Building a Better Concurrency: Cooperative Reasoning for Preemptive Execution

Jaeheon Yi

Computer Science Department
University of California at Santa Cruz
Santa Cruz, CA 95064
jaeheon@cs.ucsc.edu

1. Problem and Motivation

How can we make multithreading more reliable for the majority of programmers? Decades of experience have proven that multithreaded software is notoriously difficult to create and maintain [14]. Despite this difficulty, the widespread usage of multicore processors means that multithreading, as a technique, has now entered into the toolbox of ordinary programmers [23]. For example, bugs related to multithreading and concurrency have been implicated in regional electricity blackouts, malfunctioning space rovers, and multiple deaths due to radiation poisoning.

Simply put, the threads-and-shared-memory paradigm provides an exceptionally brittle abstraction for programmers, essentially introducing a source of non-determinism typically not present in ordinary, sequential programming: unexpected thread interference. For example, even a familiar construct such as `x++` has unfamiliar results in a multithreaded setting, where it must be considered a non-atomic read-modify-write sequence, instead of as a simple atomic increment. This odd behavior may be overlooked when writing or inspecting the code, since thread interference is invisible inside the code `x++`. Our research question is how to make this thread interference visible to programmers. We propose syntactically specifying all such thread interference with yield annotations.

2. Background and Related Work

To place our work in context, we first examine the problems with preemptive semantics. *Preemptive semantics* is the default execution behavior of threads on modern multicore hardware, in which a context switch may occur after any operation by any thread. The challenge of preemptive semantics is the exponential blowup of possible thread interleavings, and is exacerbated by the lack of syntactic clues in the source code about these thread interleavings. Much research has been devoted to constraining this blowup of preemptive semantics, either by pruning out irrelevant interleavings or by preventing undesired interleavings. We outline some well-known concurrency properties and related major research proposals below.

Data race freedom. A *data race* occurs when two threads access the same shared variable, at least one is a write, and there is no intervening synchronization between these two accesses. The presence of data races is often a good indication of multithreading errors: the lack of synchronization is symptomatic of programmer oversight, and in many languages, like C++, the behavior of a data race is machine dependent or even undefined, due to the relaxed memory consistency induced on modern multicore hardware. For example, the following code snippet for a bank account program has a race on the account's balance `bal`, where another thread may update the value of `bal` between the read and write. In such a case, the intermediate written value would be lost.

```
int bal;
void accountUpdate(int amt) {
  bal = bal + amt;
}
```

The absence of data races, or *data race freedom*, constrains observable thread interference to occur at synchronization points rather than at variable accesses. This has two benefits: 1) the number of synchronization points is often far less than the number of variable accesses, and so constrains the blowup of preemptive semantics [25], and 2) data race freedom on modern hardware guarantees sequential consistency, which is the intuitive programming model that programmers naturally expect [2].

There is extensive literature on finding and fixing data races efficiently [9, 22, 18, 4, 7, 1, 5, 17, 19, 20, 16]. However, data race freedom is a somewhat low-level property, dealing with programs at the level of memory accesses. Hence a program that is race-free can still exhibit surprising behavior due to unexpected thread interleavings – but this time at the synchronization level. For example, the following code snippet has all memory accesses protected by a lock `m`, yet exhibits the same error as before:

```
int bal; lock m;
void accountUpdate(int amt) {
  int tmp;
  sync(m) { tmp = bal + amt; }
  sync(m) { bal = tmp; }
}
```

Atomicity. *Atomicity* is a higher-level concurrency property that checks the serializability of atomic block annotations in a program [10]. These atomic block annotations more directly capture the intent of the programmer by declaring code that is to be free from thread interference. The correctness of these atomic annotations can be *checked* via static or dynamic program analysis techniques [10, 11]. If these annotations are correct, then one may use sequential semantics inside an atomic block. By enabling the limited use of sequential semantics, atomicity reduces the number of thread interleavings to consider, thereby improving over preemptive semantics. For example, if we had specified the account update function to be atomic, an atomicity checker would have caught the atomicity violation:

```
int bal;
void accountUpdate(int amt) {
  atomic {
    bal = bal + amt; //error: atomicity violation
  }
}
```

Unfortunately, atomicity introduces a form of context-dependent, bi-modal interpretation of code: to use sequential semantics, one must first establish containment in some atomic block, which may not be easy. Outside an atomic block, we fall back to the difficult preemptive semantics – the type of reasoning we wished to avoid in the first place [24].

