are two accesses to a shared object/variable v by two different threads, at least one of them is a write, and they are not ordered by synchronization — the program is then said to contain a data race, that is, the program is *improperly synchronized*. Improperly synchronized programs are common for more than one reason — (a) programmers may want to avoid synchronization overheads for low-level program fragments which are executed frequently, (b) programmers may forget to add certain synchronization operations in the program, or (c) programmers forget to maintain a common lock guarding accesses to some shared variable v since there are often many lock variables in a real-life program.

Problem Statement The work in this paper deals with formal verification (and subsequent debugging) of multi-threaded C# programs which are improperly synchronized. As a simple example consider the following schematic program fragment, and suppose initially `x = y = 0`. Moreover `l1`, `l2` are thread-local variables while `x`, `y` are shared variables.

```
x = 1;      |      l1 = y;
y = 1;      |      l2 = x;
```

If this program is executed on a uni-processor platform, we cannot have `l1 = 1, l2 = 0` at the end of the program. However, on a multiprocessor platform which allows reordering of writes to different memory locations this is possible. On such a platform, the writes to `x`, `y` may be completed out-of-order. As a result, the following completion order is possible $\langle \texttt{y} = \texttt{1}, \texttt{l1} = \texttt{y}, \texttt{l2} = \texttt{x}, \texttt{x} = \texttt{1} \rangle$.

Since an improperly synchronized program can exhibit different sets of behaviors on different platforms, how do we even specify the semantics of such programs and reason about them? Clearly, we would like to reason about programs in a *platform-independent* fashion, rather than reasoning about a program's behaviors separately for each platform. Languages like Java, C# allow such platform-independent reasoning by defining a *memory model* at the programming language level. Now, what does a memory model for a programming language like C# mean? The C# memory model (also called the .NET memory model [21]) is a set of abstract rules which capture the behaviors of multi-threaded programs on *any* implementation platform — uni-processor or multi-processor. Given a multi-threaded C# program P, the set of execution traces of P permitted under the .NET memory model is a superset of the traces obtained by interleaving the operations of program P's individual threads. The operations in any thread include read/write of shared variables and synchronization operations like lock/unlock. The .NET memory model permits *certain operations* within a thread to be completed out-of-order, that is, the programming language level memory model essentially specifies which reorderings are allowed. So, to consider all program behaviors we need to take into account — (a) arbitrary interleavings of threads, and (b) certain (not all) reorderings within a thread. This makes the formal verification of improperly synchronized multi-threaded programs especially hard.

Basic Approach In this paper, we develop a memory-model sensitive invariant checker for the programming language C#. Our checker verifies a C# program at the level of bytecodes. The checker proceeds by representing and managing states at the level of C#'s stack-based virtual machine. *Even though the approach is illustrated for the C# programming language, any language which can get compiled into CLI bytecode can be handled by our method.* Moreover, the checker's state space exploration takes the .NET memory model into account. In other words, it allows the reorderings permitted by .NET memory model to explore additional reachable states in a program. Thus, the programming language level memory model is treated as a formal contract between the program and the language implementation; we then take this contract into account during software verification.

Furthermore, we note that programmers usually understand possible behaviors of a multi-threaded program by using a stronger model called *Sequential Consistency* [16]. An execution model for multithreaded programs is sequentially consistent if for any program P (a) any execution of P is an interleaving of the operations in the constituent threads (b) the operations in each constituent thread execute in program order. Thus, if we are model checking an invariant φ, our checker may uncover counter-example traces which (a) violate φ , and (b) are not allowed under Sequential Consistency. Disallowing such counter-example traces requires disabling reorderings among operations. This is usually done by inserting *memory barriers or fence operations*; a memory barrier is an operation such that instructions before the barrier must complete before the starting of instructions after the barriers. Since memory barriers are expensive operations (in terms of performance) we use a maxflow-mincut algorithm to insert minimal number of barriers/fences for ruling out program states which are unreachable under Sequential Consistency.

Technical Contributions Our work involves the following steps — which taken together constitute the technical contributions of this paper.

- *Memory Model Specification* We first understand and formally specify the .NET memory model. Previous works [30] have investigated this issue and discussed certain corner cases in the .NET memory model description. Unlike [30], our specification is not operational/executable, making it more accessible to system designers (who may not have formal methods background).

- *The Checker* We use the .NET memory model specification to develop a memory model sensitive invariant checker at the level of bytecodes. It allows all execution traces permitted by .NET memory model. The checker issues operations in program order but allows them to complete out-of-order as long as the reordering is permitted by .NET memory model. We integrate partial-order reduction methods into our checker to combat state space explosion. This involves extending conventional partial order reduction methods developed for verifying sequentially consistent executions of concurrent programs.

- *Memory Barrier Insertion* Our checker is useful for uncovering all execution traces allowed by the .NET memory model. However, when the programmer finds "unexpected" execution traces using our checker how does (s)he disallow this behavior? We use the well-known maxflow-mincut algorithm [11] to rule out program states unreachable under Sequential Consistency. The min-cut yields (a minimal number of) places in the program where the memory barriers are to be inserted.

In Section 3 we show a simple working example to explain our identification and removal of undesirable program behaviors.

Differences from conference paper The main differences between this paper and its conference version [13] lie in enhancing the utility and the applicability of our work. Towards this goal, we have given elaboration of the issues in constructing a formal description of the C# memory model. Some of these issues were discussed in details while explaining our formal specification to readers during/after the conference. This discussion appears in Section 4 of the paper.

Furthermore, in the journal version, we have integrated partial-order reduction strategies into our checker (see Section 6). This could not be done straightforwardly since the correctness of partial order reduction is not proved for search strategies where execution traces formed by operation re-orderings are also explored. Note that partial order reduction is a state-space reduction technique which is typically employed in explicit-state model checking of asynchronous concurrent systems [6]. It avoids exploring all possible interleavings of different program threads by calculating at each program state an "ample set" of enabled actions; this ample set is a subset of all enabled actions from the state and hence all execution traces emanating from the state need not be explored. Now, when some of the enabled actions from a program state involve re-ordering of operations in a thread (instead of executing the next statement in the thread), how do we calculate the ample set of enabled actions? We show how this can be done and integrate it into our memory model sensitive checker for C#. We evaluate the checker against standard benchmarks (reader-writer, producer-consumer etc) which have been used by other recent works on software model checking with partial-order reduction. Our contributions are not necessarily C# specific — our results can be used for memory model sensitive software model checking.

2 Related Work

Developing formal specification of memory models has been well-studied in the context of hardware multiprocessors. Similar to Java threads, hardware shared-memory multiprocessors also impose a consistency model which dictates the allowed interactions among the processors via a shared memory. Dill et. al. [8, 25] developed executable memory models for SPARC architectures and used them to verify synchronization routines. Gopalakrishnan et. al. [12, 32] have used SAT solving to check whether an execution is allowed by a multi-processor memory model, in particular the Intel Itanium memory model. We are addressing a different problem in this paper. Instead of checking whether one given execution is allowed, our work automatically generates the transition system (capturing all possible executions) for a given program under the C# memory model. This is made possible because of the "local" manner in which we specify the C# memory model — it is simply specified as a table describing which operation pairs can be re-ordered. Consequently, we can employ these re-orderings on-the-fly while constructing and traversing the transition system (for a given program). It would be interesting to see whether the rules of the Intel Itanium model can be employed in a similar "constructive" fashion to construct all possible executions for a given program.

In recent works, Burckhardt, Alur and Martin [4, 5] study bounded model checking of concurrent data types under relaxed hardware memory models. Their work uses a SAT checker to verify observational equivalence between sequentially consistent behaviors of a concurrent program and behaviors allowed by relaxed hardware memory models. Their work is related to our approach, but differs in two respects. First, we focus on a programming language level memory model (that of C#) rather than hardware memory models. Even though hardware and software memory models can be specified in a common framework, the level at which the verification is carried out (source/bytecode/instructions) can be different. Since Burckhardt et. al. consider hardware memory models, the program for a thread is converted to instruction sequences. In our case, we perform verification at the bytecode level. Secondly, we give an automatic method to insert optimal number of fences/barriers. Thus, not only can we check whether a given set of fences is necessary and sufficient to prove an invariant under C#'s memory model, we can also compute the minimal number of fences required to ensure an invariant property under C#'s memory model.

