UCRL-JRNL-226950

Formal Specification of the OpenMP Memory Model

G. Bronevetsky, B. de Supinski

December 21, 2006

International Journal of Parallel Programming

Formal Specification of the OpenMP Memory Model

Greg Bronevetsky[1] and Bronis R. de Supinski[2]

[1] Department of Computer Science,
Cornell University,
Ithaca, NY 14850, USA,
greg@bronevetsky.com,
[2] Center for Applied Scientific Computing,
Lawrence Livermore National Laboratory,
Livermore, CA 94551, USA
bronis@llnl.gov

Abstract. OpenMP [2] is an important API for shared memory programming, combining shared memory's potential for performance with a simple programming interface. Unfortunately, OpenMP lacks a critical tool for demonstrating whether programs are correct: a formal memory model. Instead, the current official definition of the OpenMP memory model (the OpenMP 2.5 specification [2]) is in terms of informal prose. As a result, it is impossible to verify OpenMP applications formally since the prose does not provide a formal consistency model that precisely describes how reads and writes on different threads interact.
We expand on our previous work that focused on the formal verification of OpenMP programs through a formal memory model [**?**]. As in that work, our formalization, which is derived from the existing prose model [2], provides a two-step process to verify whether an observed OpenMP execution is conformant. This paper extends the model to cover the entire specification. In addition to this formalization, our contributions include a discussion of ambiguities in the current prose-based memory model description. Although our formal model may not capture the current informal memory model perfectly, in part due to these ambiguities, our model reflects our understanding of the informal model's intent. We conclude with several examples that may indicate areas of the OpenMP memory model that need further refinement, however it is specified. Our goal is to motivate the OpenMP community to adopt those refinements eventually, ideally through a formal model, in later OpenMP specifications.

1 Introduction

Modern systems are increasingly being built using multi-threaded architectures. These include systems with multiple processors on the same node and/or multiple cores on the same chip. Given the proximity of the processors/cores on such machines, they typically feature a single memory accessible to any processor. As such, these machines are easily and effectively programmed in a multi-threaded shared memory style.

OpenMP [2] has emerged as a popular shared memory API because it combines the performance advantages of shared memory with an easy-to-use API. However, despite the relative simplicity of the API, OpenMP applications remain difficult to write. The difficulty arises from several inherent complexities of multi-threaded execution, including non-determinism, a large space of possible executions and a very relaxed memory consistency model. Thus, although OpenMP allows programmers to improve application performance significantly, this comes at a cost of significantly higher program complexity. This complexity makes OpenMP programs much more vulnerable to bugs than sequential programs and thus, more expensive to debug. Ultimately, confidence in the correctness of the final application is reduced.

Formal verification is a family of techniques that formalize a program or protocol into a mathematically well-defined form. Correctness is verified using a variety of techniques that range in their complexity and their correctness guarantees, from model checking to theorem proving [11]. While formal verification is generally too complex to apply to real-world applications, it is feasible for the basic algorithms on which they are based.

Existing work on formally verifying shared memory algorithms [10] requires us to represent the entire computational content of the algorithm formally, including algorithm logic and the details of the underlying system. In particular the underlying memory model must be formalized. While some formal memory models

exist [8] [4], none exists for OpenMP. Instead, the official description of OpenMP's memory model (section 1.4 of version 2.5 of the OpenMP specification [2]) is written in detailed English, which is generally clear but not nearly precise enough for formal verification tasks. Similarly, while the OpenMP memory model was recently clarified further [7], this clarification is also informal.

We expand on our previous work [**?**] that focused on verification of OpenMP programs through a formal memory model that we derived from the existing prose model [2]. Our formalization provides a two-step process to verify if an observed OpenMP execution is conformant. In contrast to our previous work, the model presented here covers the full 2.5 specification. We also provide a more detailed description on how our formalization represents OpenMP programs. In addition to this formalization, we discuss ambiguities in the current prose-based memory model description. Although our formal model may not capture the current informal memory model perfectly, in part due to these ambiguities, this formalization reflects our understanding of the informal model's intent. We present several examples that demonstrate a need for further refinement of the OpenMP memory model. Our goal is to motivate the OpenMP community eventually to adopt those refinements, ideally through a formal model, in later OpenMP specifications.

This paper is divided as follows. Section 2 provides an overview of the OpenMP memory model. Section 3 discusses aspects of that model that we find ambiguous (despite one of the authors having significant input into it). Section 4 outlines the formalization of this model. Section 5 defines the language of the operations used in the formal model. Sections 6 and 7 provide the details of the two phases used by the formal specification. Finally, section 8 provides several example programs and their outcomes under the formal model specified in this paper.

2 Outline of the OpenMP Memory Model

The OpenMP memory model provides for two types of memory: shared and threadprivate. There is a single shared memory that is visible to reads and writes on all threads. Furthermore, each thread has its own threadprivate memory that is accessible to only the reads and writes on that thread. OpenMP's shared memory semantics are akin to but a little weaker than weak ordering [5]. While each thread may read from and write to data in shared memory, there is no guarantee that one thread can immediately observe a write by another thread. Thus, the value associated with a given read may not reflect all prior writes from other threads. Instead, each thread conceptually has a *temporary view* of shared memory and a `flush` operation limits the reordering of operations and synchronizes a thread's temporary view with shared memory.

Simple, intuitive concepts motivate the OpenMP memory model. In order to ensure that a read by thread j returns the value of a write by thread i, the program must provide synchronization that guarantees the following sequence of events:

1. Thread i writes to the variable
2. Thread i flushes the variable
3. Thread j flushes the variable
4. Thread j reads the variable

and no other writes to the variable are happening at the same time. Any behavior outside the above sequence can produce undefined read results and/or leave the variable's value in shared memory undefined. However, the OpenMP memory model is very complex with many potential pitfalls in practice, despite the simplicity of the underlying concepts, as we will discuss.

A thread's temporary view can be its cache, registers or other devices that speed up memory operations by not forcing the processor to go to main memory for every shared access. Reads and writes to shared variables access the thread's temporary view of shared memory. If the thread reads a shared variable and the temporary view doesn't hold a value for this variable, the read goes directly to shared memory. If a thread writes to a shared variable, it only updates the thread's temporary view of that variable. However, the system is then free to non-deterministically push the value of the write from a thread's temporary view to shared memory at any time. Since there are no atomicity constraints (e.g., a 64-bit write may not be executed as a single operation), if two writes executed on two threads are not ordered via synchronization, the value of the variable in shared memory may become garbage and is thus undefined (until it is overwritten by some

later write). Similarly, if a write to a variable and a read from the same variable are executed on different threads and are not related via appropriate flushes and synchronization, the value read is undefined.

In addition to uncertainty about when shared reads and writes will actually access shared memory, OpenMP allows the compiler and the hardware to execute application operations out of order relative to their order in the original source code (called "program order"). In particular, implementations are allowed to reorder shared operations that access different shared memory variables. It is not specified whether it is legal to reorder operations that do have a data dependence (ex: A=B and B=1), although it is possible to imagine aggressive compiler transformations that may do that.

OpenMP's `flush` operation is the application's primary means of limiting the asynchrony of memory and the degree of out-of-order execution. A given `flush` operation applies to a list of shared variables and has two major effects:

- it synchronizes the thread's temporary view with shared memory for the variables in the list;
- it prevents reordering of the thread's operations on variables in the list.