Transactional memory. *Transactional memory*, like atomicity, starts with atomic annotations to provide limited regions of sequential semantics for atomic blocks, or transactions [13]. Unlike atomicity, the transactional memory approach uses a runtime system to *enforce* the serializable execution of transactions. For example, the same code with an atomic block runs correctly under transactional memory:

```
int bal;
void accountUpdate(int amt) {
  atomic {
    bal = bal + amt; //serializable by runtime
  }
}
```

Although conceptually appealing, it has proven challenging to find a robust semantics for transactional memory [3], while performance issues hold back widespread adoption [6]. And since transactional memory is based on atomic annotations, transactional memory is also susceptible to many of the same pitfalls as atomicity.

	Transaction	Yield
Compile-time analysis	Atomicity	**Cooperability**
Run-time enforcement	Transactional Memory	Automatic Mutual Exclusion

Figure 1. Some alternatives to preemptive semantics: one may analyze or enforce either transactions or yields (transactional boundaries).

Automatic mutual exlusion. *Automatic mutual exclusion* (AME) inverts the transactional memory paradigm by placing *all* code in *some* transaction, and introduces *yield* annotations to demarcate transactional boundaries [12]. AME simplifies the reasoning burden by assuming a sequential-by-default semantics for demarcated transactions. A runtime system, similar to a transactional memory runtime, is responsible for *enforcing* the serializability of these transactions. Accordingly, AME has many of the same difficulties as with transactional memory.

```
int bal;
void accountUpdate(int amt) {
  yield;
  bal = bal + amt; //serializable by runtime
  yield;
}
```

We explain how our approach, called cooperability, improves over these prior approaches in the next section.

3. Approach and Uniqueness

To move beyond preemptive semantics, we propose an approach based on cooperability. *Cooperability* is a concurrency property that ensures all thread interference in a program is explicitly specified with a *yield* annotation [25, 24]. These yield annotations are purely for specification purposes and have no run-time effect. Similarly to AME, *all* code is placed inside *some* serializable transaction, where every transactional boundary is marked with a yield annotation. The key difference is that in cooperability, static and dynamic program analysis techniques *check* that each transaction thus specified is serializable. For example, the yield in the following code concisely shows the undesired thread interference:

```
int bal;
// correctly annotated code, no change in behavior
// but now incorrect behavior is obvious
void accountUpdate(int amt) {
  int tmp = bal;
  yield;
  bal = tmp + amt;
}
```

Several benefits arise from this design, which are significant improvements over the prior approaches of data race freedom, atomicity, transactional memory, and AME. In particular, no other approach has all three benefits listed below:

- Sequential semantics is correct by default: cooperability guarantees that any section of code unbroken by yield annotations exhibits sequential behavior.

  In the presence of explicit yield annotations, sequential semantics generalizes into *cooperative semantics*: context switches (and thread interference) appear to occur only at yield annotations. In essence, we are free to ignore the thread interleavings that don't matter – context switching at yields captures all interesting interleaving behavior. Note that although we are free to use cooperative semantics to understand the program, the program's parallelism is never constrained by the use of cooperative semantics.

- The difficult problem of verifying program correctness is now decomposed into two simpler subproblems:

  1. Verify that the program is cooperable, i.e., that yields specify all thread interference.

     This problem is amenable to mechanical verification, whereby static and dynamic program analysis tools can effectively verify or refute cooperability. These tools eliminate the accidental complexity of manually identifying thread interference in a program, and let programmers quickly focus on the essential complexity of how thread interact.

  2. Verify that the program is correct using cooperative semantics.

     This problem is much easier than verifying program correctness using preemptive semantics or atomicity, for the following two reasons: a) Under cooperative semantics, the points of thread interference are already syntactically specified with yield annotations, and these annotations are the only interference points one must consider. b) The number of yield annotations for a given program is at least an order of magnitude smaller than the number of interference points in a different semantics, such as preemptive semantics.

- Legacy programs, which are often sensitive to changes, may now have a cooperative semantics without changing its behavior. Since yield annotations do not affect the preemptive execution behavior of a program, they may always be safely placed to specify thread interference. In contrast, other approaches based on enforcement are potentially much more disruptive, and may unintentionally change the behavior of these legacy applications.