Programming language level memory models are relatively new. In the recent years, substantial research efforts have been invested in developing the Java Memory Model (*e.g.* see [1, 17, 19]). These works mostly focus on what should be the programming language level memory model for Java.

For the .NET memory model, a formal executable specification based on Abstract State Machines has been discussed in [30]. In this paper, we formally present the .NET memory model in a tabular non-operational format — clearly showing which pairs of operations can be reordered. This makes the formal specification more accessible to system designers as well. Furthermore, even though our memory model specification itself is not executable (unlike [30]) we show how it can be exploited for exploring the state space of a program.

As far as program verification is concerned, typically most works on multi-threaded program verification are oblivious of the programming language memory model. For all such works, the execution model implicitly assumed is Sequential Consistency — operations in a thread proceed in program order and any interleaving among the threads is possible. Integrating programming language level memory models for reasoning about programs has hardly been studied. In particular, our previous work [28] integrated an operational specification of the Java Memory Model for software model checking. Also, the work of [31] integrates an executable memory model specification for detecting data races in multi-threaded Java programs. This work develops executable descriptions of memory models, unlike our work. The work explores the Sequential Consistency memory model, even though in principle this restriction can be removed.

Our checker verifies programs at the level of bytecodes; its state space representation has similarities with the Java Path Finder (JPF) model checker [15]. However, JPF is not sensitive to Java memory model, and it implicitly considers sequential consistency as the program execution model. In fact, works on bytecode level formal reasoning (*e.g.*, see [24] and the articles therein) typically have not considered the programming language level memory model.

The work of [18] develops a behavioral simulator to explore program behaviors allowed by the Java memory model. Apart from the differences in programming language (Java and C#) there are at least two conceptual differences between our work and [18]. First of all, their explorer works at the level of abstract operations such as read/write/lock/unlock whereas our checker operates at the bytecode level. Secondly, and more importantly, our tool does not only explore all program executions allowed by the .NET memory model. It can also suggest which barriers are to be inserted for disallowing program executions which are not sequentially consistent but are allowed by the (more relaxed) .NET memory model. This technique is *generic* and is not restricted to C#.

Finally, an alternative to our strategy of inserting memory barriers might be to mark all shared variables in the program as *volatile* [20]. We however

Thread 0	Thread 1
1. `lock0 = 1;`	A. `lock1 = 1;`
2. `turn = 1;`	B. `turn = 0;`
3. `while(1){`	C. `while(1) {`
4. `    if (lock1!=1)||(turn==0)`	D. `    if (lock0!=1)||(turn==1)`
5. `                    break; }`	E. `                    break; }`
6. `counter++;`	F. `counter++;`
7. `lock0 = 0;`	G. `lock1 = 0;`

Figure 1: Program encoding Peterson's mutual exclusion

note this does not work due to the weak definition and implementation of volatiles in C#. In particular, C# language documents [20] and C# implementations (*e.g.,* .NET 2.0) seem to allow reordering of volatile writes occurring before volatile reads in a program thread. On the other hand, memory barriers have a clear well-understood semantics but they incur performance overheads. For this reason, given an invariant property φ we insert *minimal* memory barriers in the program text which disallow all non-sequentially consistent execution traces violating invariant φ. Note that we are inserting memory barriers to disallow execution traces (in a state transition graph) which violate a *given* invariant property. Thus, we do not seek to avoid all data races, our aim is to avoid violations of a given program invariant.

3 A Working Example

We consider Peterson's mutual exclusion algorithm [27] to illustrate our approach. The algorithm (see Figure 1) uses two `lock` variables and one shared `turn` variable to ensure mutually exclusive access to a critical section; a shared variable `counter` is incremented within the critical section. Initially, we have `lock0 = lock1 = turn = counter = 0`.

In this program we are interested in the value of the variable `counter` when the program exits. Under sequential consistency, the algorithm is proven to allow only a single thread running in the critical section at the same time and thus when the program exits, we always have *counter == 2*. However when we run the program in a relaxed memory model (such as the .NET memory model) we can observe *counter == 1* at the end. One execution trace that can lead to such an observable value is as follows.

Thread 0	Thread 1
`write lock0 = 1` (line 1)	
`write turn = 1` (line 2)	
`read 0 from lock1, break` (line 4,5)	
`read 0 from counter` (line 6)	
	`write lock1 = 1` (line A)
	`write turn=0` (line B)

At this point, `Thread 0` can write 1 to `counter` (line 6), then write 0 to `lock0` (line 7). However if the writes to `counter` and `lock0` are reordered, `lock0 = 0` is written while `counter` still holds the old value 0. Thread 1 reads `lock0 = 0`, it will break out of its loop and load the value of `counter` which is now still 0. So both threads will write the value 1 to `counter`, leading to *counter == 1* at the end of the program.

Finding out such behaviors is a complex and error-prone task if it is done manually. Moreover even after we find them, how do we disable such behaviors? A quick way to fix the problem is to disable all reorderings within each thread; this clearly ensures Sequential Consistency. Recall that a memory barrier requires all instructions before the barrier to complete before the starting of all operations after the barrier. We can disable all reorderings allowed by a given relaxed memory model by inserting a memory barrier after each operation which can possibly be reordered. This will lead to very high performance overheads.

Note that running the above code with all shared variables being volatile also does not work. In Microsoft .NET Framework 2.0 on a Intel Pentium IV multi-processor platform, the variable `counter` is still not always observed to be 2 at the end of the program. This seems to be due to the possibility of (volatile-write $\rightarrow$ volatile-read) reorderings, an issue about which the CLI specification is also ambiguous. We discuss this matter in more details in the next section.

In this paper, we provide a solution to the problem of finding additional behaviors under a relaxed memory model and then disabling those behaviors without compromising program efficiency. Using our checker we can first explore all reachable states under Sequential Consistency and confirm that *counter == 2* is guaranteed at the end of the program. This amounts to verifying the invariant property $AG((pc == end) \Rightarrow (counter == 2))$ expressed in Computation Tree Logic (CTL). Here *pc* stands for the program counter (capturing the control locations of both the threads) and *end* stands for the last control location (where both threads have terminated). We then check the same invariant property under the .NET memory model; this check amounts to exploring more reachable states from the initial state (as compared to the set of reachable states computed under Sequential Consistency). We find that under the .NET memory model, our property can be violated since *counter == 1* is possible at the end of the program. The checker does a full reachable state space exploration and returns all the counter-example traces, that is, all possible ways of having $counter \neq 2$ at the end of the program.

However, more importantly, our checker does not stop at *detecting* possible additional (and undesirable) behaviors under the .NET memory model. After finding that the property $AG((pc == end) \Rightarrow (counter == 2))$ is violated under .NET memory model, our checker employs a memory barrier insertion heuristic to suggest an error *correction* strategy; it finds three places in each thread for inserting memory barriers. We only show the modified code for Thread1; Thread2's modification is similar.

```
lock0 = 1; MemoryBarrier; turn = 1;
while(1){ MemoryBarrier; if((lock1 != 1) || (turn == 0)) break; }
counter++; MemoryBarrier; lock0 = 0;
```

The inserted memory barriers are sufficient to ensure that the algorithm will work correctly under the relaxed memory model of C# (while still allowing the compiler/hardware to reorder other operations for maximum performance). This claim can again be verified using our checker — that is, by running the checker on the program with barriers under the relaxed .NET memory model we can verify that $AG((pc == end) \Rightarrow (counter == 2))$ holds. Moreover, the number of inserted barriers is also "optimal" — that is, at least so many barriers are needed to disallow all possible violations of $AG((pc == end) \Rightarrow (counter == 2))$ under the .NET memory model.

4 Memory Model Specification

In this section, we describe the programming language level memory model for C#, also called the .NET memory model, based on the information in two Microsoft's official ECMA standard document [21] and [20].

We present which reorderings are allowed by .NET memory model as a reordering table. We first describe the bytecode types it considers and then present allowed bytecode reorderings. The bytecode types are:

- Volatile reads/writes: Reads/writes to volatile variables (Variables in a C# program can be marked by the programmer by the keyword "volatile" indicating that any access to such a variable should access its master copy).