The first effect ensures that any preceding writes to the list variables by the thread have completed in the shared memory before the `flush` completes. It also ensures that the first read that follows the `flush` to each of the list variables must come directly from shared memory. The second effect ensures that shared memory operations that access a variable in the `flush`'s variable list are executed in program order relative to the `flush`. Furthermore, all `flush` operations with overlapping variable lists must be executed in program order.

A program's `flush` operations also restrict the interleaving of operations by different threads. All threads must observe any two `flush` operations with overlapping variable lists in some sequential order. This makes it possible to organize non-flush operations on different threads into a partial temporal order that in turn determines which writes are visible to which reads.

OpenMP provides several synchronization operations that can be used to explicitly order operations on different threads. These operations are necessary because of the great difficulty of implementing synchronization using OpenMP's basic reads, writes and flushes. Synchronization operations include `locks`, `barriers`, `critical` sections, `ordered` sections and `atomic` updates. All of these operations are preceded and/or followed by implied `flush` operations that apply either to all variables or just the variable involved in the operation.

3 Ambiguities in the OpenMP Memory Model

Despite the precise prose that defines the OpenMP memory model, formulating a formal memory model has uncovered some questions about the model's meaning. Some of the questions indicate ambiguities that should be resolved in future specifications. Other questions arise from discrepancies between the prose and our understanding of the intent of the OpenMP language committee. We discuss several of these questions in this section.

3.1 Dependence-breaking Compilers

The OpenMP memory model clearly defines reordering restrictions with respect to flush operations. However, reordering restrictions for non-`flush` operations are much less clear. For example, most sequential compilers

reorder operations that access different variables; does the memory model allow these? While the specification makes it clear that the intent is to allow such reorderings, this is supported with only this sentence: "The flush operation restricts reordering of memory operations that an implementation might otherwise do." However, the model goes further, stating that "OpenMP does not apply restrictions to the reordering of memory operations executed by a single thread except for those related to a flush operation." This appears to imply that compilers may reorder any other operations, including those that access the same shared variable. In particular, they can reorder not only reads but also writes, as long as these writes are not separated by a flush to the variable and as long as this preserves the application's sequential semantics.

```
if(threadNum!=0)
    A=5;  Barrier if(threadNum==0)
    A=20;
Barrier
if(threadNum!=0) {
    B=5;
    B=A;
    print B;
}
```

For example, in this sample code the application's sequential semantics would be preserved if the two writes to B were exchanged or the write $B = A$ eliminated, since in a single-threaded execution the write $B = A$ is guaranteed to assign 5 to B. However, if this code were to be executed by two threads, the write $B = A$ would assign B to 20, rather than 5. As such, reordering these two writes, while apparently legal in OpenMP and in sequential execution, can in fact produce unexpected results for parallel applications. Since there exist apparently legal dependence-breaking compiler optimizations that violate the spirit of the OpenMP memory model, the OpenMP specification should include a clear statement about the validity of different types of variable access reordering.

3.2 Intra-thread Dependencies

The OpenMP memory model clearly states that a `flush` does not complete until the values of all preceding writes have been completed in shared memory. However, it is not clear if the OpenMP memory model enforces program order, i.e., processor consistency [6].

Section 2 presents the events required for a read by thread j to return the value written by thread i. If thread i writes another value between steps 1 and 2, the value of which write should be read in step 4? The question is related to the reordering questions in the preceding section, but it is also different.

If the first value is captured in the writer thread's temporary view but not the second for some reason (for example, the writes are executed out of order), is it legal not to propagate the captured value?

The memory model prose states, "the `flush` does not complete until the value of the variable has been written to the variable in memory." Simply put, the memory model does not address multiple writes to the same shared variable by the same thread between two `flush` operations. Ultimately, the question is: does OpenMP guarantee that writes by a given thread must be seen in program order by other threads as long as the appropriate `flushes` have been issued (i.e. writes, `flush`, `flush`, read)?

We can also ask about the impact of reads by thread i: suppose that thread i reads the variable between steps 1 and 2 and that value is different from what was written by the write in step 1 due to a write by some other thread. This scenario includes a race condition and the specification is clear that the variable's value becomes undefined. However, completing the write would now be inconsistent with program order. Does the race imply that the `flush` should not see the write from step 1 and the read in step 4 will get some other value? The specification provides little detail on how local state evolves so the issue is unclear.

3.3 Effect of Privatization

The memory model section, section 1.4, of the 2.5 specification [2] states that OpenMP has two types of memory: shared and threadprivate. The bulk of the section defines the semantics of the shared memory. It provides few details of the second type, which corresponds to threadprivate variables and to variables included in private clauses. The only issue discussed is the interaction with nested parallelism.

The memory model does not address any interactions between the two types. In particular, it does not discuss the impact on shared variables that are included in private clauses. However, section 2.8.3.3, which

discusses the private clause of a given region, includes: "The value of the original list item is not defined upon entry to the region. The original list item must not be referenced within the region. The value of the original list item is not defined upon exit from the region." Including a shared variable in a private clause essentially writes the shared variable with an undefined value, an effect that is easily overlooked by someone trying to understand the OpenMP memory model. We understand that this effect is being reconsidered for the OpenMP 3.0 specification. However, our point here is that any interactions between the two types of memory should be included in the memory model section. In the very least, a forward reference is needed.

3.4 Captured Writes

The OpenMP memory model states that "If a thread has captured the value of a write in its temporary view of a variable since its last `flush` of that variable, then when it executes another `flush` of the variable, the `flush` does not complete until the value of the variable has been written to the variable in memory." This appears to be ambiguous. What does it mean for a thread to capture a value of a write? Does this only refer to a write by the thread that executes the `flush`? This appears to be the intent but the actual wording could refer to writes on other threads that have been read by the given thread. The ultimate point here is that English is a rich and complex language in general and the phrase "precise English" is an oxymoron. For this reason, a formal, mathematical model is needed.

4 Formal Specification of the OpenMP Memory Model

4.1 Informal Outline of Memory Model

The OpenMP memory model defined in this paper specifies an ordering on the evaluation of operations on the same thread as well as on different threads. Operations on the same thread are ordered via their read/write/flush dependencies. The only operations that may define an order across threads are flushes, with all other inter-thread orderings derived from this flush-induced order.

4.1.1 Operations on the Same Thread Read/write dependencies restrict the order in which operations within a thread can occur. We show how to derive these restrictions for a simple example in Figure 1. Figure 1(a) shows the original application source code and Figure 1 shows the read/write dependence graph of the same operations. Figure 1(c) then takes the operations in Figures 1(a) and (b) and translates them into their constituent reads and writes, adding the appropriate dependence relations ($Read\ var \rightarrow val$ corresponds to a read of variable var that returned the value val, while $Write\ var \leftarrow val$ corresponds to a write of val to var). At runtime it is legal to execute these reads and writes in any order that agrees with this dependence order.

Flush operations create additional dependence relations, as shown in Figure 2. The source code in Figure 2(a) corresponds to the partial orders in Figures 2 (b) and (c). Since the flush operation only applies to variable $data$, it depends on the write to $data$ and the execution of the flush must follow the write to $data$ in any valid execution. However, since neither the write to $data$, nor the flush of $data$ relate to the write to $flag$, there is no dependence relationship between these operations. As such, the write to $flag$ may occur either before or after the other operations. Thus, the write to $flag$ cannot be used to signal other threads that the write to $data$ has occurred. In Figure 2(d), we replace $flush(data)$ with $flush(data, flag)$. As a result, the partial orders in Figures 2 (e) and (f) place the three operations into a total order. This ensures that during any valid execution the write to $flag$ will be executed after the flush, which will be executed after the write to $data$.

