Analysis of Multithreaded Programs

Martin Rinard

Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139
rinard@lcs.mit.edu,
WWW home page: http//:www.cag.lcs.mit.edu/~rinard

Abstract. The field of program analysis has focused primarily on sequential programming languages. But multithreading is becoming increasingly important, both as a program structuring mechanism and to support efficient parallel computations. This paper surveys research in analysis for multithreaded programs, focusing on ways to improve the efficiency of analyzing interactions between threads, to detect data races, and to ameliorate the impact of weak memory consistency models. We identify two distinct classes of multithreaded programs, activity management programs and parallel computing programs, and discuss how the structure of these kinds of programs leads to different solutions to these problems. Specifically, we conclude that augmented type systems are the most promising approach for activity management programs, while targeted program analyses are the most promising approach for parallel computing programs.

1 Introduction

Multithreading is a widely used structuring technique for modern software. Programmers use multiple threads of control for a variety of reasons: to build responsive servers that interact with multiple clients, to run computations in parallel on a multiprocessor for performance, and as a structuring mechanism for implementing rich user interfaces. In general, threads are useful whenever the software needs to manage a set of tasks with varying interaction latencies, exploit multiple physical resources, or execute largely independent tasks in response to multiple external events.

Developing analyses for multithreaded programs can be a challenging activity. The primary complication is characterizing the effect of the interactions between threads. The obvious approach of analyzing all interleavings of statements from parallel threads fails because of the resulting exponential analysis times. A central challenge is therefore developing efficient abstractions and analyses that capture the effect of each thread's actions on other parallel threads.

Researchers have identified several ways to use the results of analyzing multithreaded programs. Multithreading enables several new kinds of programming errors; the potential severity of these errors and difficulty of exposing them via

testing has inspired the development of analyses that detect these errors statically. Most of the research in this area has focused on detecting data races (which occur when two threads access the same data without synchronization and one of the accesses is a write) and deadlocks (which occur when threads are permanently blocked waiting for resources). Researchers have also developed optimizations for multithreaded programs; some of these optimizations generalize existing optimizations for sequential programs while others are specific to multithreaded programs.

After surveying research in the analysis and optimization of multithreaded programs, we discuss the issues associated with detecting data races in more depth. We first identify two distinct classes of multithreaded programs, *activity management programs*, which use threads to manage a set of conceptually concurrent activities, and *parallel computing programs*, which use threads to execute computations in parallel for performance on a multiprocessor. For activity management programs, we conclude that the appropriate mechanism is an augmented type system that guarantees that the program is free of data races. Because such a type system would provide information about the potential interactions between parallel threads, it could also serve as a foundation for new, very precise analyses. For parallel computing programs, we conclude that the appropriate mechanism is a set of specialized analyses, each tailored for a specific concurrency and data usage pattern.

The remainder of the paper is structured as follows. Section 2 surveys uses of the analysis information while Section 3 discusses the analyses researchers have developed for multithreaded programs, focusing on ways to improve the efficiency of analyzing interactions between parallel threads. Sections 4 and 5 discuss data race detection for activity management programs and parallel computing programs, respectively. Section 6 presents several issues associated with the use of weak memory consistency models. We conclude in Section 7.

2 Analysis Uses

Researchers have proposed several uses for analysis information extracted from multithreaded programs. The first use is to enable optimizations, both generalizations of traditional compiler optimizations to multithreaded programs and optimizations that make sense only for multithreaded programs. The second use is to detect anomalies in the parallel execution such as data races or deadlock.

2.1 Optimization Uses

A problem with directly applying traditional compiler optimizations to multithreaded programs is that the optimizations may reorder accesses to shared data in ways that may be observed by threads running concurrently with the transformed thread [72]. One approach is to generalize standard program representations, analyses, and transformations to safely optimize multithreaded programs even in the presence of accesses to shared data [91, 87, 57, 62, 56, 64]. The presence

of multithreading may also inspire optimizations with no obvious counterpart in the optimization of sequential programs. Examples include communication optimizations [59, 100], optimizing mutual exclusion synchronization [30, 31, 79, 3, 98, 11, 13, 21, 82], and optimizing barrier synchronization [96]. A more conservative approach is to ensure that the optimizations preserve the semantics of the original program by first identifying regions of the program that do not interact with other threads, then applying optimizations only within these regions. The analysis problem is determining which statements may interact with other threads and which may not. Escape analysis is an obvious analysis to use for this purpose — it recognizes data that is captured within the current thread and therefore inaccessible to other threads [11, 21, 98, 13, 82]. The programming model may also separate shared and private data [92, 89, 81, 58], in some cases the analysis may automatically infer when pointers point to private data [65]. More elaborate analyses may recognize actions (such as acquiring a mutual exclusion lock or obtaining the only existing reference to an object) that temporarily give the thread exclusive access to specific objects potentially accessed by multiple threads. A final approach is to expect the programmer to correctly synchronize the program, then enable traditional compiler optimizations within any region that does not contain an action (for example, a synchronization action or thread creation action) that is designed to mediate interactions between threads [78]. This approach has the advantage that it eliminates the need to perform a potentially expensive interthread analysis as a prerequisite for applying traditional optimizations to multithreaded programs. The (serious) disadvantage is that optimization may change the result that the program computes.

2.2 Data Race Detection

In an unsafe language like C, there are a number of program actions that are almost always the result of programmer error, regardless of the context in which they occur. Examples include array bounds violations and accessing memory after it has been deallocated. If the program engages in these actions, it can produce behavior that is very difficult to understand. Several well-known language design and implementation techniques (garbage collection, array bounds checks) can completely eliminate these kinds of errors. The cost is additional execution overhead and a loss of programmer control over aspects of the program's execution. The result was that, for many years, the dominant programming programming language (C) provided no protection at all against this class of errors.