- Normal reads/writes: Reads/writes to variables which have not been marked as volatile in the program.

- Lock/unlock: The synchronization operations.

Among these operations, the model allows the reorderings summarized by Table 1. The model leaves a lot of possibility for optimization as long as

Reorder	2nd bytecode					
1st bytecode	Read	Write	Vol. Read	Vol. Write	Lock	Unlock
Read	Yes	Yes	Yes	No	Yes	No
Write	Yes	Yes	Yes	No	Yes	No
Vol. Read	No	No	No	No	No	No
Vol. Write	Yes	Yes	*Yes*	No	Yes	No
Lock	No	No	No	No	No	No
Unlock	Yes	Yes	Yes	No	No	No

Table 1: *Bytecode reordering allowed by the .NET memory model*

program dependencies within a thread are not violated (*e.g.*, `store x; load x` is never executed out-of-order due to data dependency on `x`). While data-dependency removal may allow more optimizations, the CLI documents explicitly *prohibit* doing so — see execution order rules in section 10.10 of [20]. Furthermore, here we are presenting the memory model in terms of *allowed bytecode reorderings* and not in terms of reorderings of abstract program actions. Optimizations which remove dependencies are usually performed by the compiler (the hardware platforms respect program dependencies) and hence would already be reflected in the bytecode.

Our reordering table is constructed based on the following considerations.

- Normal Reads and Writes are freely reordered.
- Locks and Unlocks are never reordered.
- Volatile Reads and writes have acquire-release semantics, that is, operations after (before) volatile-read (volatile-write) cannot be moved to before (after) the volatile-read (volatile-write).

More on (Volatile-write → Volatile-read) Re-orderings An interesting case is when a volatile write is followed by a volatile read (to a different variable). If we adhere to a strict ordering of all volatile operations, this reordering is disallowed; note that even if we allow (volatile-write → volatile-read) reorderings, we can still ensure that all writes to volatile variables are seen in the same order from all threads of execution. But it seems that Microsoft's .NET 2.0 allows this reordering on Peterson's mutual exclusion example shown in Section 3. Thus, in the program of Figure 1, the reads in Line 4 (or Line D) can get reordered w.r.t. writes in Lines 1,2 (Lines A, B) thereby leading to violation of mutual exclusion. The ECMA documents [21] and [20] are also silent on this issue; they only mention that operations cannot be moved before (after) a volatile read (volatile write), thus leaving out the case when a volatile write is followed by a volatile read.

In the following, we elaborate on this issue; we believe that this serves to clarify the common questions/doubts the readers may have on this matter.

We consider Peterson's mutual exclusion example shown in Figure 1. Let us suppose that all shared variables in this program — `lock0`, `lock1`, `turn`, `counter` — are marked as volatile. Note that, if (volatile-write → volatile-read) re-orderings are allowed, we still cannot guarantee mutual exclusion. Following is an execution trace which leads to `counter == 1` at the end of the program (which is only possible if mutual exclusion of critical section access is violated).

Thread 0	Thread 1
`(lock1 != 1)`	
`exit loop`	
`read counter` (reads 0)	
	`lock1 = 1`
	`(lock0 != 1)`
	`exit loop`
`lock0 = 1`	
`turn = 1`	
	`turn = 0`
	`counter++`
	`lock1 = 0`
`increment counter` (set 1)	
`lock0 = 0`	

In the preceding execution, note that the execution trace in each thread does not violate any data/control dependencies, as required by ECMA execution order rules 10.10 [20]. However, the write to volatile variables `lock0, turn` are moved after the read to volatile variable `lock1`. This seems to be allowed by the ECMA specification since:

- the execution order rules of C# (ECMA 334 10.10 [20]) point to ECMA 334 17.4.3 [20] for re-orderings w.r.t. volatile fields, and
- The rules in ECMA 334, 17.4.3 explicitly give an "acquire-release" semantics to volatile reads and writes.

In particular, ECMA 334, 17.4.3 says the following about re-orderings w.r.t. volatile reads and writes which we believe is relevant for deriving a formal specification of the C# memory model.

> *For non-volatile fields, optimization techniques that reorder instructions can lead to unexpected and unpredictable results in multi-threaded programs that access fields without synchronization such as that provided by the lock-statement (15.12). These optimizations can be performed by the compiler, by the runtime system, or by hardware. For volatile fields, such reordering optimizations are restricted:*
> *A read of a volatile field is called a volatile read. A volatile read has acquire semantics; that is, it is guaranteed to occur prior to any references to memory that occur after it in the instruction sequence.*
> *A write of a volatile field is called a volatile write. A volatile write has release semantics; that is, it is guaranteed to happen after any memory references prior to the write instruction in the instruction sequence.*

Based on the above informal specification, we can see that operations occurring before a volatile read can occur after the read. Similarly, operations occurring after a volatile write can occur before the write. Indeed, this is what is happening in our example execution trace for Peterson's program where we observe mutual exclusion violation *despite marking all shared variables as volatile.*

Finally, and most importantly, we also *observed* violation of mutual exclusion on .NET. We set up the experiment by letting each thread in Peterson' program (see Figure 1) access the critical section n times where n is a large number. If mutual exclusion is not violated, the value of `counter` at the end of the program should be $2 * n$. However, for large values of n (such as $n = 10,000,000$), we observed that the value of `counter` is less than $2 * n$ at the end of the program, showing mutual exclusion violation. The runs were taken on an IBM xSeries 255 multiprocessor machine with four processors.

Ongoing development of the memory model The current C# memory model presented in this paper was considered too weak, and there are currently proposals and discussions to strengthen it. But it is agreed that the new memory model will still be weaker than the x86 memory model for performance reasons. Considering that reasoning about correctness of programs under x86 memory model is already difficult, it will be even more difficult when the programmers consider the weaker C# model. A memory model aware checker is thus essential for verifying programs' correctness. Having such an checker also allows programmers to write very "clever", lock-free high performance code with confidence.

Towards a Checker Our checker implements the .NET Common Language Infrastructure (CLI) instruction set specified in [21]. We allow reordering of operations by (a) requiring all bytecodes to issue in program order and (b) allow certain bytecodes (whose reordering is allowed by the memory model) to complete out-of-order. Allowing reorderings according to the .NET memory model involves additional data structures in the state representation of our checker. In particular, for each thread we now need to maintain a list of “incomplete” bytecodes — bytecodes which have been issued but have not completed. The execution model allows a program thread to either execute its next bytecode or complete one of the incomplete bytecodes. We now proceed to elaborate on the state space representation and the reachability analysis.

5 Invariant Checker

The core of our checker is a virtual machine that executes .NET Common Language Infrastructure (CLI) bytecode using explicit state representation. It supports many threads of execution by interleaving issuing and completing of bytecodes from all threads. We implemented only a subset of the CLI. Features such as networking, I/O, class polymorphism and exception handling are not included in the implementation. *Bytecodes from .NET libraries are interpreted by our checker in the same manner as the bytecodes from the application program.* Our checker is geared towards verifying short sections of code (which have probably been optimized by the programmer for performance). Our checker cannot handle native code. Also, the state space construction in our explicit-state model checker may not terminate if the state space is unbounded (say due to unbounded recursion).

5.1 State Representation

We first consider the global state representation without considering the effects of the reorderings allowed by .NET memory model. To describe a global state we use the notion of *data units* of the CLI virtual machine. The virtual machine uses *data units* to hold the value of variables and stack items in the program. Each data unit has an identifier (for it to be referred to), and a modifiable value. The type of the modifiable value can be (a) one of the primitive data types, (b) reference types (pointers to objects), or (c) objects. New data units are created when a variable or a new object instance is allocated, or when a load instruction is executed. A global state of a program now consists of the following data units, corresponding to the different memory areas of the CLI virtual machine [21].

Program counter for each thread Each thread has a program counter to keep track of the next bytecode to be issued.

Stack for each thread Each thread has a stack which is used by most bytecodes to load/store data, pass parameters and return values from functions (or certain arithmetic / branch operations).

Heap The virtual machine has a single heap shared among all threads. Object instances and arrays are allocated from the heap. A data unit is created for each object as well each of its fields.

Static variables Static variables are shared among threads.