4.1.2 Operations on Different Threads Figure 3(a) shows a sample execution of an application where one thread executes a write to variable x while another thread executes two reads of x. Because the first read clearly races with the write, its output is undefined. The second read clearly follows the write in this execution but these operations also race since they are not separated by flushes, as specified in Section 2. In

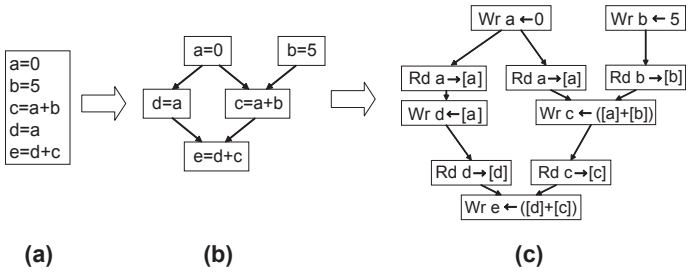

Fig. 1. Generation of the Dependence Order

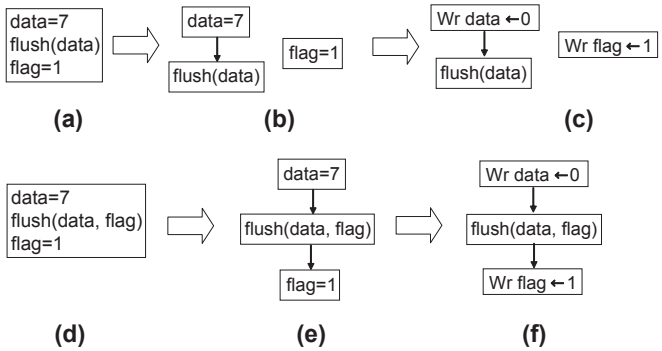

Fig. 2. Generation of the Dependence Order with Flushes

Figure 3(b), we add of two flushes, one after the write and the other before the read, that create an inter-thread dependency (shown by the dashed arrow) and eliminate the race. This dependency causes the write to precede the read, and, thus, the read returns the value written. However, the flushes are not sufficient to ensure that the write precedes the read in every execution; the use of an explicit synchronization construct, such as a lock, can provide that guarantee, as shown in Figure 3(c).

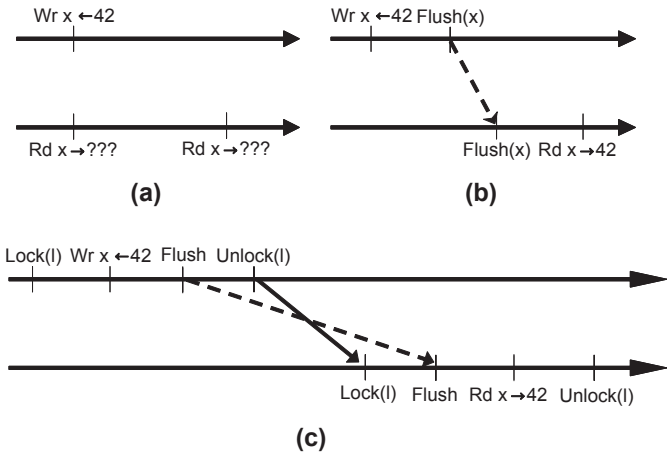

Fig. 3. Execution of Read-Write Race

Flushes are also required to ensure inter-thread dependencies between two writes. Figure 4 shows a race between unsynchronized writes to x from two threads. Flushes and reads of x then follow these writes. Since the flushes ensure that the reads follow the writes, it might appear that the reads should return valid values. However, the write race leaves the state of x undefined, making the output of subsequent reads also undefined, regardless of how well x is flushed.

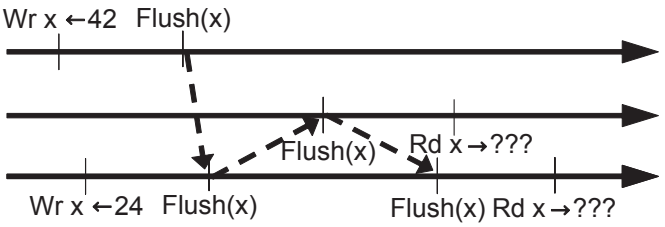

Fig. 4. Execution of Write-Write Race

4.1.3 Summary of the OpenMP Memory Model Our formal OpenMP memory model is defined in two phases. The first focuses on the operations on each thread, which are converted into individual read, write and flush operations, which are ordered by their dependencies. The second phase focuses on the runtime execution of multiple threads, where each thread's operations may be executed in any order that agrees with the dependence order established above. Parallel execution of multiple threads causes their operations to interleave in some non-deterministic order, where each particular interleaving determines the values returned by all read operations. For a given interleaving, the only way for a given read to return the value of the

most recent write is if (i) the read follows the write via the inter-thread dependence established by flushes (as specified in Section 2) and (ii) the write was not involved in a race with another write to the same variable. Synchronization operations such as locks, critical sections and barriers can be used to properly order reads, writes and flushes to ensure that all read values are well-defined in all possible executions. While certain limited forms of synchronization through variables are allowed, OpenMP's weak guarantees for racing accesses make such algorithms an advanced topic. Interested users should read the formal memory model in detail before attempting this.

4.2 Outline of the Formal Memory Model

The following sections describe the OpenMP memory model in formal, mathematical language. Our model takes as input an application and a trace of how the application executed under some OpenMP implementation. The trace includes the order in which each thread executed its operations and the values returned by all reads. The model uses a set of rules to judge if the application could have generated the trace and if there exists under the OpenMP memory model a valid interleaving of thread operations that results in the values read in the trace.

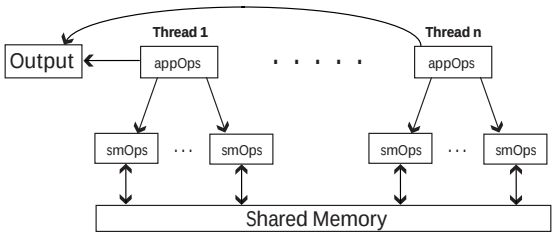

Fig. 5. Diagram of the formal memory model

Our OpenMP formalization is an operational model (outlined in Figure 5). It defines a system state and valid transition rules for modifying this state. At a high level, this model defines the state of one or more application threads running on top of shared memory and transition rules for evaluating the next application operation on some thread. Applications are specified as lists of high-level operations such as $(var_A = var_B \otimes var_C)$ and $(While(var = val)\ bodyList)$, called "application operations" or "appOps". Each appOp is made up of one or more simpler operations such as $(Read\ var_A)$ or $(Write\ var_B)$, called "shared memory operations" or "smOps". Every thread's state transition either:

- Evaluates the next smOp that makes up the thread's currently-executing appOp; or
- Moves to evaluation of the thread's next appOp in its remaining application source code.