For programs that use threads, an analogous error is a data race, which occurs when multiple threads access the same data without an intervening synchronization operation, and one of the accesses is a write. A data race is almost always the result of a programming error, with a common outcome being the corruption of the accessed data structures. The fact that data races may show up only intermittently due to different timings on different executions adds an extra layer of complexity not present for sequential programs. The most widely used multithreaded languages, Java, C, and C++ (augmented with a threads package), leave the programmer totally responsible for avoiding data races by correctly

synchronizing the computation. The result is that many systems builders view the use of threads as an inherently unsafe programming practice [76].

Presented with this problem, researchers have developed a set of analyses for determining if a program may have a data race. Some analyses allow the programmer to declare an association between data and locks, then check that the program holds the lock whenever it accesses the corresponding data [94, 28]. Other analyses trace the control transfers associated with the use of synchronization constructs such as the `post` and `wait` constructs used in parallel dialects of Fortran [71, 18, 36, 17], the Ada rendezvous constructs [95, 99, 33, 70, 35], or the Java `wait` and `notify` constructs [73, 74]. The goal is to determine that the synchronization actions temporally separate conflicting accesses to shared data. In some cases it may be important to recognize that parallel tasks access disjoint regions of the same data structure. Researchers have developed many sophisticated techniques for extracting or verifying this kind of information. There are two broad categories: analyses that characterize the accessed regions of dense matrices [8, 53, 7, 50, 77, 9, 5, 38, 47, 84], and analyses that extract or verify reachability properties of linked data structures [60, 19, 51, 43, 85]. Although many of these analyses were originally developed for the automatic parallelization of sequential programs, the basic approaches should generalize to handle the appropriate kinds of multithreaded programs. Researchers have also developed dynamic race detection algorithms, which monitor a running program to detect races in that specific execution [32, 88, 20], but provide no guarantees about other executions.

Despite the sophistication of existing static techniques, the diversity and complexity of sharing patterns in multithreaded programs means that the static data race detection problem is still far from solved. In fact, as we discuss further in Section 4, we believe the ultimate solution for most programs will involve an augmented type system that eliminates the possibility of data races at the language level.

2.3 Deadlock Detection

Researchers have developed a variety of analyses for detecting potential deadlocks in Ada programs which use rendezvous synchronization [95, 99, 69, 29, 66, 34, 24, 16, 12]. A rendezvous takes place between a call statement in one thread and an accept statement in another. The analyses match corresponding calls and accepts to determine if every call will eventually participate in a rendezvous. If not, the program is considered to deadlock. We note that deadlock traditionally arises from circular waiting to acquire resources, and is a classic problem in multithreaded programs. In this context, programs typically use mutual exclusion synchronization rather than rendezvous synchronization. We expect that a deadlock detection analysis for programs that use mutual exclusion synchronization would obtain a partial order on the acquired resources and check that the program always respects this order. The order could be obtained from the programmer or extracted automatically from an analysis of the program.

3 Analysis Algorithms

We next discuss some of the issues that arise when applying standard approaches to analyze multithreaded programs. We focus on ways to improve the efficiency of analyzing interactions between different threads.

3.1 Dataflow Analysis For Multithreaded Programs

Dataflow analysis performs an abstract interpretation of the program to discover program invariants at each program point [55, 54, 26]. Conceptually, one can view these analyses as propagating information along control-flow paths, an approach that works reasonably well for sequential programs in part because each statement typically has few direct control-flow successors. The straightforward generalization of this approach to multithreaded programs would propagate information between statements of parallel threads [27, 22]. The issue is that the direct control-flow successors of a statement in one thread typically include most if not all of the statements in all parallel threads. Propagating information along all of these potential control-flow edges leads to an algorithm with intractable execution times. The driving question is how to reduce the number of paths that the analysis must explicitly consider.

Control-Flow Analysis One approach is to analyze the program's use of synchronization constructs to discover regions of tasks that may not execute concurrently, then remove edges between these regions. The characteristics of the analysis depend on the specific synchronization constructs. Researchers have developed algorithms for programs that use the `post` and `wait` constructs used in parallel dialects of Fortran [18, 36, 17], for the Ada rendezvous constructs [95, 33, 70, 35], and for the Java `wait` and `notify` constructs [73, 74]. The basic idea behind these algorithms is to match each blocking action (such as a `wait` or `accept`) with its potential corresponding trigger actions (such as `post` or `notify`) from other threads. The analysis uses the information to determine that the statements before the trigger action must execute before the statements after the blocking action.

In general, the algorithms for `post` and `wait` constructs are designed to work within parallel loops that access dense matrices. These programs use the `post` and `wait` constructs to ensure that a write to an array element in one parallel loop iteration precedes reads to that same element in other iterations. The techniques therefore focus on correlating the array accesses with the corresponding `post` and `wait` constructs that order them. The algorithms for the Ada rendezvous and Java `wait` and `notify` constructs tend to be most effective for programs in which the threads execute different code, enabling the analysis to distinguish between threads at the level of the code that each thread executes. We expect the algorithms to be less effective for server programs in which many threads execute the same code [61].

Coarsening The Analysis Granularity Another way to reduce the analysis time is to collect adjacent instructions from threads into larger groups, then treat each group as a unit in the interthread analysis [97, 45, 23, 75]. The typical approach is to collect together instructions that do not interact with other threads; in this case the resulting coarsening of the analysis granularity does not affect the precision of the final analysis result. Because the relevant interactions usually take place at instructions from different threads that access the same data, the presence of references may significantly complicate the determination of which instructions may interact with other threads. One approach is to interleave a pointer analysis with the analysis that determines the instructions that may interact with other threads [23, 25], another approach would use the results of a previous efficient pointer analysis to find these instructions (candidate analyses include flow-insensitive analyses [93, 4] and analyses that do not analyze interleavings of instructions from different threads [83]).