Frame Frames store local variables, arguments and return address of a method. Each time a method is called, a new frame is created and pushed into frame stack; this frame is popped when the method returns. Each local variable/argument is assigned one data unit.

All of the above data areas of the virtual machine are included in the global state space representation of a program. Now, in order to support the memory model, a new data structure is added to each thread: a list of incomplete bytecodes (given in program order). Each element of this list is one of the following type of operations — read, write, volatile read, volatile write, lock, unlock (the operation types mentioned in the .NET memory model, see Table 1). This completes the state space representation of our checker. We now describe the state space traversal.

5.2 Search Algorithm

Our checker performs reachability analysis by an explicit state depth-first search (starting from the initial state) over the state space representation discussed in the preceding. Given any state, how do we find the possible next states? This is done by picking any of the program threads, and letting it execute a single step. So, what counts as a single step for a program thread? In the usual model checkers (which implicitly assume Sequential Consistency), once a thread is chosen to take one step, the next operation from that thread forms the next step. In our checker the choices of next-step for a thread includes (a) issuing the next operation and (b) completing one of the pending operations (*i.e.*, operations which have started but not completed). The ability to complete pending operations out of order allows the checker to find all possible behaviors reachable under a given memory model (in this case the .NET memory model).

Thus, the search algorithm in our checker starts from the initial state, performs depth-first search and continues until there are no new states to be traversed. In order to ensure termination of this search, our checker of course needs to decide whether a given state has been already encountered. In existing explicit state software model checkers, this program state equivalence test is often done by comparing the so-called *memory image* in the two states, which includes the heap, stacks, frames and local variables. Our checker employs a similar test; however it also considers the list of incomplete operations in the two states. Two states are considered equivalent if we can establish a bijection between the two sets of data units such that the structural relation between two data units in one state is the same as the relation between their corresponding units in the other state.

Formally, two states `s` and `s'` with two sets of data units $D = \{d_1, d_2, ..., d_n\}$ and $D' = \{d'_1, d'_2, ..., d'_n\}$ are equivalent if and only if the program counters in all threads are equal and there exists a bijective function $f : D \rightarrow D'$ satisfying:

- For all $1 \leq i \leq n$, the value stored in d_i and $f(d_i)$ are equal.
- A static variable `x` in `s` is allocated data unit d_i if and only if it is allocated data unit $f(d_i)$ in `s'`.
- Data unit d_i is the k^{th} item on the stack (or frame, local variable, argument list, list of incomplete bytecodes) of the j^{th} thread in `s` iff $f(d_i)$ is the k^{th} item on the stack (or frame, local variable, argument list, list of incomplete bytecodes) of the j^{th} thread in `s'`.
- The reference type data unit d_i points to data unit d_j in `s` if and only if $f(d_i)$ points to $f(d_j)$ in `s'`.

In our implementation, *the global state representation is saved into a single sequence* so that if two state's sequences are identical, the two states are equivalent. Like the Java Path Finder model checker [15], we also use a hash function to make the state comparison efficient.

5.3 Experiments

In this section, we report the experiments used to evaluate our checker. Our checker for C# programs is itself implemented in C#. It takes the binaries of the benchmarks, disassembles them and checks the bytecode against a given invariant property via state space exploration.

The multi-threaded programs used in our experiments are listed in Table 2. Out of these, `peterson`, and `tbarrier` are standard algorithms that work correctly under Sequential Consistency, but require more synchronizations to do so in the C# memory model. The tournament barrier algorithm (taken from Java Grande benchmarks) provides an application program level implementation of the concurrency primitive "barrier" (*different from our memory barriers which prevent reordering of operations*) which allows two or more threads to handshake at a certain program point.

The programs `rw-vol` and `dc` have been discussed recently in the context of developing the new Java memory model [1]. In particular, `dc` has been used in recent literature as a test program to discuss the possible semantics of volatile variables in the new Java memory model [10]; this program involves the lazy initialization of a shared object by several threads.

The programs `rowo` and `po` are test programs taken from the ARCHTEST benchmark suite [7, 23]. ARCHTEST is a suite of test programs where

Program	Description	# of b.	# of p.b.	# of threads
`peterson`	Peterson's mutual exclusion [27]	120	37	2
`tbarrier`	Tournament barrier algorithm — *Barrier* benchmark from [14]	153	104	2
`dc`	Double-checked locking pattern [29]	77	59	2
`rw-vol`	Read-after-Write Java volatile semantic test [26]	92	20	2
`rowo`	Multiprocessor diagnostic tests ARCHTEST (ROWO)[7]	87	14	2
`po`	Multiprocessor diagnostic tests ARCHTEST (PO) [7]	132	20	2
`iw1`	Independent workers problem 1 [9]	102	64	2
`iw2`	Independent workers problem 2 [9]	105	67	2
`rw`	Two readers, single writer lock algorithm	161	62/51	3
`bb`	Bounded buffer Producer-consumer problem	252	87/90	3

Table 2: *Test programs used in our experiments. Column 3shows the total number of bytecodes. Column 4 shows the number of bytecodes in the parallel region of each thread, for problems with different code for each thread (i.e. reader/writer) two numbers are shown.*

the programs have been systematically constructed to check for violations of memory models (by generating violation of memory ordering rules imposed by the memory models). In particular, the program `rowo` checks for violation of ordering between multiple reads as well as multiple writes within a program thread; the program `po` checks for violation of program order among all operations in a program thread. These programs are effective for evaluating whether our checker can insert memory barriers to avoid behaviors not observable under Sequential Consistency.

Independent workers `iw1` and `iw2` are benchmarks to show the effectiveness of Partial Order Reduction 6 on programs with most of their data are local to the thread. The other programs `rw` and `bb` are two popular patterns for multithreaded programming programs: Reader-Writer and Producer-Consumer problem.

Program invariants For each of our benchmarks in Table 2 we also provide a program invariant for the reachability analysis to proceed and report violations. For the Peterson's algorithm (`peterson`) this invariant is the mutually exclusive access of shared resource. The invariant for `tbarrier` follows from the definition of the concurrency primitive "barrier". For the Double checked Locking pattern (`dc`) this invariant states that whenever the shared object's data is read, it has been initialized. The invariant for `rw-vol` benchmark is obtained from [26]. For the ARCHTEST programs `rowo` and `po`, this invariant is obtained from the rules of read/write order and program order respectively (see [7, 23]). The invariants for independent-workers, readers-writers, and producer-consumer — come from the problem definitions.

For all of the above benchmarks we employ our checker to find all reachable states under (a) Sequential Consistency and (b) .NET memory model. For the latter, recall that we allow each program thread to maintain a list of incomplete bytecodes so that bytecodes can be completed out of order. For our experiments we do not impose any a-priori bound on the size of this list of incomplete bytecodes. So in practice it is bounded only by the (finite) number of bytecodes in a program thread, and the program dependencies (if a bytecode is data/control dependent on another, they cannot be re-ordered). This exposes *all* possible behaviors of a given program under the .NET memory model.

Pgm.	# states	# transitions	# S.C. states	max queue	First C.E time(s)	Total time(s)
peterson	903	2794	101	5	0.16	0.91
tbarrier	1579	5812	154	8	0.7	1.51
dc	228	479	94	8	0.11	0.41
rw-vol	1646	5616	251	6	0.33	2.02
rowo	1831	4413	257	7	0.3	1.41
po	6143	22875	415	7	0.52	7.71
iw1	14886	67589	677	6	N/A	22.81
iw2	92613	285430	3477	5	N/A	86.53
rw	7762	28310	1832	8	N/A	8.82
bb	16072	51447	5707	10	N/A	25.11

Table 3: *Summary of invariant verification results. Column 2 and 4 shows the number of states under .NET memory model and Sequential Consistency (S.C.) Column 5 shows the maximum length of the incomplete bytecode queue in any thread during the execution with .NET memory model. Column 6 shows the time taken to find an invariant violating behavior (or counter-example abbreviated as C.E.) under .NET memory model that cannot be found in S.C. Column 7 shows the total time to search the entire state graph.*

Our checker performs reachability analysis to explore the reachable states under Sequential Consistency and the .NET memory model. Clearly, the reachability analysis under the .NET memory model takes more time since it involves exploring a superset of the states reachable under Sequential Consistency. In Table 3 we report the running time of our checker for obtaining the first counter-example (column *CE*) and for performing full reachability analysis (column *FR*). The time taken to find the first counter-example is not high; so if the user is only interested in detecting a violation our checker can produce one in a short time. The time to perform full reachability analysis (thereby finding all counter-example traces) is tolerable, but much larger than the time to find one counter-example. All experiments were conducted on a 2.2 Ghz machine with 2.5 GB of main memory. We do not report the time taken to find the first counter-example in certain benchmarks — these benchmarks do not have any counter-examples, that is, the invariant being verified is true.