The first action can change the shared memory state. The second action typically removes an appOp from the remaining application source code but can add appOps in the case of a while loop appOp that performs multiple loop iterations. A trace records each thread's view of a particular execution of the system. As such, it is a tuple of lists of smOps, one for each thread, (each list is some thread's "sub-trace"). Each sub-trace contains the smOps and the values associated with them during their thread's execution. Traces do not specify the interleaving of smOps from different threads.

We use the two thread execution shown in Figure 6 to illustrate the intuition of the model's operation. One thread executes the appOp $c = a \otimes b$ while the second simultaneously executes appOp $e = c \otimes d$ ($\otimes$ is some binary operation). Each appOp is composed of multiple *Read* and $Write$ smOps, which can be determined independently of the execution of the appOps. However, we must observe the execution to associate values with the read operations. Thus, a trace of this execution is two lists of smOps and their associated values: $< Rd\ a \rightarrow 6; Rd\ b \rightarrow 12; Wr\ c \leftarrow 6 \otimes 12 >$ for the top thread and $< Rd\ d \rightarrow 1; Rd\ c \rightarrow 42; Wr\ e \leftarrow 1 \otimes 42 >$ for the bottom thread. Note that we assume the system correctly computes $6 \otimes 12$ and $1 \otimes 42$, as the calculation

occurs outside of the memory system. We apply our operational memory model's rules to determine if a valid interleaving that associates the values with the smOps exists. For our example, the interleaving $< T0 : Read\ a \rightarrow 6; T1 : Read\ d \rightarrow 1; T1 : Read\ c \rightarrow 42; T0 : Read\ b \rightarrow 12; T1 : Write\ e \leftarrow 1 \otimes 42; T1 : Write\ c \leftarrow 6 \otimes 12 >$, as shown in Figure 6, verifies that execution is valid under the OpenMP memory model.

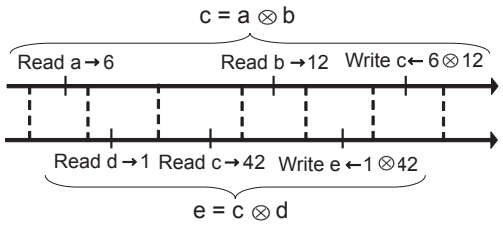

Fig. 6. Example Two Thread Execution

Although we could specify the operational model in a single set of rules, we break it into two sub-models, the Compiler Phase and the Runtime Phase. This separation makes it possible to reason independently about different aspects of the memory model: the translation from application source code into basic shared memory operations (Compiler Phase) and the results of interleaving these operations during execution (Runtime Phase).

The compiler phase evaluates each thread's source code independently from any other thread to verify that the application could have generated the list of smOps in each sub-trace. Its state consists of:

- a list of the current thread's remaining appOps;
- a list of smOps generated by this thread so far;
- the suffix of the thread's sub-trace that contains the yet unverified smOps.

During each state transition the compiler phase evaluates the next appOp, breaks it up into its constituent smOps (ex: the appOp $(var_A = var_B \otimes var_C)$ breaks up into $(Read\ var_B)$, $(Read\ var_C)$ and $(Write\ var_A)$ smOps) and checks whether these smOps are contained in the sub-trace. Since the thread's control flow depends on the values read from shared memory, whenever an appOp reads a value from shared memory (e.g., as part of $(var_A = var_B \otimes var_C)$ or $(While(var = val)\ bodyList)$), it looks them up in the sub-trace. The trace correctly corresponds to the application's source code if the compiler phase independently verifies this for each sub-trace. In addition to this verification, the compiler phase determines any ordering required by the application's data dependences. The compiler phase outputs this order for consumption by the runtime phase.

The runtime phase determines if the smOps in the individual threads' sub-traces correspond to each other. More specifically, it evaluates all of the threads' sub-traces in parallel to determine whether a conformant interleaving exists that results in the associated read values. It assumes that the smOps in the individual threads' sub-traces correspond to the application's source code (i.e. the compiler phase has already validated that aspect of the trace). Therefore, its state consists of:

- the reads, writes, flushes and synchronization operations that each thread has already performed (one list per thread);
- a partial order that relates these smOps in time (used for determining the values that a read may return);
- the system's synchronization state: currently held locks, critical and ordered sections and the identities of threads that are currently blocked on a barrier;
- the smOps that remain to be evaluated by each thread (one list per thread).

During each state transition the runtime phase chooses a thread and evaluates its pending smOp. It may evaluate smOps out of order if this does not break their data dependences determined during the compiler

phase. Evaluation of the read and atomic update smOps examines the values available to be read and verifies that the value returned by the read in the trace could actually have been read during this interleaving. Every state transition also causes the state to change, including updating the synchronization state and adding new relations to the above partial order. Since the runtime phase is non-deterministic, the trace is self-consistent if there exists some interleaving of the different threads' smOps such that all reads performed by the formal model match their return values recorded in the trace.

Section 5 details the full language of appOps and smOps. Sections 6 and 7 provide more details on the mechanics of the compiler phase and runtime phase.

5 Language Specification

5.1 Application Operations

The application language (specified below) models the major relevant features of C/Fortran and OpenMP. It contains basic computational and control flow operations as well as flushes, locks, critical section, ordered regions and barriers. Section number references refer to the OpenMP 2.5 specification [2]. The while loop primitive makes the application language Turing-complete in its use of shared memory operations.

- $var_A = var_B \otimes var_C$
  - Represents any local computation performed by the application.
  - $\otimes$ is a Turing-complete binary operation that does not use shared memory.
  - var_A, var_B and var_C are shared variables.
  - Corresponds to ($Read\ var_B$), ($Read\ var_C$) and ($Write\ var_A$) smOps.
- *Atomic var* $\oplus =$ *updVal*
  - Models the atomic update construct [section 2.7.4].
  - $\oplus$ may be one of the following operations: +, $*$, −, /, &, ^, |, <<, or >> (++ and - - are modelled via +=1 and -=1).
  - *var* is a shared variable.
  - *updVal* is a constant.
  - Corresponds to a *Read var* and an *Write var* smOp that are surrounded by ($Flush_{mm}$ (*var*)) smOps, that are themselves surrounded by *BlockSynchs* that synchronize this atomic update with other updates.
- *Flush varList*
  - Models explicit flushes [sections 1.4.2 and 2.7.5].
  - *varList* is a list of shared variables.
  - An explicit flush operation with a list maps to *Flush varList*, where *varList* is its variable list.
  - An explicit flush operation without a list maps to *Flush allVarList*, where *allVarList* contains all application shared variables.
  - Corresponds to a single $Flush_{mm}$ smOp that applies to the same *varList*.
- *BlockSynch blockF updF synchID*
  - Represents a generic blocking synchronization operation that models the synchronization semantics of higher-level operations such as locks, critical and ordered regions and barriers.
  - *blockF* is function.
    - Result depends on the formal system synchronization state.
    - Returns False if the thread may continue executing (i.e., is not blocked).
    - Returns True if the thread is blocked.
  - *updF* is a function.
    - Result depends on the formal system current synchronization state.
    - Returns the next sychronization state.
    - Applied only when *blockF* returns True.
    - Ensures the synchronization state reflects that the thread has become unblocked.
  - *blockF* and *updF* are different for each high-level synchronization construct.