Interference-Based Analyses Interference-based analyses maximally coarsen the analysis granularity — they analyze each thread as a unit to compute a result that characterizes all potential interactions with other threads. The extracted analysis information then flows from the end of each thread to the beginning of all other parallel threads. For standard bitvector analyses such as live variables and reaching definitions, this approach somewhat surprisingly delivers an efficient algorithm with the same precision as an algorithm that explicitly analyzes all possible interleavings [57]. For more complicated analyses such as pointer analysis, existing algorithms based on this approach overestimate the effect of potential interactions between threads and lose precision [83, 86]. Finally, if the language semantics rules out the possibility of interactions between tasks, analyzing each task as a unit seems obviously the correct way to proceed [46].

3.2 Flow-Insensitive Analyses

Unlike dataflow analyses, flow-insensitive analyses produce the same result regardless of the order in which the statements appear in the program or the number of times that they are executed [93, 4, 37]. They therefore trivially extend to handle multithreaded programs. The analysis results can be used directly or as a foundation to enhance the effectiveness of more detailed flow-sensitive analyses.

3.3 Challenges

The primary challenge for analyzing multithreaded programs remains developing abstractions and analyses that precisely characterize interactions between threads. For explicit interactions that take place at synchronization constructs, the primary goal is to match interacting pairs of constructs. For implicit interactions that take place at memory locations accessed by multiple threads, the primary goal is to find instructions that access the same memory locations, then

characterize the combined effect of the instructions. The use of dynamic memory allocation, object references, and arrays significantly complicates the analysis of these implicit interactions because they force the analysis to disambiguate the accesses to avoid analyzing interactions that never occur when the program runs. The problem is especially acute for programs that use references because interactions between instructions that access references may, in turn, affect the locations that other instructions access. One of the main challenges is therefore to develop efficient disambiguation analyses for multithreaded programs. We see several potential foundations for these analyses: an augmented type system (see Section 4), efficient interference-based or flow-insensitive pointer analyses, or exploiting structured control constructs such as parallel loops to confine the concurrency to a small part of the program and enable the use of very precise, detailed analyses.

Many existing analyses assume a very simple model of multithreaded execution characterized by the absence of one or more of dynamic object creation, dynamic thread creation, references to objects (including thread objects), and procedure or method calls. Given the pervasive use of these constructs in many multithreaded programs, an important challenge is to develop algorithms that can successfully analyze programs that use these constructs.

4 Data Race Freedom in Activity Management Programs

Given the problems associated with data races and the current inability of automated techniques to verify that a range of programs are free of data races, techniques that guarantee data race freedom are of interest. The primary issue that shapes the field is the reason for using multiple threads and the resulting data usage patterns of the program. In this section we focus on *activity management programs*, or programs that use threads to manage a set of conceptually parallel activities such as interacting with a remote client [10, 49]. Because of the loose connection between the computations of the threads, these programs typically use an unstructured form of concurrency in which each thread executes independently of its parent thread. These programs typically manipulate several different kinds of data with different synchronization requirements. To successfully verify data race freedom for these programs, the implementation must take these differences into account and use algorithms tailored for the properties that are relevant for each kind of data.

- **Private Data:** Data accessed by only a single thread.
- **Inherited Data:** Data created or initialized by a parent thread, then passed as a parameter to a child thread. Once the child threads starts its execution, the parent thread no longer accesses the data.
- **Migrating Data:** Data that is passed between parallel threads, often as part of producer/consumer relationships. Although multiple threads access migrating data, at each point in time there is a single thread that has conceptual ownership of the data and no other threads access the data until ownership changes.

- **Published Data:** Data that is initialized by a single thread, then distributed to multiple reader threads for read-only access.
- **Mutex Data:** Data that is potentially accessed and updated by multiple parallel threads, with the updates kept consistent with mutual exclusion synchronization.
- **Reader/Writer Data:** An extension of mutex data to support concurrent access by readers and exclusive access by writers.

Program actions temporally separate accesses from different threads and ensure data race freedom. For inherited data, the thread creation action separates the parent accesses from the child accesses. For mutex and reader/writer data, the lock acquire and release actions separate accesses from different threads. For published data, the action that makes a reference to the data accessible to multiple reader threads separates the writes of the initializing thread from the reads of the reader threads. For migrating data, the actions that transfer ownership of the data from one thread to the next separate the accesses. Mutex, published, and migrating data often work together to implement common communication patterns in multithreaded programs. For example, a shared queue usually contains mutex data (the queue header) and migrating data (the elements of the queue).

Given the diversity of the different kinds of data and the complexity of their access patterns, we believe it will be extremely difficult for any analysis to automatically reconstruct enough information to verify data race freedom in the full range of activity management programs. We therefore focus on language mechanisms that enable the programmer and the analysis to work together to establish that the program is free of data races.

4.1 Augmented Type Systems for Race-Free Programs

Many of the first researchers to write multithreaded programs were acutely aware of the possibility of data races, and developed languages that prevented the programmer from writing programs that contained them. The basic idea was to force each thread to acquire exclusive ownership of data before writing it, either by acquiring a lock on the data or by ensuring that the data is inaccessible to other threads. Concurrent Pascal, for example, carefully limits the use of references to ensure that the sharing between threads takes place only via data copied into and out of mutex data encapsulated in monitors [15]. In effect, the language uses copy operations to convert migrating, inherited, and published data into private data. Because these copy operations take place in the context of a synchronized update to mutex data, they execute atomically with respect to the threads sharing the data. It is possible to generalize this approach to handle a wider range of data structures, including linked data structures containing references [6].