6 Partial Order Reduction

By allowing program operations to complete out-of-order, our checker explores more behaviors than the normal model checkers based on Sequential Consistency. It is necessary to reduce the number of states visited by the search while preserving the correctness of the algorithms. Partial Order Reduction is a well-established technique (*e.g.* see the book [6], Chapter 10 to get a background on the topic and its relevant references) to reduce size of the state space using the commutativity of independent transitions. In [6], the independence relation between transitions is defined statically based on the effect of the transition, and regardless of the state they are in. Partial Order Reduction with dynamic escape analysis [9] expanded the concept of independence so that transitions can be independent in some states while dependent in other states.

6.1 Forever Thread-local Transitions and Thread-local Independence Condition

In this paper, we have modified the algorithm in [6] to work with instruction re-orderings, dynamic independence relation, and also improved the ample set choosing algorithm. We first give some basic definitions.

Definition 1 (Transition)**.** *A transition in the state transition graph of a program can be one of the following.*

- *Execute an action (one bytecode).*
    - *complete immediately (branching, arithmetic, comparison or synchronization bytecodes)*
    - *schedule an incomplete action (read, write, lock or unlock bytecodes)*

- *Complete an incomplete action.*

Definition 2 (Thread-local data)**.** *Data elements that can be accessed by only one thread are defined as thread-local data. Static variables are globally accessible and thus not thread-local. Stack items, local variables, formal parameters are thread-local. Data elements on the heap are thread-local or not depending on the memory structure at each state.*

Definition 3 (Thread-local transition)**.** *Using the definition of transitions and thread-local data, we can define a thread-local transition as follows: A transition t at state s is thread-local if t is one of the following:*

- *Execute an instruction that completes immediately and have effect on* thread-local *data only.*
- *Schedule an incomplete action.*
- *Complete an incomplete action having effect on* thread-local data *only.*

A data item can change its *thread-local* property in another state because of data passing and/or changing data on the heap. So a transition is *thread-local* or not depends on the state it is in. If a heap-allocated object is accessible from only one thread in a certain state s, we say that the object is thread local in s. However, there are data items that can only be accessed by a single thread: operations on them are local regardless of the states they are in. We call these *forever-local-transitions*, defined in the following.

Definition 4 (Forever-local-transition)**.** *Forever-local-transitions are transitions that only affects local thread state: scheduling an incomplete action or completing an incomplete action that only affects a thread's stack, local variables or formal parameters.*

We are now in a position to define the notion of *thread-local independence of transitions.*

Definition 5 (Thread-local Independence of transitions)**.** *For all state s, for all threads t_1 and t_2, for all transitions α of t_1 and β of t_2, α and β are independent iff*

- *α is forever thread-local*
- *$t_1 \neq t_2$ and α is thread-local at s.*

We need to prove that the above independence relation satisfies two conditions of *Enabledness* and *Commutativity* in [6]. The set of transitions enabled in a state s is denoted as *enabled*(s).

- Enabledness: If $\alpha, \beta \in \mathit{enabled}(s)$ then $\alpha \in \mathit{enabled}(\beta(s))$.

  The only way in which β disables α is through thread synchronization actions by locking the resource that α is going to acquire. But by definition, one of them must be a thread-local or forever-thread local transition, which cannot be a synchronization action. So *enabledness* is preserved.

- Commutativity: If $\alpha, \beta \in \mathit{enabled}(s)$ then $\alpha(\beta(s)) = \beta(\alpha(s))$.

  When α and β are on different threads, one of them must be local so the effect of each transition is not visible to the other, thus $\alpha(\beta(s)) = \beta(\alpha(s))$. When α and β are on the same thread, by definition one of them must be forever-local. There are two types of forever-local transition: scheduling an incomplete action and complete an incomplete action that only affects data structure local to the threads (stack, local variable and formal parameters). There are three cases:

  – If α is scheduling an incomplete action and β is an access to thread-local data structures, then it is clear that their effects are not overlapping, and $\alpha(\beta(s)) = \beta(\alpha(s))$.

  – α and β are both scheduling an incomplete action. Because in the same thread scheduling incomplete actions must respect program order, there can never be two such transitions enabled in a state. So this case never happens.

  – α and β are both accesses to thread local data structures. If the values they are accessing are overlapping, data dependency restricts one to complete after the other, thus they can not be in $\mathit{enabled}(s)$ at the same time. If the values they are accessing are not overlapping, it is clear that $\alpha(\beta(s)) = \beta(\alpha(s))$.

6.2 Choosing the ample set

The ample set construction proceeds as follows. When there is a *forever-thread-local* transition α in a state s, we choose $\mathit{ample}(s) = \{\alpha\}$. If such a transition does not exist, the ample set is chosen as the whole set of transitions of a thread t if it satisfies three conditions **C1**, **C2**, **C3** for ample sets. If no such thread t can be found we simply set $\mathit{ample}(s) = \mathit{enabled}(s)$, the set of enabled transitions in state s (*i.e.*, there is no reduction in the set of transitions to be explored from s).

The conditions **C1**, **C2**, **C3** are given in [6], but in our setting they need to be modified as follows. The modifications are similar to [9]. However [9] does not consider operation re-orderings which is crucial to any memory model sensitive checker. In the following, we use $\mathit{enabled\_thd\_trans}(s, t) \subseteq$

enabled(s) to denote the set of transitions of thread t which are enabled in state s.

We also use the notion of ***invisible*** as in [6] — a transition is invisible when its execution from any state does not affect the invariant we want to check.

Algorithm 1 *Ample set construction*

```
function AMPLE($s$)
    for all transition $a \in$ state $s$ do
        if $a$ is forever-local then
            return $a$
        end if
    end for
    for all thread $t$ do
        if $enabled\_thd\_trans(s,t) \neq \varnothing$ and $check\_c1(s,t)$ and $check\_c2(s,t)$
and $check\_c3(s,t)$ then
            return $enabled\_thd\_trans(s,t)$
        end if
    end for
    return $enabled(s)$
end function
```

We modify condition **C1** in [6] for state-wise independence relation as follows. Along every path in the full state graph that starts at s, the following condition holds: a transition that is dependent *at s* on a transition in *ample*(s) cannot be executed without a transition in *ample*(s) occurring first. In order to check **C1**, we use a conservative approach to make sure that the ample set we are choosing does not violate **C1**, but may rule out some feasible ones. In particular, let E_i be the set of data elements that thread t_i can reach. E_i is computed by including t_i's local data structures (stack, local variables, incomplete action queue, ...) and following the references/pointers from these data. Let $E = E_1 \cup \ldots \cup E_{k-1} \cup E_{k+1} \cup \ldots \cup E_n$. If a transition α in t_k can access E then the set of transitions of thread t_k is not a candidate for *ample*(s). In the absence of such transitions, the transitions in t_k are independent of the transitions in *enabled*$(s) \setminus$ *ample*(s), and *ample*(s) is chosen as all transitions from thread t.

Algorithm 2 *Check_C1 algorithm*

```
function CHECK_C1(state $s$, thread $t$)
    $G$ = static variables and data accessed from static variables
    for all thread $t' \neq t$ do
        $G = G \cup \{$data accessible by $t'\}$
    end for
    for all transition $a \in t$ do
        if $a$ accesses data in $G$ then
            return false
        end if
    end for
    return true
end function
```

Condition **C2** in [6] is *If s is not fully expanded, then every $\alpha \in ample(s)$ is invisible.* (C2) is satisfied if all transitions in t is invisible.