Cooperability is unique: no prior work combines yield annotations with program analysis. Figure 1 indicates how cooperability relates to the prior approaches of atomicity, transactional memory, and AME. Data race freedom is not depicted in this diagram, since it is a low-level property dealing with variable accesses, while the other properties are higher-level in the sense that they all specify varying degrees of thread non-interference.

4. Results and Contributions

Building on this approach, our research has produced results which demonstrate that cooperability is attainable, effective, and desirable.

Tools to check cooperability. We have developed program analysis tools, both static and dynamic, to check for cooperability. Together, these tools reduce the manual labor involved in identifying thread interference, placing yield annotations, and ensuring cooperability for a program. Furthermore, we show that these tools are correct and efficient.

COPPER is a dynamic analysis tool that detects *cooperability violations*: unspecified thread interference. COPPER observes the execution trace of the target program, and uses a graph-based algorithm to verify that this observed trace is serializable with respect to the yield-demarcated transactions. In particular, it reports an error if the yield annotations are not sufficient to capture all thread interference. One drawback of any dynamic analysis is the scheduling coverage problem: it is difficult to observe all possible traces and verify serializability for each trace. Hence, COPPER is more suitable for use to *refute* cooperability, finding counterexamples traces that are not serializable.

Static analysis is a compile-time approach to examining program behavior. One particular benefit is that we can construct sound analyses that hold for all possible traces. Type and effect systems are a standard approach to achieving a lightweight static analysis. We developed JCC, a type and effect system to verify that yield annotations in a program capture all possible thread interference. The type system is based on Lipton's theory of reduction [15], which characterizes when a sequence of actions from one thread form a serializable transaction.

Tool to infer yield annotations. Cooperability has wider applicability if we can lower the annotation barrier: the tedium of placing the initial yield annotations in an existing program. We present SILVER, a preliminary inference tool that allows us to quickly obtain a minimal set of yield annotations for existing programs. SILVER works as a dynamic analysis: if a thread's action would violate serializability by causing unspecified thread interference, then SILVER automatically inserts an implicit yield just before the offending action, thus avoiding the cooperability violation. Due to the inherent scheduling coverage problem, SILVER may under-approximate the set of required yields. Nonetheless, we found the inferred yield annotations to be extremely helpful.

Program	Size (lines)	Annot. Time (min.)	Annot. Count	Interference Points			Unintended Yields
				Preemptive	Atomic	Yields	
java.util.zip.Inflater	317	9	0	38	0	0	0
java.util.zip.Deflater	381	7	1	44	0	0	0
java.lang.StringBuffer	1,276	20	1	207	9	1	1
java.lang.String	2,307	15	2	154	5	1	0
java.io.PrintWriter	534	40	71	54	69	26	9
java.util.Vector	1,019	25	7	183	19	1	1
java.util.zip.ZipFile	490	30	42	81	69	30	0
sparse	868	15	13	231	41	8	0
tsp	706	10	34	358	365	19	0
elevator	1,447	30	31	367	134	23	0
raytracer-fixed	1,915	10	36	445	96	28	2
sor-fixed	958	10	20	200	137	13	0
moldyn-fixed	1,352	10	29	922	651	25	0
Total	13,570	231	287	3,284	1,595	175	13

Figure 2. Interference Points and Unintended Yields

Experimental data. We obtained empirical data on how cooperability improves over preemptive semantics and atomicity. As a proxy for comparing the set of traces in each semantics, we instead compare the number of yield annotations versus the number of preemptive interference points and atomic interference points, in Figure 2.

Preemptive interference points consist of shared variable accesses and lock acquires in a program, and provide a good upper-bound approximation to reasoning under preemptive semantics, since we count only accesses to shared variables and locks, while ignoring the majority of expressions that cannot participate in thread interference, such as method calls and lock releases. *Atomic interference points* consist of preemptive interference points inside non-atomic methods, as well as any calls to atomic methods, and provides a good upper-bound to reasoning under atomicity. We present interference density as a metric to capture the difficulty of program reasoning, where the *interference density* of a program for a particular set of interference points is the number of interference points in that set per line of code.

Empirical studies of Java programs indicate that typically, very few yield annotations are necessary: the density of yield annotations averaged roughly 1% of lines of code. In comparison, the interference density for preemptive interference points averaged 29% of lines of code. This order of magnitude reduction in interference density reveals the relative simplicity and efficiency of yield annotations in determining where to consider real thread interference.