Another approach is to provide an augmented type system that enables the programmer to explicitly identify shared data accessible to multiple threads [40, 41, 14]. Each piece of shared data is associated with a mutual exclusion lock and

the type system enforces the constraint that the program holds the associated lock whenever it accesses the corresponding shared data. The type system may also support a variety of other kinds of data that can be safely accessed without synchronization; examples include private data accessible to only a single thread, constant data that is never modified once it has been initialized, and value data that may be copied into and out of shared data. It is also possible to use a linear type system to ensure the existence of at most one reference to a given piece of data, with the data owned by the thread that holds its reference [14]. In this scenario, the movements of inherited and migrating data between threads correspond to acquisitions and releases of the unique reference to the moving data.

In spirit, these type systems extend the basic safe monitor approach developed in the 1970s to work for modern languages with linked data structures. The key challenge is controlling the use of references to eliminate the possibility of inadvertently making unsynchronized data reachable to multiple threads concurrently. Note that the most general solution to this problem would be to track all references to inherited, migrating, or published data and verify that threads did not use these references to incorrectly access the data. The difficulty of solving this general problem inspired the variety of other, more constrained, solutions described above.

4.2 Future Directions in Augmented Type Systems

The next step is to use some combination of language design and program analysis to better understand the referencing behavior of the program and support a wider range of thread interaction patterns. We anticipate that the implementation will focus on inherited, migrating, and published data. We view the situation for mutex and read/write data as comparatively settled — current type systems or their relatively straightforward generalizations should be adequate for ensuring that mutex data is correctly synchronized. The implementation will therefore focus on extracting or verifying the following kinds of information:

- **Reachability:** We anticipate that the implementation will use reachability information to verify the correct use of private, migrating, and inherited data. Specifically, it will verify that private data is reachable only from the thread that initially created the data and that when an ownership change takes place for inherited or migrating data, the data is inaccessible to the previous owner.
- **Write Checking:** For published data, which is reachable to multiple threads, the implementation must verify that the data is never written once it becomes accessible to multiple threads. There are two key components: identifying the transition from writable to read only, and verifying the absence of writes after the transition.

  For read/write data, we anticipate that programmers will use locking constructs that enable reads to execute concurrently but serialize writes with respect to all other accesses. The implementation must verify that all reads

are protected by a held read lock and all writes are protected by a held write lock.

4.3 Impact on Other Analyses

Because the augmented type information would enable the analysis to dramatically reduce the number of potential interthread interactions that it must consider, we expect it to enable researchers to develop quite precise and practical analyses that extract or verify detailed properties of the shared data. We anticipate an approach that divides the program into atomic regions that access only shared or private data, then analyzes the program at the granularity of these regions. The analysis would analyze sequential interactions between regions from the same thread and some subset of the interleaved interactions between regions from different threads that access the same data, obtaining a result valid for all interleavings that might occur when the program runs. In effect, the analysis would view each region as an operation on shared or private data. Potentially extracted or verified properties include representation invariants for shared data, monotonicity properties of operations on shared data, and recognition of sets of commuting operations on shared data.

4.4 Adoption Prospects

For activity management programs, we anticipate that it will be both technically feasible and valuable to develop an expressive augmented type system that guarantees data race freedom. The key question is whether such a type system would be accepted in practice. Factors that would influence its acceptance include how widespread multithreaded programming becomes, the ability of programmers to develop programs without data races in the absences of such a type system, the consequences of the data races programmers leave in the code, how well the extended type system supports the full range of thread interaction patterns, and whether programmers perceive the extended information as a burden or a benefit. One potential approach might separate the extended type information from the rest of program, enabling programmers to use the standard type system for sequential programs and the extended system for multithreaded programs. Another approach might provide standard defaults that work for most cases, with the programmer adjusting the defaults only when necessary. We note that over time, sequential languages have moved towards providing more safety guarantees, which argues for acceptance of increased safety in multithreaded languages.

5 Data Race Freedom in Parallel Computing Programs

Parallel computing programs use threads to subdivide a single computational task into multiple parallel subtasks for execution on a multiprocessor. Unlike

activity management programs, parallel computing programs often execute a sequence of steps, with the concurrency exploited within but not between steps. The structure therefore closely corresponds to the structure one would use for a sequential program that performed the same computation. Because different steps may use the same piece of data in different ways, it is crucial for the implementation to identify the threads in different phases and treat each phase separately. The difficulty of identifying parallel phases depends on the specific concurrency generation constructs. If the program uses long-lived threads that persist across steps but periodically synchronize at a barrier, reconstructing the structure is a challenging analysis problem [2]. If the program uses structured control constructs such as parallel loops or recursively generates parallel computations in a divide and conquer fashion [42], the parallel phases are obvious from the syntactic structure of the program.

Parallel computing programs use many of the same kinds of data as activity management programs. An additional complication is the fact that the parallel tasks often access disjoint parts of the same data structure. Over the years researchers have developed many sophisticated techniques for extracting or verifying this kind of information, both for programs that access dense matrices [8, 53, 7, 50, 77, 9, 5, 38, 47, 84] and for programs that manipulate linked data structures [60, 19, 51, 43, 85]. Parallel computing programs may also use reductions and commuting operations, in which case it may be important to generalize algorithms from the field of automatic parallelization to verify that the program executes deterministically [39, 44, 48, 80]. In general, the programmer can reasonably develop programs with quite sophisticated access patterns and data structures, with the data race freedom of the program depending on the detailed properties of the data structures and the algorithms that manipulate them. It therefore seems unlikely that a general approach would be able to verify data race freedom for the full range of parallel computing programs.