  - *synchID* is the ID associated with this synchronization, such as the name of the critical section or the lock variable being acquired or released; used to order this *BlockSynch* relative to *Flush* operations.
  - *BlockSynch* blocks the parent thread until *blockF* returns True.
  - *BlockSynch* updates the thread's synchronization state using *updF* once *blockF* returns True.
  - Corresponds to a single $BlockSynch_{mm}$ smOp that applies to the same *blockF* and *updF*.
- *NonBlockSynch blockF updF synchID* $\rightarrow$ *successFlag*
  - Represents generic non-blocking synchronization operation that models the synchronization semantics of locks, specifically non-blocking lock acquires.
  - *blockF* is function, defined as in *BlockSynch*.
  - *updF* is a function, defined as in *BlockSynch*.
  - *synchID* is defined as in *BlockSynch*.
  - *NonBlockSynch* evaluates *blockF* to determine whether it can synchronize successfully.
    - If *blockF* returns True, *NonBlockSynch* returns True as the *successFlag* and updates the thread's synchronization state using *updF*.
    - If *blockF* returns False, *NonBlockSynch* returns False as the *successFlag*.
  - Corresponds to a single $NonBlockSynch_{mm}$ smOp that applies to the same *blockF* and *updF*.
- $While(var = testVal)$ *bodyList*
  - A while loop control flow primitive.
  - *var* is a shared variable.
  - *testVal* is a value.
  - *bodyList* is a list of appOps.
  - Corresponds to a single (*Read var*) smOp.
- *Print var*
  - Outputs the value of a given shared variable to the user; primarily used in examples to reason about outcomes of application executions.
  - *var* is a shared variable.
  - Corresponds to a single (*Read var*) smOp.
- *End*
  - The last operation in the application's source code.
  - Ensures each thread's sub-trace ends correctly.

5.2 Shared Memory Operations

The shared memory operation language is designed to be simple but sufficient for the functionality needs of the higher-level appOps. The smOps include reads, writes, flushes and blocking synchronizations (from which higher-level synchronizations are built) and are detailed below.

- *Write var* $\leftarrow$ *val*: writes *val* to variable *var*.
  - *var* is a shared variable.
  - *val* is a constant.
- *Read var* $\rightarrow$ *val*: read of variable *var* returns *val*.
  - *var* is a shared variable.
  - *val* is a constant.
- $Flush_{mm}$ *varList*: flushes this thread's temporary view variables in *varList*.
  - *varList* is a list of shared variables.
  - Updates thread's temporary view of those variables with writes from other threads and vice versa.
  - Provides flush semantics for explicit and implicit flush operations.
- $BlockSynch_{mm}$ *blockF updF*: generic synchronization operation.
  - Used to implement synchronization semantics of higher-level operations such as locks, critical and ordered regions and barriers.
  - *blockF* is function, defined as in *BlockSynch*.
  - *updF* is a function, defined as in *BlockSynch*.

  - smOp that implements the semantics of *BlockSynch* during the runtime phase.
- $NonBlockSynch_{mm}$ *blockF updF*: generic synchronization operation.
  - Used to implement synchronization semantics of higher-level operations such as locks, critical and ordered regions and barriers.
  - *blockF* is function, defined as in *BlockSynch*.
  - *updF* is a function, defined as in *BlockSynch*.
  - smOp that implements the semantics of *NonBlockSynch* during the runtime phase.

5.3 Translation of OpenMP into the Formal Language

The appOp and smOp languages presented in this paper were designed to balance simplicity against similarity to the real OpenMP specification and real OpenMP implementations. As a result, the appOps are not OpenMP constructs and the smOps do not directly map to OpenMP implementation internals. In this section, we discuss how to translate full OpenMP constructs into appOps and relate smOps to existing and future OpenMP implementations so our formal model can be applied to real OpenMP applications and implementations.

5.3.1 OpenMP to AppOps The OpenMP specification allows application programmers to implement their applications in C/C++ or Fortran with additional annotations and library calls that identify different variables as private or shared, manage threads and perform inter-thread synchronization.

5.3.1.1 Private Computations: Although OpenMP applications operate on private and shared memory, our formalism focuses on shared memory behavior. Thus, we abstract all portions of the applications that operate only on private state through the $\otimes$ operator. For our purposes, the $(var_A = var_B \otimes var_C)$ appOp corresponds to any arbitrarily complex computation $\otimes$ that uses only private data and the values of shared variables var_A and var_B. The results of this computation are written to shared variable var_C. While this abstraction greatly simplifies reasoning about the shared memory behavior of OpenMP applications, it goes too far by abstracting away too much of the dependence of application control flow on the state of shared memory. In particular, it cannot represent applications where a thread's control flow depends on shared reads, influencing the thread's subsequent choice of shared operations. This issue is overcome via the $(While(var = testVal)\ bodyList)$ appOp, which makes the appOp language Turing-complete, allowing appOps to model arbitrary complex shared memory behaviors.

5.3.1.2 Thread management: OpenMP provides the `#pragma omp parallel` directive to enable programmers to create and destroy threads and several functions such as `omp_set_max_threads` to manipulate thread creation. In contrast, our formalism does not include model thread creation or deletion but instead keeps the total number of threads static throughout the application's execution. This simplification is appropriate since our formalism is an operational model that consumes a trace of actual shared memory operations. The static number of threads can simply be set to the total number of threads in the trace.

5.3.1.3 Private vs Shared Data: The OpenMP specification includes several mechanisms for identifying different memory regions as private or shared. In contrast, our formalism ignores private variables completely, as already discussed. Further, we treat all shared data as individual variables and provide no functionality for changing the status of a variable from shared to private. Conversion from OpenMP shared data to our shared variables is simply a matter of treating each address in virtual memory as an individual variable. Privatization of shared variables can be handled by overwriting the shared variable at the time that it is privatized with an undefined value, which is consistent with the 2.5 specification.

5.3.1.4 Synchronization Constructs: OpenMP provides application programmers with a variety of synchronization constructs, including locks, barriers, critical regions, ordered regions and atomic updates. Our formalism does not individually model these OpenMP synchronization constructs. Instead, we support them with the $(BlockSynch\ blockF\ updF\ synchID)$ and $(NonBlockSynch\ blockF\ updF\ synchID)$ appOps.

BlockSynch blocks its parent thread until the *blockF* function returns false (not blocked). It then executes the *updF* function to update the application's synchronization state. *NonBlockSynch* uses the *blockF* to determine whether it can pass through the synchronization point instead of blocking. If it can, it updates the application synchronization state using *updF* and returns True. If not, it simply returns False.

We map the OpenMP synchronization constructs to ours through specific blocking and update functions. In these definitions, σ is the application's synchronization state and each type of synchronization can store its state in different fields of σ, such as $\sigma.HeldLocks$ for the lock functions. Each function takes the current σ as an argument and returns either the new σ or True/False.