Additionally, anecdotal evidence suggests that applying cooperability to legacy programs reveals bugs. The final column in Figure 2 shows the number of unintended yields in each program. These are program points that may suffer from thread interference in ways that we determined, based on manual code inspection, are unintentional. These unintended yields, 13 in total, highlight concurrency bugs, such as atomicity violations and data races. In particular, our benchmarks contain a number of previously identified concurrency errors [8, 10].

User study. A user study we conducted demonstrates that yield annotations are associated with a statistically significant improvement in the ability of programmers to identify synchronization defects [21].

Cooperative methodology. As described above, our cooperative methodology decomposes the difficulty of program verification under preemptive semantics into two simpler steps: a cooperability check (done automatically with tools), followed by program verification under cooperative semantics – comparatively a much simpler task.

By enabling cooperative reasoning for preemptive executions, we believe that cooperability can significantly reduce the reasoning burden for multithreaded programs.

References

[1] M. Abadi, C. Flanagan, and S. N. Freund. Types for safe locking: Static race detection for Java. *Transactions on Programming Languages and Systems (TOPLAS)*, 28(2):207–255, 2006.

[2] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, 1996.

[3] C. Blundell, E. Christopher, L. Milo, and M. K. Martin. Subtleties of transactional memory atomicity semantics. *Computer Architecture Letters*, 5:2006, 2006.

[4] M. D. Bond, K. E. Coons, and K. S. McKinley. PACER: Proportional detection of data races. In *Conference on Programming Language Design and Implementation (PLDI)*, 2010.

[5] C. Boyapati and M. Rinard. A parameterized type system for race-free Java programs. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2001.

[6] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2009.

[7] J. Erickson, M. Musuvathi, S. Burckhardt, and K. Olynyk. Effective data-race detection for the kernel. In *Operating Systems Design and Implementation (OSDI)*, 2010.

[8] C. Flanagan and S. Freund. Adversarial memory for detecting destructive races. In *Conference on Programming Language Design and Implementation (PLDI)*, 2010.

[9] C. Flanagan and S. N. Freund. FastTrack: Efficient and precise dynamic race detection. In *Conference on Programming Language Design and Implementation (PLDI)*, 2009.

[10] C. Flanagan, S. N. Freund, M. Lifshin, and S. Qadeer. Types for atomicity: Static checking and inference for Java. *Transactions on Programming Languages and Systems (TOPLAS)*, 30(4):1–53, 2008.

[11] C. Flanagan, S. N. Freund, and J. Yi. Velodrome: A sound and complete dynamic atomicity checker for multithreaded programs. In *Conference on Programming Language Design and Implementation (PLDI)*, 2008.

[12] M. Isard and A. Birrell. Automatic mutual exclusion. In *Workshop on Hot Topics in Operating Systems (HOTOS)*, 2007.

[13] J. R. Larus and R. Rajwar. *Transactional Memory*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2006.

[14] E. A. Lee. The problem with threads. *Computer*, 39, 2006.

[15] R. J. Lipton. Reduction: A method of proving properties of parallel programs. *Communications of the ACM*, 18(12):717–721, 1975.

[16] N. D. Matsakis and T. R. Gross. A time-aware type system for data-rrace protection and guaranteed initialization. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2010.

[17] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. In *Conference on Programming Language Design and Implementation (PLDI)*, 2006.

[18] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2003.

[19] P. Pratikakis, J. S. Foster, and M. Hicks. Context-sensitive correlation analysis for detecting races. In *Conference on Programming Language Design and Implementation (PLDI)*, 2006.

[20] S. Qadeer and D. Wu. Kiss: keep it simple and sequential. In *Conference on Programming Language Design and Implementation (PLDI)*, 2004.

[21] C. Sadowski and J. Yi. Applying usability studies to correctness conditions: A case study of cooperability. In *Onward! Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU)*, 2010.

[22] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems (TOCS)*, 15(4), 1997.

[23] H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3), 2005.

[24] J. Yi and C. Flanagan. Effects for cooperable and serializable threads. In *Workshop on Types in Language Design and Implementation (TLDI)*, 2010.

[25] J. Yi, C. Sadowski, and C. Flanagan. Cooperative reasoning for preemptive execution. In *Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2011.