Because of the close correspondence between the parallel and sequential versions of the program, it is often useful to view the threading constructs in parallel computing programs as annotations that express the programmer's expectations about the lack of dependences between parts of the program rather than as constructs that must generate parallel computation to preserve the semantics of the program. In this context, the analysis problem would be framed as a sequential program analysis that determines whether the identified parts of the program lack dependences. An advantage of this approach is that it eliminates the need to analyze interactions between parallel threads.

In general, we view guaranteed data race freedom as both less feasible and potentially less important for parallel computing programs than for activity management programs. It is less feasible because it may depend on very detailed properties of arbitrarily sophisticated array access patterns or linked data structures. It is potentially less important because the parallelism tends to be confined within single parallel algorithms rather than operating across the entire execution of the program. While the algorithms in parallel computing programs may have very complicated internal structure, the fact that the potential inter-

actions can be localized significantly increases the programmer's ability to avoid inadvertent data races. Somewhat paradoxically, these properties raise the value of automatic program analysis algorithms that can verify the data race freedom of parallel computing programs. There is room for a suite of targeted analyses, each of which is designed to analyze programs that access a certain kind of data in a certain way. The ability to confine the concurrency within a small part of the program makes it feasible to use very detailed, precise analyses.

6 Weak Memory Consistency Models

For a variety of performance reasons, many implementations of multithreaded languages have a weak memory consistency model that allows the implementation to change the order in which writes from one thread are observed in parallel threads [1, 78]. Moreover, standard weak consistency models enable executions in which different threads observe different orders for the same sequence of writes from a parallel thread. Weak consistency models are often considered to be counterintuitive because they break the abstraction of a single memory accessed by sequentially executing threads [52].

One might wonder how programmers are expected to successfully develop programs in languages with weak memory consistency models. Conceptually, weak consistency models do not reorder writes across synchronization operations. So the intention is that programmers will write properly synchronized, data-race free programs and never observe the reorderings. It is worth noting that weak consistency models are complex enough that researchers are still in the process of developing a rigorous semantics for them [67, 68]. And the proposed semantics are significantly more complicated than the standard semantics for multithreaded programs, which simply interleave the statements from parallel threads.

6.1 Short-Term Program Analysis Opportunities

In the short term, weak memory consistency models will be a fact of life for developers of multithreaded software. Most modern processors implement weak consistency models in hardware, and Java specifies a weak consistency model for multithreaded programs, in part because if threads can access shared data without synchronization, many standard compiler optimizations may change the order in which threads perform (and other threads potentially observe) accesses to shared data [78]. In this context, the alternative to a weak consistency model is to disable these optimizations unless the compiler performs the global analysis required to determine that parallel threads do not observe the reordered memory accesses [59, 64]. Requiring the extraction of this kind of global information as part of the standard compilation process is clearly problematic, primarily because it rules out optimized separate compilation.

Another approach is to develop analyses and transformations that restore the abstraction of a single consistent shared memory with no reordered writes. The basic idea is to analyze the program, discover situations in which the threads

may observe reordered writes, then augment the program with additional instructions that prevent the hardware from reordering these writes [90, 63]. This research holds out the promise of providing the efficiency of a weak memory consistency model in the implementation combined with the abstraction of a single shared memory for the programmer. Because programs do not observe the effect of a weak consistency model unless they access shared data without explicit synchronization, we see these techniques as appropriate primarily for low-level programs that synthesize their own custom synchronization operations out of shared memory.

6.2 Impact on Existing Analysis Algorithms

Almost all existing analyses for multithreaded programs assume an interleaving model of concurrency. But weak consistency models generally increase the set of possible program behaviors as compared with the standard interleaving model, raising the possibility that existing analyses are unsound in the presence of weak consistency models. Furthermore, the complexity of the semantics for programs with weak consistency models increases the difficulty of developing provably sound analyses for these programs. We suspect that many existing analyses are sound for programs with weak consistency models [4, 93, 37, 57, 83], but this soundness is clearly inadvertent, in some cases a consequence of imprecision in the analysis, and not necessarily obvious to prove formally.

We expect the difficulty of dealing with weak memory consistency models to inspire multiphase approaches. The first phase will either verify the absence of data races or transform the program to ensure that it does not observe any of the possible reorderings. The subsequent phases will then assume the simpler interleaving model of concurrency. Another alternative would be to use an augmented type system that guarantees race-free programs (see Section 4). The analysis could use the type information to identify regions within which it could aggressively reorder accesses to optimize the program without changing the result that the program computes.

7 Conclusion

Multithreaded programs are significantly more complicated to analyze than sequential programs. Many analyses have focused on characterizing interactions between threads to detect safety problems such as data races and deadlock or to hide anomalies associated with weak memory consistency models. Future directions include generalizing abstractions and analyses to better handle constructs such as dynamically allocated memory, dynamic thread creation, procedures and methods, and threads as first-class objects. We also anticipate the further development of augmented type systems for race-free programs, which will reduce the potential interthread interactions that the analysis must consider and enable the development and use of more detailed, precise analyses.