- **Resource Acquire/Release Operations** for resource $resID$, where $resID$ may be a lock, a critical region or a given loop's ordered region.
  $\sigma.HeldRes$ contains tuples that describe whether each resource is held by some thread or not. Resource acquire operations (lock acquire, entry into critical or ordered region) are be translated to a *BlockSynch* that blocks to acquire the resource, followed by a $Flush$ of all variables. Resource release operations (lock release, exit from critical or ordered region) are translated to a $Flush$ of all variables, followed by a *BlockSynch* that releases the resource. Non-blocking resource acquisition operations (`omp_test_lock`) are translated to a similar pattern, replacing *BlockSynch* with *NonBlockSynch*.
  - Blocking Resource Acquire: ($BlockSynch\ acqBlockF\ acqUpdF\ resID$).
    $acqBlockF$ blocks as long as $\sigma.HeldRes$ contains an ownership record for $resID$ and $acqUpdF$ adds this ownership record to $\sigma.HeldRes$.
    $acqBlockF = (\lambda\sigma.\ \neg < resID > \ \in\ \sigma.HeldRes)$
    $acqUpdF = (\lambda\sigma.\ \sigma.HeldRes := \sigma.HeldRes \cup \{< resID >\})$.
  - Non-Blocking Resource Acquire: ($NonBlockSynch\ acqBlockF\ acqUpdF\ resID$).
    $acqBlockF$ does not block (because its used as part of the *NonBlockSynch* appOp) and instead returns to the application whether the resource is currently available. If it is, $acqUpdF$ is executed to acquire the resource. $acqBlockF$ and $acqUpdF$ are defined as above.
  - Resource Release: ($BlockSynch\ releaseBlockF\ releaseUpdF\ resID$).
    $releaseBlockF$ does not block and $releaseUpdF$ removes this resource's ownership record from $\sigma.HeldRes$.
    $releaseBlockF = (\lambda\sigma.\ False)$
    $releaseUpdF = (\lambda\sigma.\ \sigma.HeldRes := \sigma.HeldRes - \{< resID >\})$.

  The above template can be readily instantiated to apply to OpenMP locks and critical regions. Ordered regions can be dealt with by treating each loop's iterations and ordered regions as a single resource and adding logic to $acqBlockF$ and $releaseUpdF$ to track the current loop iteration number.
- **Barrier** on thread t.
  $\sigma.BarBlocked$ records for each thread whether that thread is currently blocked on a barrier. A single barrier operation corresponds to a *BlockSynch* for arrival at the barrier, a $Flush$ of all variables, followed by a *BlockSynch* for exiting the barrier. $barVar$ is a unique variable. Its appearance in the *BlockSynchs* below is used to order the *BlockSynchs* relative to $Flush$ operations. While the translation below works for barriers in the case where nested parallelism is not used, it can be easily upgraded to cover the nested case as well.
  - Barrier Arrival: ($BlockSynch\ barArrBlockF\ barArrUpdF\ barVar$).
    $barArrBlockF$ does not block and $barEntrUpdF$ updates $\sigma.BarBlocked$ to record that thread t has arrived at the barrier.
    $barArrBlockF = (\lambda\sigma.\ False)$
    $barArrUpdF = (\lambda\sigma.\ \sigma.BarBlocked[t \mapsto True])$.
  - Barrier Exit: ($BlockSynch\ barExitBlockF\ barExitUpdF\ barVar$).
    $barExitBlockF$ blocks until all threads become blocked at a barrier or this thread's blocked status is set to False by another thread. $barExitUpdF$ sets the blocked status of all threads to False when it is executed at a time when all threads are blocked on a barrier. If this is not true, it leaves σ alone.
    $barExitBlockF = (\lambda\sigma.\ \sigma.Blocked(t) = True \wedge \exists t_i.\ \sigma.Blocked(t_i) = False)$
    $barExitUpdF = (\lambda\sigma.\ if\ (\forall t_i.\ \sigma.Blocked(t_i) = True)$

$then\ \sigma.Blocked := (\lambda t_i.\ False)$
$else\ \sigma$).

– **Atomic Updates** are more complex than other synchronization operations and as such get their own appOp, which is defined here in full detail.

Note that this translation of OpenMP synchronization operations into appOps allows the application to use synchronization operations incorrectly, for example, having a thread release a lock that it doesn't hold. While appropriate synchronization semantics can be encoded in a straightforward fashion, this is not currently possible OpenMP does not define detailed semantics for such erroneous behavior. Therefore, the above encoding should be treated as a sample and will be amended if and when appropriate semantics are defined.

5.3.2 SmOps to OpenMP Implementations Because this formalization describes the semantics of the memory model as seen by the application, it is not strictly necessary for the smOps in this formalism to be at all similar to the the shared memory APIs that most OpenMP implementations are written in (indeed, some are written using message passing APIs). However, given the importance of making this specification easily understandable by OpenMP implementors, the smOps were designed to feature simple semantics that are readily translatable to concepts that underly existing and future OpenMP implementations. As such, the set of smOps is limited to reads, writes, flushes and primitive synchronizations. *Reads* and $Writes$ are fundamental operations of shared memory and variants of the $Flush$ smOp exist in almost every memory model (e.g. memory barriers or release/acquire operations). $BlockSynch$ and $NonBlockSynch$ are different in that it is unlikely for most OpenMP implementations to natively implement synchronization operations with similar semantics. However, their simplicity makes it easy to both (i) understand the relationship between synchronization operations and reads, writes and flushes as well as (ii) express real synchronization operations in terms of these primitive operations.

6 Compiler Phase

The compiler phase, diagrammed in Figure 7, independently evaluates each thread of the application. It relates the application's source code to the smOps recorded in the thread's sub-trace. The evaluation pass reads the appOps of the application source code in program order and unwraps its while loops as appropriate. In the process, it translates each appOp into its constituent smOp(s). These application smOps are looked up in the thread's sub-trace during this evaluation process to verify that they actually do appear there. The values of all shared reads are also looked up in the trace. This phase also defines a dependence order $\overrightarrow{DepO}$ on each thread's smOps, which the evaluation in the runtime phase must not violate. The remainder of this section defines the state and transition function of the compiler phase.

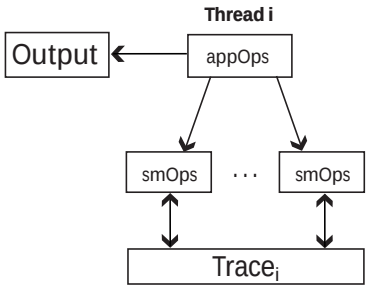

Fig. 7. Diagram of the Compiler Phase

This phase's operational model is applied to each thread's sub-trace. During every transition it evaluates the next appOp from the list of remaining appOps and verifies that its smOps occur in the sub-trace and have

the appropriate step counter labels. The phase fails if it cannot verify those smOps. Whenever an appOp's evaluation depends on the outcome of a read, the read value is looked up in the trace and used in the appOp. For example, the while loop transition behaves differently depending on whether the value returned by its read is $testVal$ or not.

The full trace is valid only if the above transition system independently passes each of its sub-traces. The Dependence Order $\overrightarrow{DepO}$ defined during the compiler pass is preserved for use in the runtime pass to ensure that whenever smOps are evaluated out of order, this new ordering does not violate their read-write dependences.

6.1 Compiler State

$[n, app, trace_{sub}, \overrightarrow{DepO}]$

- n: the number of smOps evaluated by this thread thus far. Initially $n = 0$.
- app : The list containing the appOps that remain to be evaluated by the thread. Initially, it is the original source code of the application.
- $trace_{sub}$: The list containing the thread's sub-trace that is to be validated relative to application source code. The m^{th} smOp generated on this thread during the compiler phase is listed as $< smOp, m >$ (recall that the smOps in $trace_{sub}$ may have been executed out of order, meaning that they may be listed out of program order). No two entries in $trace_{sub}$ have the same m field.
- $\overrightarrow{DepO}$: The dependence order established so far between thread's smOps; initially the empty relationship.