Algorithm 3 *Check_C2 algorithm*

```
function CHECK_C2(state $s$, thread $t$)
    for all transition $a \in t$ do
        if $visible(a)$ then
            return false
        end if
    end for
    return true
end function
```

Condition **C3** in [6] is: *A cycle is not allowed if it contains a state in which some transitions α is enabled, but is never included in $ample(s)$ for any state s on the cycle.* **C3** is check using the algorithm in [6]: if the resulting state of a transition $\alpha \in t$ is on the search stack, we do not choose this set of t's transitions as $ample(s)$ because doing so will make a cycle violating the condition.

Algorithm 4 *Check_C3 algorithm*

```
function CHECK_C3(state $s$, thread $t$)
    for all transition $a \in t$ do
        if $on\_stack(a(s))$ then
            return false
        end if
    end for
    return true
end function
```

The reduced state graph is built using depth first search, at each state s the next transitions are retrieved from $ample(s)$ instead of $enabled(s)$. Because the size of $ample(s)$ is usually much smaller than $enabled(s)$, the size of the state space is significantly reduced.

6.3 Correctness Proof

We now modify the correctness proof of partial order reduction given in Chapter 10 of [6] to obtain the correctness proof of our ample set choosing method.

The correctness proof in Section 10.6 of [6] shows that for every path in the full state graph, we can construct a stuttering equivalent path belonging to the reduced state graph by reordering the order of independent transitions. The properties of the path allows us to prove that the reduced state graph and the full state graph are stuttering equivalent, so searching in either of them gives us the same reachability or LTL$_{-X}$ result. We show that the proof is still applicable with our method for choosing ample sets.

Let $\eta = s_0 \overset{\alpha_1}{\rightarrow} s_1 \overset{\alpha_2}{\rightarrow} ... \overset{\alpha_k}{\rightarrow} s_k$ and $\theta = s'_0 \overset{\beta_1}{\rightarrow} s'_1 \overset{\beta_2}{\rightarrow} ... \overset{\beta_k}{\rightarrow} s'_{k'}$. If $s_k = s'_0$ then $\eta \circ \theta$ denotes the path $s_0 \overset{\alpha_1}{\rightarrow} s_1 \overset{\alpha_2}{\rightarrow} ... \overset{\alpha_k}{\rightarrow} s_k \overset{\beta_1}{\rightarrow} s'_1 \overset{\beta_2}{\rightarrow} ...s'_{k'}$

We will construct recursively π_{i+1} from $\pi_i = \eta_i \circ \theta_i$ to produce an infinite sequence of paths $\pi_0, \pi_1, ...$ from some infinite path σ in the full state graph such that $\pi_0 = \sigma$ and $\pi_i = \eta_i \circ \theta_i$, $|\eta_i| = i$. Let $\theta_i = s_0 \overset{\alpha_0 = \alpha}{\longrightarrow} s_1 \overset{\alpha_1}{\rightarrow} s_2 \overset{\alpha_2}{\rightarrow} ...$

Consider the following two cases.

- $\alpha \in ample(s_0)$. Let $\eta_{i+1} = \eta_i \circ (s_0 \overset{\alpha}{\rightarrow} \alpha(s_0))$ and $\theta_{i+1} = s_1 \overset{\alpha_1}{\rightarrow} s_2 \overset{\alpha_2}{\rightarrow}...$

- $\alpha \notin ample(s_0)$, so $\alpha = \alpha_0$ is independent with the transitions in $ample(s_0)$. The transitions in $ample(s_0)$ continue to be independent with other transitions at subsequent states s_k from s. We prove this by induction on k with the help of the following Lemma.

  Lemma 1. If α is independent with β_0, β_1 in state s, and $s' = \beta_0(s)$ then α is independent with β_1 in s'.

  Proof of Lemma 1: There are three cases.

  - α is *forever-thread-local*, it is obvious that α is still independent with β_1 because it is not possible to change thread-locality of α
  - α is *thread-local*, the only way to make β_1 dependent on α is making α non thread-local. To do so, β_0 must be in the same thread with α in order to make the data accessed by α to be globally accessible. But β_0 is independent with α, so it must be either on another thread or *forever-thread-local.* In both cases β_0 cannot affect thread-locality of α.

- α is not *thread-local*, so β_0 and β_1 must be *thread-local* or *forever-thread-local*. Because they are both accessing local data, they cannot make the other non-thread-local. So α is still dependent in $\beta_0(s)$.

This completes the proof of Lemma 1. □

Continuing with our induction argument for the case $\alpha \notin \mathit{ample}(s_0)$, let us consider a transition β in $\mathit{ample}(s_0)$.

- $k = 1$. By Lemma 1, $\beta \in \mathit{enabled}(s_1)$ and by applying Lemma 1 with $\beta_1 = \alpha_i$ we have β still independence with each α_i for $i = 1, ...n$
- Induction hypothesis: $k = m$: If $\beta \in \mathit{enable}(s_0)$ and β is independent with $\alpha_0, \alpha_1, ...\alpha_k$, then $\beta \in \mathit{enabled}(s_m)$ and β is independent with $\alpha_m, \alpha_{m+1}, ...\alpha_n$
- $k = m + 1$. We have $\beta \in \mathit{enabled}(s_m)$ and β is independent with $\alpha_m, \alpha_{m+1}, ...\alpha_n$. Applying Lemma 1 to s_m, α_m and β we have $\beta \in \mathit{enabled}(s_{m+1})$ and β is independent with $\alpha_{m+1}, \alpha_{m+2}...\alpha_n$.

This completes the induction argument for the case $\alpha \notin \mathit{ample}(s_0)$. We have proved that, in this case, the transitions in $\mathit{ample}(s_0)$ continue to be independent with other transitions at subsequent states.

Because every transition $\beta \in \mathit{enabled}(s_0)$ stays independent with the other transitions, we have two cases to consider.

- $\beta \in \mathit{ample}(s_0)$ appears on η_i after k independent transitions:
  $s_0 \overset{\alpha_0}{\rightarrow} s_1 \overset{\alpha_1}{\rightarrow} ...s_{k-1} \overset{\alpha_{k-1}}{\rightarrow} s_k \overset{\beta=\alpha_k}{\rightarrow} s_{k+1} \overset{\alpha_{k+1}}{\rightarrow} ...$
  Because β and α_{k-1} are independent, by the commutativity property, we can change the order of applying transition β and α_{k-1}:
  $s_{k-1} \overset{\beta=\alpha_k}{\rightarrow} \beta(s_{k-1}) \overset{\alpha_{k-1}}{\rightarrow} \alpha_{k-1}(\alpha_k(s_{k-1})) = \alpha_k(\alpha_{k-1}(s_{k-1})) = s_{k+1}$
  It has the effect of moving up β before α_{k-1}. By repeatedly moving up β one transition at a time with $\alpha_{k-2}, \alpha_{k-3}, ...$, we have the path
  $\xi = s_0 \overset{\beta}{\rightarrow} \beta(s_0) \overset{\alpha_0}{\rightarrow} \beta(s_1) \overset{\alpha_1}{\rightarrow} ... \overset{\alpha_{k-1}}{\rightarrow} \beta(s_k) \overset{\alpha_{k+1}}{\rightarrow} ...$
  Let $\eta_{i+1} = \eta_i \circ (s_0 \overset{\beta}{\rightarrow} \beta(s_0))$ and θ_{i+1} be the path obtained from ξ by removing the first transition.
- β is independent of all $\alpha_i \in \theta$. Similarly, there is also a path $\xi = s_0 \overset{\beta}{\rightarrow} \beta(s_0) \overset{\alpha_0=\alpha}{\rightarrow} \beta(s_1) \overset{\alpha_1}{\rightarrow} ...$ and we can define η_{i+1}, θ_{i+1} as above.

The construction of ξ is crucial in providing a stuttering equivalent path belonging to the reduced state graph for every path in the full state graph. The rest of the proof follows exactly as in [6].

Work of [9] We note that exploiting independence by locality of data has been studied in [9] but without the use of *forever-thread-local* transitions (Def. 4). The execution model assumed in [9] is the traditional sequentially consistent interleaving model. The notion of forever thread-local transitions (introduced by us), turns out to be very useful in reducing behaviors when operation re-orderings are allowed (by relaxed memory models). This was confirmed by our experiments. We now present our experiments to show the state space reduction achieved.