References

1. S. Adve and K. Gharachorloo. Shared memory consistency models: a tutorial. *IEEE Computer*, 29(12):66–76, Dec. 1996.
2. A. Aiken and D. Gay. Barrier inference. In *Proceedings of the 25th Annual ACM Symposium on the Principles of Programming Languages*, Paris, France, Jan. 1998. ACM.
3. J. Aldrich, C. Chambers, E. Sirer, and S. Eggers. Static analyses for eliminating unnecessary synchronization from Java programs. In *Proceedings of the 6th International Static Analysis Symposium*, Sept. 1999.
4. L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994.
5. D. Bacon, S. Graham, and O. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–420, Dec. 1994.
6. D. Bacon, R. Strom, and A. Tarafdar. Guava: A dialect of Java without data races. In *Proceedings of the 15th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, Portland, OR, Oct. 2000.
7. V. Balasundaram and K. Kennedy. A technique for summarizing data access and its use in parallelism enhancing transformations. In *Proceedings of the SIGPLAN '89 Conference on Program Language Design and Implementation*, Portland, OR, June 1989.
8. U. Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, Boston, MA, 1988.
9. U. Banerjee, R. Eigenmann, A. Nicolau, and D. Padua. Automatic program parallelization. *Proceedings of the IEEE*, 81(2):211–243, Feb. 1993.
10. A. Birrell. *Systems Programming with Modula-3*, chapter An Introduction to Programming with Threads. Prentice-Hall, Englewood Cliffs, N.J., 1991.
11. B. Blanchet. Escape analysis for object oriented languages. application to Java. In *Proceedings of the 14th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, Denver, CO, Nov. 1999.
12. J. Blieberger, B. Burgstaller, and B. Scholz. Symbolic dataflow analysis for detecting deadlocks in Ada tasking programs. In *Proceedings of the 5th International Conference on Reliable Software Technologies Ada-Europe 2000*, June 2000.
13. J. Bogda and U. Hoelzle. Removing unnecessary synchronization in Java. In *Proceedings of the 14th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, Denver, CO, Nov. 1999.
14. C. Boyapati and M. Rinard. A parameterized type system for race-free Java programs. In *Proceedings of the 16th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, Tampa Bay, FL, Oct. 2001.
15. P. Brinch-Hansen. The programming language Concurrent Pascal. *IEEE Transactions on Software Engineering*, SE-1(2):199–207, June 1975.
16. E. Bruneton and J. Pradat-Peyre. Automatic verification of concurrent Ada programs. In *Proceedings of the 4th International Conference on Reliable Software Technologies Ada-Europe 1999*, June 2000.
17. D. Callahan, K. Kennedy, and J. Subhlok. Analysis of event synchronization in a parallel programming tool. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Seattle, WA, Mar. 1990.
18. D. Callahan and J. Subhlok. Static analysis of low-level synchronization. In *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, Madison, WI, May 1988.

19. D. Chase, M. Wegman, and F. Zadek. Analysis of pointers and structures. In *Proceedings of the SIGPLAN '90 Conference on Program Language Design and Implementation*, pages 296–310, White Plains, NY, June 1990. ACM, New York.
20. G. Cheng, M. Feng, C. Leiserson, K. Randall, and A. Stark. Detecting data races in Cilk programs that use locks. In *Proceedings of the 10th Annual ACM Symposium on Parallel Algorithms and Architectures*, June 1998.
21. J. Choi, M. Gupta, M. Serrano, V. Sreedhar, and S. Midkiff. Escape analysis for Java. In *Proceedings of the 14th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, Denver, CO, Nov. 1999.
22. J. Chow and W. Harrison III. Compile time analysis of programs that share memory. In *Proceedings of the 19th Annual ACM Symposium on the Principles of Programming Languages*, Albuquerque, NM, Jan. 1992. ACM, New York.
23. J. Chow and W. Harrison III. State space reduction in abstract interpretation of parallel programs. In *Proceedings of the 1994 IEEE International Conference on Computer Language*, May 1994.
24. J. Corbett. Evaluating deadlock detection methods for concurrent software. *IEEE Transactions on Software Engineering*, 22(3), Mar. 1996.
25. J. Corbett. Using shape analysis to reduce finite-state models of concurrent Java programs. In *Proceedings of the International Symposium on Software Testing and Analysis*, Mar. 1998.
26. P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the 4th Annual ACM Symposium on the Principles of Programming Languages*, Los Angeles, CA, 1977.
27. P. Cousot and R. Cousot. *Automatic Program Construction Techniques*, chapter Invariance Proof Methods and Analysis Techniques for Parallel Programs. Macmillan Publishing Company, New York, NY, 1984.
28. D. Detlefs, K. R. Leino, G. Nelson, and J. Saxe. Extended static checking. Technical Report 159, Compaq Systems Research Center, 1998.
29. L. Dillon. Using symbolic execution for verification of Ada tasking programs. *ACM Transactions on Programming Languages and Systems*, 12(4), 1990.
30. P. Diniz and M. Rinard. Synchronization transformations for parallel computing. In *Proceedings of the 24th Annual ACM Symposium on the Principles of Programming Languages*, pages 187–200, Paris, France, Jan. 1997. ACM, New York.
31. P. Diniz and M. Rinard. Lock coarsening: Eliminating lock overhead in automatically parallelized object-based programs. *Journal of Parallel and Distributed Computing*, 49(2):2218–244, Mar. 1998.
32. A. Dinning and E. Schonberg. Detecting access anomalies in programs with critical sections. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, Santa Cruz, CA, May 1991.
33. E. Duesterwald and M. Soffa. Concurrency analysis in the presence of procedures using a data-flow analysis framework. In *Proceedings of the ACM Symposium on Analysis, Verification, and Testing*, Victoria, B.C., Oct. 1991.
34. S. Duri, U. Buy, R. Devarapalli, and S. Shatz. Application and experimental evaluation of state space reduction methods for deadlock analysis in Ada. *ACM Transactions on Software Engineering and Methodology*, 3(4), Oct. 1994.
35. M. Dwyer and L. Clarke. Data-flow analysis for verifying properties of concurrent programs. In *Proceedings of the 2nd ACM SIGSOFT Symposium on the Foundations of Software Engineering*, New Orleans, LA, Dec. 1994.