6.2 Compiler Transitions

The valid state transitions are shown below. Transition are specified using using Structured Operational Semantics as follows: $\frac{\text{Precondition}}{\text{Original State } \Rightarrow \text{ Next State}}$ where Precondition is the logical expression defining the conditions that must hold in order for this transition to happen, and the Original and Next states describe the transition itself. For any state variable x, x denotes its value in the original state and x' denotes its value in the next state. Given the state expression defined above, the transition format becomes:

$$\frac{\text{Conditions relating } \overrightarrow{DepO},\ \overrightarrow{DepO}',\ appOp,\ app,\ app' \text{ and } trace_{sub}.}{< n, appOp :: app, trace_{sub}, \overrightarrow{DepO} > \ \Rightarrow \ < n + c, app', trace_{sub}, \overrightarrow{DepO}' >}$$

One compiler transition exists for each appOp type. While loops have two transitions, one for the while loop performing an extra iteration and another for the while loop's termination (depends on the value read for the iteration variable). The transition used depends on the associated value of the loop variable, as described in the transitions. Whenever the partial order $\overrightarrow{DepO}$ is updated with new ordering relations, the new $\overrightarrow{DepO}$ is the transitive closure of the old $\overrightarrow{DepO}$ and the the new relations.

One compiler transition exists for each appOp type. While loops have two transitions, one for the while loop performing an extra iteration and another for the while loop's termination. Each rule does the following:

- Advances the app list to the next appOp on the list. In the case of while loops this may mean that the app list becomes longer since when the while loop iterates, its body is pushed back in the front of app.
- Identifies the smOp(s) that make up this appOp and ensures that each of these smOp(s) is in $trace_{sub}$.
- Increments the step counter n by $c =$"the number of smOps this appOp contains".
- Updates $\overrightarrow{DepO}$ to connect all of the appOp's constituent smOp(s) to each other and to previously evaluated smOps that these smOps depend on. Thus, writes are made to follow prior reads, writes and flushes to the same variable. Reads follow prior writes and flushes to the same variable. Blocking synchronizations follow prior blocking synchronizations. Flushes follow all prior operations that touch variables in their lists. All smOps must follow the read inside the most recent while loop iteration test since this test decides whether or not later smOps are executed.

6.3 Formal Definitions

// The transitive closure of the union of two partial orders
$\overrightarrow{Order_1} \uplus \overrightarrow{Order_2} \equiv \overrightarrow{Order_1 \uplus Order_2}$
$\{< op, op' > \mid \exists op'' \in (\overrightarrow{Order_1 \cup Order_2}).$
$op \overrightarrow{Order_1 \cup Order_2} op'' \overrightarrow{Order_1 \cup Order_2} op'\}$

$appVars$ = set of all shared variables that may be used directly by the application. Locks variables are included in this list but things like critical section names and barriers are not.

$allVars$ = set of all shared variables, including $appVars$ as well as the special variables: names of critical sections, ids of loops and $barVar$.

// $RelatesBefore$ is true if the given smOp relates to a variable in the given list
// and happened before the n^{th} smOp in $trace_{sub}$ and false otherwise.
$RelatesBefore(op, varsList, n, trace_{sub}) =$
// op was the m^{th} smOp and it relates to a variable in $varsList$ if ...
$\exists m < n.$
// if op is a read of a variable in the list OR
$(op =< Read\ var \rightarrow val, m > \wedge op \in trace_{sub} \wedge var \in varsList) \vee$
// if op is a write to a variable in the list OR
$(op =< Write\ var \leftarrow val, m > \wedge op \in trace_{sub} \wedge var \in varsList) \vee$
// if op is an flush with an intersecting variable list
$(op =< Flush_{mm}\ flushVarList, m > \wedge op \in trace_{sub} \wedge (flushVarList \cup varsList) \neq \emptyset)$
// if op is an $BlockSynch$ operation that relates to variable in the list
$(op =< BlockSynch\ blockF\ updF\ synchID, m > \wedge op \in trace_{sub} \wedge synchID \in varsList)$

// $BlockSynchBefore$ is true if the given smOp is a $BlockSynch$ operation that happened
// before the n^{th} smOp in $trace_{sub}$ and false otherwise.
$BlockSynchBefore(op, n, trace_{sub}) =$
// op was the m^{th} smOp and was a $BlockSynch$
$\exists m < n.\ op =< (< BlockSynch\ blockF\ updF\ synchID, m > \in trace_{sub}), m >$

$Trans_{comp}$ = set of all compiler transitions

// Definition of a legal sequence of of compiler transitions for a given
// application and trace starting with thread i's initial state
$LegalCompSeq(seq, app, trace, t_i) =$
// The sequence begins with the initial state
$seq[0] = [0, app, trace_i, \emptyset] \wedge$
// And every pair of adjacent compiler states is related via some valid
// compiler transition
$\forall n \in [0, seq.length).\ \exists transition \in Trans_{comp}.$
$transition(seq[n] \Rightarrow seq[n+1])$

// Definition of a trace being verified by the compiler phase
$ValidTraceComp(app, trace) =$
// For every thread there exists some legal sequence of compiler states
// that validates that thread's sub-trace relative to the application
$\forall t_i.\ \exists seq_i.\ LegalCompSeq(seq_i, app, trace, t_i)$

6.4 Formal Transition System

The transitions below validate the sub-trace of thread t_i.

Computation Step: $var_A = var_B \otimes var_C$

// The next operation in the source code is a computation
$app = (\mathbf{var_A} = \mathbf{var_B} \otimes \mathbf{C}) :: app'$

// All three variables are actual application variables
$var_A \in appVars \wedge var_B \in appVars \wedge var_C \in appVars$

// All three smOps that make up this appOpp appear in the sub-trace
$< Read\ var_B \rightarrow val_B, n > \in trace_{sub}$
$< Read\ var_C \rightarrow val_C, n+1 > \in trace_{sub}$
$< Write\ var_A \leftarrow (val_B \otimes val_C), n+2 > \in trace_{sub}$
// Update $\overrightarrow{DepO}$ to contain new dependencies:
$\overrightarrow{DepO}' = \overrightarrow{DepO} \uplus$
  // The write in this update depends on the reads.
  $\uplus \{<< Read\ var_B \rightarrow val_B, n >, < Write\ var_A \leftarrow (val_B \otimes val_C), n+2 >\}$
  $\uplus \{<< Read\ var_C \rightarrow val_C, n+1 >, < Write\ var_A \leftarrow (val_B \otimes val_C), n+2 >>\}$
  // The reads depend on all prior non-read smOps that relate to var_B
  // and var_C, respectively
  $\uplus \{< op_{prev}^{var_B}, < Read\ var_B \rightarrow val_B, n >> \mid$
    $RelatesBefore(op_{prev}^{var_B}, \{var_B\}, n, trace_{sub}) \wedge op_{prev}^{var_B}\ not\ a\ Read\}$
  $\uplus \{< op_{prev}^{var_C}, < Read\ var_C \rightarrow val_C, n >> \mid$
    $RelatesBefore(op_{prev}^{var_C}, \{var_C\}, n, trace_{sub}) \wedge op_{prev}^{var_C}\ not\ a\ Read\}$
  // And the write depends on all prior smOps that relate to var_A
  $\uplus \{< op_{prev}^{var_A}, < Write\ var_A \leftarrow (val_B \otimes val_C), n+2 >> \mid$
    $RelatesBefore(op_{prev}^{var_A}, \{var_A\}, n, trace_{sub})\}$
  // All operations depend on the last read that was part of a while
  // loop iteration test
  $\uplus \{< R_{prev}^{while}, < Read\ var_B \rightarrow val_B, n >> \mid R_{prev}^{while} = last\ while\ loop\ read\}$
  $\uplus \{< R_{prev}^{while}, < Read\ var_C \rightarrow val_C, n+1 >> \mid R_{prev}^{while} = last\ while\ loop\ read\}$
  $\uplus \{< R_{prev}^{while}, < Write\ var_A \leftarrow (val_B \otimes val_C), n+2 >> \mid R_{prev}^{while} = last\ while\ loop\ read\}$