6.4 Experimental results

We have measured the efficacy of partial reduction by evaluating the time to perform full reachability analysis with and without partial reduction. The results for these experiments appear in Table 4. The multi-threaded programs used in our experiments have been described in the last section (see table 2).

We can observe that the benefits from partial order reduction vary widely across programs. If a program uses lot of local data (especially like `iw1` and `iw2`), many of the transitions are discarded by our partial-order reduction and hence the time for reachability analysis reduces dramatically. However, for multi-threaded programs which have lot of data shared across threads, the benefits from partial-order reduction are not substantial.

7 Disabling Undesirable Program Behaviors

Given a multi-threaded C# program, we are interested in computing the set of reachable states from the initial state. The set of reachable states under the .NET memory model is guaranteed to be a superset of the reachable state set under Sequential Consistency. In this section, we discuss tactics for disallowing the additional states reached under the .NET memory model. Since these additional states are reached due to certain reordering of operations within program threads, we can avoid those states if such reorderings are disabled by inserting barriers/fences in the program text.

While doing reachability analysis we build (on-the-fly) the state transition graph. Each vertex represents one state, each directed edge represents a transition from one state to another. Consider the state transition system constructed for the .NET memory model. Because this memory model is more relaxed than Sequential Consistency, we can divide the graph edges into two types: solid edges correspond to transitions which can be

Pgm.	Normal			With POR		
	# of states	# of transitions	time (sec)	# of states	# of transitions	time (sec)
peterson	903	2794	0.91	881	2697	0.90
tbarrier	1579	5812	1.51	801	2641	1.5
dc	228	479	0.41	150	272	0.36
rw-vol	1646	5616	2.02	1613	5458	2.06
rowo	1831	4413	1.41	1776	4231	1.36
po	6143	22875	7.71	6091	22595	7.64
iw1	14886	67589	22.81	246	557	0.59
iw2	92613	285430	86.53	353	560	0.63
rw	7762	28310	8.82	6851	22119	7.48
bb	16072	51447	25.11	1704	4048	2.08

Table 4: *Comparison of state transition graph size and time taken to build the graph*

performed under Sequential Consistency (complete the bytecodes in order within a thread) and dashed edges correspond to transitions which can *only* be performed under .NET memory model (requires completing bytecodes out-of-order). From the initial state, if we traverse only solid edges we can visit all states reachable under Sequential Consistency. We color the corresponding vertices as white and the remaining vertices as black. The black vertices denotes the additional states which are reached due to the reorderings allowed by the relaxed memory model (see Figure 2 for illustration). Note that if (a) we are seeking to verify an invariant property φ under Sequential Consistency as well as the .NET model, (b) φ is true under Sequential Consistency and (c) φ is false under the .NET memory model — the states violating φ must be black states. However, not all the black states may denote violation of φ as shown in the schematic state transition graph of Figure 2.

Basic Mincut Formulation To prevent the execution from reaching the violating black states, we need to remove some of the edges from the graph. The solid edges cannot be removed because their corresponding transitions are allowed under Sequential Consistency. The dashed edges can be removed selectively by putting barriers. However note that the barriers will appear in the program text, so inserting *one barrier* in the program can disable *many dashed edges* in the state transition graph. We find out the minimal number of dashed edges to be removed so that the violating black states become unreachable; we then find out the memory barriers to be inserted in the program text for removing these dashed edges. Now we describe our strat-

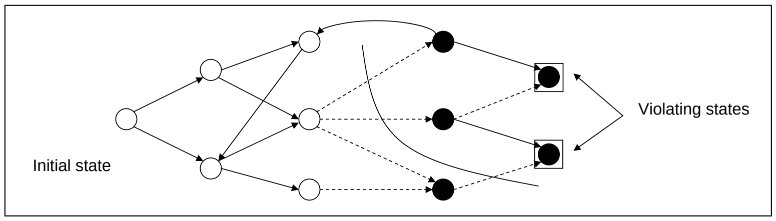

Figure 2: *State transition graph under a relaxed memory model; only white states can be reached under Sequential Consistency. A cut is shown separating the initial state from "violating" states.*

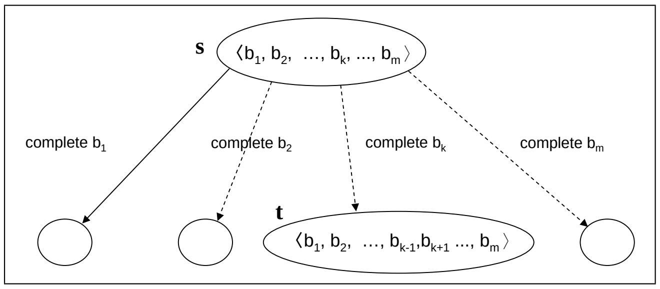

Figure 3: *Transitions from a state, a dashed edge indicates the transition requires an out-of-order completion of bytecodes*

egy for computing the minimal number of dashed edges to be removed. We compute the minimum cut $C = \{e_1, e_2, ..., e_n\}$ where $e_1, \ldots, e_n$ are dashed edges in the state transition graph such that there is no directed path from the initial state to any violating black state (denoting violation of the invariant φ being verified) without passing through an edge in C. We find the minimal set of dashed edges by employing the well-known Ford-Fulkerson maxflow-mincut algorithm [11]. To find the minimal number of dashed edges in the state transition graph as the mincut, we can set the capacity of each dashed edge to 1 and each solid edge to infinity.

How can we locate the barrier insertion point in the program such that a given dashed edge in the state transition graph is removed? Recall that a dashed edge in the state transition graph denotes a state transition which is caused by out-of-order completion of a bytecode. In Figure 3 state s has m incomplete bytecodes $\langle b_1, b_2, \ldots, b_k, \ldots, b_m\rangle$ (given in program order). The transition that completes bytecode b_1 does not require an out-of-order completion of bytecodes while the transitions that complete b_k with $2 \leq k \leq m$ do. The removal of edge from state s to state t (corresponding to the completion of bytecode b_k, see Figure 3) is identified with inserting a barrier before bytecode b_k.

Modified Mincut Formulation Note that the minimal set of dashed edges in the state transition graph may not always produce the minimal number of barriers in the program text. At the same time, inserting minimal number of barriers in the program text may not be desirable in the first place since they do not indicate the actual number of barriers encountered during program execution.[1] However if we want to minimize the number of barriers inserted into the program, we can do so by simply modifying the capacities of the dashed edges in the state transition graph. We partition the dashed edges in the state transition graph into disjoint partitions s.t. edges e, e' belong to the same partition iff disabling of both e and e' can be achieved by inserting a single barrier in the program. We can then assign capacities to the edges in such a way that the sum of capacities of the edges in each partition is equal — thereby giving equal importance to each possible program point where a barrier could be inserted. The maxflow-mincut algorithm is now run with these modified capacities (of the dashed edges); the solid edges still carry a weight of infinity to prevent them from appearing in the min cut.

Complexity The Maxflow-mincut algorithm has time complexity of $O(m * f)$ where m is the number of edges in the state transition graph and f is the value of the maximum flow. The quantity f depends on how the capacities of the state transition graph edges are assigned. In all our experiments, f was less than 150 for all our test programs (using basic or modified mincut formulation).

Experiments After exploring all reachable states under the .NET memory model, our checker can insert barriers via a maxflow-mincut algorithm (we used the "Modified Mincut Formulation" presented in Section 7). The time to run the maxflow algorithm is small as shown in column *Mflow* of Table 3. The results of the barrier insertion step are shown in the *# barriers* column of Table 3. This column shows the total number of barriers inserted by our tool into the program so that any program execution trace which (a) violates the invariant being verified and (b) is disallowed by Sequential Consistency, — is not observed even when the program is run under the relaxed .NET memory model. Among the 10 benchmarks, only 6 of them produce behaviors not found in Sequential Consistency. Our checker has successfully put a small number of barriers each to disallow the execution to visit such states.

A note about Doubled Checked Locking Interestingly, the reader may notice that our checker inserts only one barrier for the Double Checked

[1] A single barrier inside a loop which is iterated many times can introduce higher performance overheads than several barriers outside the loop.