36. P. Emrath, S. Ghosh, and D. Padua. Event synchronization analysis for debugging parallel programs. In *Proceedings of Supercomputing '89*, Reno, NV, Nov. 1989. IEEE Computer Society Press, Los Alamitos, Calif.
37. M. Fahndrich, J. Foster, Z. Su, and A. Aiken. Partial online cycle elimination in inclusion constraint graphs. In *Proceedings of the SIGPLAN '98 Conference on Program Language Design and Implementation*, Montreal, Canada, June 1998.
38. P. Feautrier. Compiling for massively parallel architectures: A perspective. *Microprogramming and Microprocessors*, 1995.
39. A. Fisher and A. Ghuloum. Parallelizing complex scans and reductions. In *Proceedings of the SIGPLAN '94 Conference on Program Language Design and Implementation*, pages 135–144, Orlando, FL, June 1994. ACM, New York.
40. C. Flanagan and M. Abadi. Types for safe locking. In *Proceedings of the 8th European Symposium on Programming*, Amsterdam, The Netherlands, Mar. 2000.
41. C. Flanagan and S. Freund. Type-based race detection for Java. In *Proceedings of the SIGPLAN '00 Conference on Program Language Design and Implementation*, Vancouver, Canada, June 2000.
42. M. Frigo, C. Leiserson, and K. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the SIGPLAN '98 Conference on Program Language Design and Implementation*, Montreal, Canada, June 1998.
43. R. Ghiya and L. Hendren. Is it a tree, a DAG or a cyclic graph? A shape analysis for heap-directed pointers in C. In *Proceedings of the 23rd Annual ACM Symposium on the Principles of Programming Languages*, pages 1–15, Jan. 1996.
44. A. Ghuloum and A. Fisher. Flattening and parallelizing irregular, recurrent loop nests. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 58–67, Santa Barbara, CA, July 1995. ACM, New York.
45. P. Godefroid and P. Wolper. A partial approach to model checking. In *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science*, Amsterdam, The Netherlands, July 1991.
46. D. Grunwald and H. Srinivasan. Data flow equations for explicitly parallel programs. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Diego, CA, May 1993.
47. M. Gupta, S. Mukhopadhyay, and N. Sinha. Automatic parallelization of recursive procedures. In *Proceedings of the 1999 Conference on Parallel Algorithms and Compilation Techniques (PACT) '99*, Newport Beach, CA, Oct. 1999.
48. M. Hall, S. Amarasinghe, B. Murphy, S. Liao, and M. Lam. Detecting coarse-grain parallelism using an interprocedural parallelizing compiler. In *Proceedings of Supercomputing '95*, San Diego, CA, Dec. 1995. IEEE Computer Society Press, Los Alamitos, Calif.
49. C. Hauser, C. Jacobi, M. Theimer, B. Welch, and M. Weiser. Using threads in interactive systems: A case study. In *Proceedings of the Fourteenth Symposium on Operating Systems Principles*, Asheville, NC, Dec. 1993.
50. P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, July 1991.
51. L. Hendren, J. Hummel, and A. Nicolau. Abstractions for recursive pointer data structures: Improving the analysis and transformation of imperative programs. In *Proceedings of the SIGPLAN '92 Conference on Program Language Design and Implementation*, San Francisco, CA, June 1992. ACM, New York.
52. M. Hill. Multiprocessors should support simple memory consistency models. *IEEE Computer*, 31(8), Aug. 1998.

53. F. Irigoin and R. Triolet. Supernode partitioning. In *Proceedings of the 15th Annual ACM Symposium on the Principles of Programming Languages*, San Diego, CA, Jan. 1988.
54. J. Kam and J. Ullman. Global data flow analysis and iterative algorithms. *Journal of the ACM*, 23(1):159–171, Jan. 1976.
55. G. Kildall. A unified approach to global program optimization. In *Conference Record of the Symposium on Principles of Programming Languages*. ACM, Jan. 1973.
56. J. Knoop and B. Steffen. Code motion for explicitly parallel programs. In *Proceedings of the 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Atlanta, GA, May 1999.
57. J. Knoop, B. Steffen, and J. Vollmer. Parallelism for free: Efficient and optimal bitvector analyses for parallel programs. *ACM Transactions on Programming Languages and Systems*, 18(3):268–299, May 1996.
58. A. Krishnamurthy, D. Culler, A. Dusseau, S. Goldstein, S. Lumetta, T. von Eicken, and K. Yelick. Parallel programming in Split-C. In *Proceedings of Supercomputing '93*, Nov. 1993.
59. A. Krishnamurthy and K. Yelick. Analyses and optimizations for shared address space programs. *Journal of Parallel and Distributed Computing*, 38(2), Nov. 1996.
60. J. Larus and P. Hilfinger. Detecting conflicts between structure accesses. In *Proceedings of the SIGPLAN '88 Conference on Program Language Design and Implementation*, Atlanta, GA, June 1988. ACM, New York.
61. D. Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley, Reading, Mass., San Mateo, CA, 1996.
62. J. Lee, S. Midkiff, and D. Padua. A constant propagation algorithm for explicitly parallel programs. *International Journal of Parallel Programming*, 26(5), 1998.
63. J. Lee and D. Padua. Hiding relaxed memory consistency with compilers. In *Proceedings of the 2000 International Conference on Parallel Algorithms and Compilation Techniques*, Philadelphia, PA, Oct. 2000.
64. J. Lee, D. Padua, and S. Midkiff. Basic compiler algorithms for parallel programs. In *Proceedings of the 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Atlanta, GA, May 1999.
65. B. Liblit and A. Aiken. Type systems for distributed data structures. In *Proceedings of the 27th Annual ACM Symposium on the Principles of Programming Languages*, Boston, MA, Jan. 2000.
66. D. Long and L. Clarke. Data flow analysis of concurrent systems that use the rendezvous model of synchronization. In *Proceedings of the ACM Symposium on Analysis, Verification, and Testing*, Victoria, B.C., Oct. 1991.
67. J. Maessen, Arvind, and X. Shen. Improving the Java memory model using CRF. In *Proceedings of the 15th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, Portland, OR, Oct. 2000.
68. J. Manson and W. Pugh. Core multithreaded semantics for Java. In *Proceedings of the Joint ACM Java Grande - ISCOPE 2001 Conference*, Stanford, CA, June 2001.
69. S. Masticola and B. Ryder. Static infinite wait anomaly detection in polynomial time. In *Proceedings of the 1990 International Conference on Parallel Processing*, St. Charles, IL, Aug. 1990.
70. S. Masticola and B. Ryder. Non-concurrency analysis. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Diego, CA, May 1993.