Pgm.	Time to run Maxflow (secs)	# barriers inserted
`peterson`	0.04	3
`tbarrier`	0.05	3
`dc`	0.03	1
`rw-vol`	0.23	4
`rowo`	0.05	2
`po`	1.48	6

Table 5: *Experiments on disabling observable behaviors not reachable under sequential consistency*

```
class Helper{
private Helper helper = null;
public Helper GetHelper() {
MemoryBarrier();               // Barrier (1)
if (helper == null){
    synchronized(this){
        if (helper == null){
            tmp = new Helper();
            MemoryBarrier();     // Barrier(2)
            helper = tmp;
        }
    }
}
return helper;
}
// Other methods of Helper class
}
```

Figure 4: *Schematic Code for Double Checked Locking*

Locking pattern (same as the solution in [22], [3] and [2]) while the solution using "explicit memory barriers" given in [10] suggests putting two barriers. Both solutions are correct, because they work for different memory models. We show the schematic code for Double Checked Locking pattern in Figure 4. Execution of this code without the memory barriers under the C# memory model, allows a reader to read a helper object whose fields have not yet been fully initialized; this behavior is prevented by the two memory barriers.

We note that all solutions put barrier (2) in Figure 4 to disallow the reordering of the constructor's writes to `helper`'s data and the writes to `helper` pointer. Such a reordering will result in main memory having helper pointing to uninitialized data. This reordering of writes are common and allowed in most memory model, so barrier (2) are required.

However, only [10] puts barrier(1) in Figure 4. The purpose of this barrier is to ensure that the read of helper's data structure (in the function that calls `GetHelper()` is not reordered with the read of the pointer `helper`). How can we read the data before we read its address ? It seems unreasonable, but in some systems with local cache memory for each processors, the cache controller may decide to update its local copy of the data at the address `helper` is *going to* point to from main memory, while the data is uninitialized. After that it updates its local copy of `helper`, now is not null. So the thread will see uninitialized data although the main memory is written in order. The effect of this can be seen from the abstract view point as reordering two data-dependent reads.

For systems that respect data dependency while reordering such as *.NET memory model compliant implementations*, the compiler needs to insert barriers to disable such behaviors, which are not even allowed by the .NET memory model. The programmer's responsibility should be to indicate that the compiler needs to put a barrier (2) to disable the reorderings allowed by .NET memory model which lead to undesirable program behaviors (in this case reading an uninitialized helper object).

8 Discussion

In this paper, we have presented an invariant checker which works on the bytecode representation of multi-threaded C# programs. The main novelties of our work are (a) we can expose non sequentially consistent execution traces of a program which are allowed by the .NET memory model, and (b) after inspecting the counter-example traces violating a given invariant, we can automatically insert barriers to disallow such executions. Furthermore, to ensure that our checker can scale up to realistic programs, we also extended partial order reduction strategies for model checking multi-threaded software to the situation where re-ordering of operations in a thread is allowed. Our memory model sensitive checker (including its source code) and all the test programs can be freely downloaded from `http://www.comp.nus.edu.sg/~release/mmchecker`

Acknowledgments This work was supported partly by a grant from Microsoft under the Shared Source CLI (SSCLI aka Rotor) programme, partly by a Public Sector grant from A*STAR (Singapore) and partly by an internal grant from NUS (Singapore). Tulika Mitra and Weng-Fai Wong gave us useful feedback during the project.

References

[1] Java Specification Request (JSR) 133. Java Memory Model and Thread Specification revision, 2004.

[2] B. Abrams. `http://blogs.msdn.com/brada/archive/2004/05/12/130935.aspx`.

[3] C. Brumme. Weblog: Memory model. `http://blogs.msdn.com/cbrumme/archive/2003/05/17/51445.aspx`.

[4] S. Burckhardt, R. Alur, and M. Martin. Bounded model checking of concurrent data types on relaxed memory models: A case study. In *International Conference on Computer Aided Verification (CAV)*, 2006.

[5] S. Burckhardt, R. Alur, and M. Martin. Checkfence: Checking consistency of concurrent data types on relaxed memory models. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, 2007.

[6] E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[7] W. W. Collier. *Reasoning about Parallel Architectures*. Prentice Hall, 1992. Details available from `http://www.mpdiag.com/archtest.html`.

[8] D.L. Dill, S. Park, and A. Nowatzyk. Formal specification of abstract memory models. In *Symposium on Research on Integrated Systems*. MIT Press, 1993.

[9] M.B. Dwyer, J. Hatcliff, Robby, and V. R. Prasad. Exploiting object escape and locking information in partial order reduction for concurrent object-oriented programs. *Formal Methods in System Design*, 25(2/3):199–240, 2004.

[10] D. Bacon et al. The "Double-checked Locking is Broken" declaration. `http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html`.

[11] L.R. Ford and D.R. Fulkerson. Maximum flow through a network. In *Canadian Journal on Mathematics*, volume 8, pages 399–404, 1956.

[12] G. Gopalakrishnan, Y. Yang, and G. Lindstrom. QB or not QB: An efficient execution verification tool for memory orderings. In *International Conference on Computer Aided Verification CAV*, 2004.

[13] T.Q. Huynh and A. Roychoudhury. A memory model sensitive checker for C#. In *Formal Methods Symposium (FM)*, 2006. `http://www.comp.nus.edu.sg/~abhik/pdf/fm06.pdf`.

[14] JGF. The Java Grande Forum Multi-threaded Benchmarks, 2001. `http://www.epcc.ed.ac.uk/computing/research_activities/java_grande/threads.html`.

[15] JPF. The Java Path Finder model checking tool, 2005. `http://javapathfinder.sourceforge.net/`.

[16] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9), 1979.

[17] D. Lea. The JSR-133 cookbook for compiler writers. `http://gee.cs.oswego.edu/dl/jmm/cookbook.html`.

[18] J. Manson and W. Pugh. The java memory model simulator. *Workshop on Formal Techniques for Java-like Programs, in association with ECOOP*, 2002.

[19] J. Manson, W. Pugh, and S. Adve. The Java Memory Model. In *ACM Symposium on Principles of Programming Languages (POPL)*, 2005.

[20] Microsoft. Standard ECMA-334 C# Specification, 2005. `http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-334.pdf`.

[21] Microsoft. Standard ECMA-335 Common Language Infrastructure (CLI), 2005. `http://www.ecma-international.org/publications/standards/Ecma-335.htm`.

[22] V. Morrison. Dotnet discussion: The DOTNET Memory Model. `http://discuss.develop.com/archives/wa.exe?A2=ind0203B&L=DOTNET&P=R375`.

[23] R. Nalumusu et al. The 'test model checking' approach to the verification of memory models of multiprocessors. In *International Conference on Computer Aided Verification (CAV)*, 1998.

[24] T. Nipkow et al. Special issue on Java bytecode verification. *Journal of Automated Reasoning (JAR)*, 30(3–4), 2003.

[25] S. Park and D.L. Dill. An executable specification and verifier for relaxed memory order. *IEEE Transactions on Computers*, 48(2), 1999.

[26] W. Pugh. Test for sequential consistency of volatiles. `http://www.cs.umd.edu/~pugh/java/memoryModel/ReadAfterWrite.java`.

[27] M. Raynal. *Algorithms for mutual exclusion*. MIT Press, 1986.

[28] A. Roychoudhury and T. Mitra. Specifying multithreaded Java semantics for program verification. In *ACM International Conference on Software Engineering (ICSE)*, 2002.

[29] D. Schmidt and T. Harrison. Double-checked locking: An optimization pattern for efficiently initializing and accessing thread-safe objects. In *3rd annual Pattern Languages of Program Design conference*, 1996.

[30] R.F. Stark and E. Borger. An ASM specification of C# threads and the .NET memory model. In *Abstract State Machines Workshop, LNCS 3065*, 2004.

[31] Y. Yang, G. Gopalakrishnan, and G. Lindstrom. Memory model sensitive data race analysis. In *International Conference on Formal Engineering Methods (ICFEM)*, 2004.

[32] Y. Yang, G. Gopalakrishnan, G. Lindstrom, and K. Slind. Analyzing the intel itanium memory ordering rules using logic programming and SAT. In *Correct Hardware Design and Verification Methods CHARME*, 2003.