71. S. Midkiff and D. Padua. Compiler algorithms for synchronization. *IEEE Transactions on Computers*, 36(12):1485–1495, Dec. 1987.
72. S. Midkiff and D. Padua. Issues in the optimization of parallel programs. In *Proceedings of the 1990 International Conference on Parallel Processing*, pages II–105–113, 1990.
73. G. Naumovich, G. Avrunin, and L. Clarke. Data flow analysis for checking properties of concurrent Java programs. In *Proceedings of the 21st International conference on Software Engineering*, Los Angeles, CA, May 1999.
74. G. Naumovich, G. Avrunin, and L. Clarke. An efficient algorithm for computing MHP information for concurrent Java programs. In *Proceedings of the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Toulouse, France, Sept. 1999.
75. G. Naumovich, L. Clarke, and J. Cobleigh. Using partial order techniques to improve performance of data flow analysis based verification. In *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, Toulouse, France, Sept. 1999.
76. J. Ousterhout. Why threads are a bad idea (for most purposes). Invited Talk at the 1996 USENIX Technical Conference.
77. W. Pugh. A practical algorithm for exact array dependence analysis. *Commun. ACM*, 35(8):102–114, Aug. 1992.
78. W. Pugh. Fixing the Java memory model. In *Proceedings of the ACM 1999 Java Grande Conference*, San Francisco, CA, June 1999.
79. M. Rinard. Effective fine-grain synchronization for automatically parallelized programs using optimistic synchronization primitives. *ACM Transactions on Computer Systems*, 17(4):337–371, Nov. 1999.
80. M. Rinard and P. Diniz. Commutativity analysis: A new analysis technique for parallelizing compilers. *ACM Transactions on Programming Languages and Systems*, 19(6):941–992, Nov. 1997.
81. M. Rinard and M. Lam. The design, implementation, and evaluation of jade. *ACM Transactions on Programming Languages and Systems*, 20(3):483–545, May 1998.
82. E. Ruf. Effective synchronization removal for Java. In *Proceedings of the SIGPLAN '00 Conference on Program Language Design and Implementation*, Vancouver, Canada, June 2000.
83. R. Rugina and M. Rinard. Pointer analysis for multithreaded programs. In *Proceedings of the SIGPLAN '99 Conference on Program Language Design and Implementation*, Atlanta, GA, May 1999.
84. R. Rugina and M. Rinard. Symbolic bounds analysis of pointers, array indexes, and accessed memory regions. In *Proceedings of the SIGPLAN '00 Conference on Program Language Design and Implementation*, Vancouver, Canada, June 2000.
85. M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems*, 20(1):1–50, Jan. 1998.
86. A. Salcianu and M. Rinard. Pointer and escape analysis for multithreaded Java programs. In *Proceedings of the 8th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Snowbird, UT, June 2001.
87. V. Sarkar and B. Simons. Parallel program graphs and their classification. In *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, Aug. 1993.
88. S. Savage, M. Burrows, G. Nelson, P. Solbovarro, and T. Anderson. Eraser: A dynamic race detector for multi-threaded programs. In *Proceedings of the*

Sixteenth Symposium on Operating Systems Principles, Saint-Malo, France, Oct. 1997.

89. D. Scales and M. S. Lam. The design and evaluation of a shared object system for distributed memory machines. In *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation*. ACM, New York, Nov. 1994.
90. D. Shasha and M. Snir. Efficient and correct execution of parallel programs that share memory. *ACM Transactions on Programming Languages and Systems*, 10(2), Apr. 1988.
91. H. Srinivasan, J. Hook, and M. Wolfe. Static single assignment for explicitly parallel programs. In *Proceedings of the 20th Annual ACM Symposium on the Principles of Programming Languages*, Jan. 1993.
92. G. Steele. Making asynchronous parallelism safe for the world. In *Proceedings of the 17th Annual ACM Symposium on the Principles of Programming Languages*, pages 218–231, San Francisco, CA, Jan. 1990. ACM, New York.
93. B. Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd Annual ACM Symposium on the Principles of Programming Languages*, St. Petersburg Beach, FL, Jan. 1996.
94. N. Sterling. Warlock: A static data race analysis tool. In *Proceedings of the 1993 Winter Usenix Conference*, Jan. 1994.
95. R. N. Taylor. A general purpose algorithm for analyzing concurrent programs. *Commun. ACM*, 26(5):362–376, May 1983.
96. C. Tseng. Compiler optimizations for eliminating barrier synchronization. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 144–155, Santa Barbara, CA, July 1995.
97. A. Valmari. A stubborn attack on state explosion. In *Proceedings of the 2nd International Workshop on Computer Aided Verification*, New Brunswick, NJ, June 1990.
98. J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *Proceedings of the 14th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, Denver, CO, Nov. 1999.
99. M. Young and R. Taylor. Combining static concurrency analysis with symbolic execution. *IEEE Transactions on Software Engineering*, 14(10), Oct. 1988.
100. H. Zhu and L. Hendren. Communication optimizations for parallel C programs. In *Proceedings of the SIGPLAN '98 Conference on Program Language Design and Implementation*, Montreal, Canada, June 1998.