$< n, app, trace_{sub}, \overrightarrow{DepO} > \Rightarrow < n+3, app', trace_{sub}, \overrightarrow{DepO}' >$

Flush Step: $Flush\ varList$

// The next operation in the source code is a flush
$app = (\textbf{Flush varList}) :: app'$

// The set of variables that this flush applies to consists exclusively of actual application variables
$(varList - appVars) = \emptyset$

// The $Flush_{mm}$ smOp that corresponds to the $Flush$ appOp must appear
// in the sub-trace
$< Flush_{mm}\ varList, n > \ \in \ trace_{sub}$
// Update $\overrightarrow{DepO}$ to contain the dependence of the flush
$\overrightarrow{DepO}' = \overrightarrow{DepO} \uplus$
  // on all previous operations that relate to variables in $varList$.
  $\uplus \{< op_{prev}, < Flush_{mm}\ varList, n >> \ | \ RelatesBefore(op_{prev}, varList, n, trace_{sub})\}$
  // and on the last read that was part of a while loop iteration test
  $\uplus \{< R^{while}_{prev}, < Flush_{mm}\ varList, n >> \ | \ R^{while}_{prev} = last\ while\ loop\ read\}$

$< n, app, trace_{sub}, \overrightarrow{DepO} > \ \Rightarrow \ < n+1, app', trace_{sub}, \overrightarrow{DepO}' >$

Blocking Synchronization: $BlockSynch\ blockF\ updF\ synchID$

// The next operation in the source code is a $BlockSynch$
$app = (\textbf{BlockSynch blockF updF synchID}) :: app'$

// The $BlockSynch_{mm}$ smOp that corresponds to the $BlockSynch$ appOp must appear
// in the sub-trace
$< BlockSynch_{mm}\ blockF\ updF, n > \ \in \ trace_{sub}$
// Update $\overrightarrow{DepO}$ to contain the dependence of the $BlockSynch$
$\overrightarrow{DepO}' = \overrightarrow{DepO} \uplus$
  // on all previous operations that relate to $synchID$.
  $\uplus \{< op_{prev}, < BlockSynch_{mm}\ blockF\ updF >> \ |$
    $RelatesBefore(op_{prev}, \{synchID\}, n, trace_{sub})\}$
  // and on the last read that was part of a while loop iteration test
  $\uplus \{< R^{while}_{prev}, < BlockSynch_{mm}\ blockF\ updF >> \ | \ R^{while}_{prev} = last\ while\ loop\ read\}$

$< n, app, trace_{sub}, \overrightarrow{DepO} > \ \Rightarrow \ < n+1, app', trace_{sub}, \overrightarrow{DepO}' >$

Non-Blocking Synchronization: $NonBlockSynch\ blockF\ updF\ synchID\ \rightarrow\ successFlag$

// The next operation in the source code is a $NonBlockSynch$
$app = (\textbf{NonBlockSynch blockF updF synchID} \ \rightarrow \ \textbf{successFlag}) :: app'$

// The $BlockSynch_{mm}$ smOp that corresponds to the $NonBlockSynch$ appOp must appear
// in the sub-trace and must return the same $successFlag$ as the $NonBlockSynch$.
$< NonBlockSynch_{mm}\ blockF\ updF \ \rightarrow \ successFlag, n > \ \in \ trace_{sub}$
// Update $\overrightarrow{DepO}$ to contain the dependence of the $NonBlockSynch$
$\overrightarrow{DepO}' = \overrightarrow{DepO} \uplus$
  // on all previous operations that relate to $synchID$.
  $\uplus \{< op_{prev}, < NonBlockSynch_{mm}\ blockF\ updF >> \ |$
    $RelatesBefore(op_{prev}, \{synchID\}, n, trace_{sub})\}$
  // and on the last read that was part of a while loop iteration test
  $\uplus \{< R^{while}_{prev}, < NonBlockSynch_{mm}\ blockF\ updF >> \ | \ R^{while}_{prev} = last\ while\ loop\ read\}$

$< n, app, trace_{sub}, \overrightarrow{DepO} > \ \Rightarrow \ < n+1, app', trace_{sub}, \overrightarrow{DepO}' >$

Atomic Update Step: $Atomic\ var \oplus =\ updVal$

// The next operation in the source code is an atomic update
$app = (\textbf{Atomic var} \oplus = \textbf{updVal}) :: app'$

// The variable is an actual application variable
$var \in appVars$

// An atomic update of var cannot start while another update is going on.
$atomicEntryBlock = (\lambda\sigma.\ \forall t_j.\ \neg < var, t_j > \in \sigma.ActiveUpdates)$
// Once it is true that no thread is updating var, thread t_i may grab the update
// token for var and begin its update.
$atomicEntryUpd = (\lambda\sigma.\ \sigma.ActiveUpdates := \sigma.ActiveUpdates \cup \{< var, t_i >\})$
// An exit may terminate its atomic update of var only if it is currently performing it
$atomicExitBlock = (\lambda\sigma.\ \neg < var, t_i > \in \sigma.ActiveUpdates)$
// And it removes var from the set of variables currently being updated
$atomicExitUpd = (\lambda\sigma.\ \sigma.ActiveUpdates := \sigma.ActiveUpdates - \{< var, t_i >\})$

// The ($Atomic\ var \oplus =\ updVal$) operation breaks up into a $Read$ and $Write$ smOps that compute the update.
// They are surrounded by $Flush_{mm}$ (var)'s that update the thread's temporary view of var for the read
// and update the memory after the write. These are themselves surrounded by $BlockSynchs$ smOps that provide
// the necessary synchronization relative to other atomic updates. All six smOps must appear in the sub-trace
$< BlockSynch\ atomicEntryBlock\ atomicEntryUpd\ var, n > \in trace_{sub}$
$< Flush_{mm}\ (var), n+1 > \in trace_{sub}$
$< Readvar \rightarrow readVal, n+2 > \in trace_{sub}$
$< Write\ var \leftarrow (readVal \oplus updVal), n+3 > \in trace_{sub}$
$< Flush_{mm}\ (var), n+4 > \in trace_{sub}$
$< BlockSynch\ atomicExitBlock\ atomicExitUpd\ var, n+5 > \in trace_{sub}$
// Update $\overrightarrow{DepO}$ to contain new dependencies:
$\overrightarrow{DepO}' = \overrightarrow{DepO} \uplus$
  // The smOps must appear in the order: $BlockSynch$, $Flush_{mm}$, $Read$, $Write$, $Flush_{mm}$, $BlockSynch$.
  $\uplus \{<< BlockSynch\ atomicEntryBlock\ atomicEntryUpd\ var, n >, Flush_{mm}\ (var), n+1 >>,$
    $<< Flush_{mm}\ (var), n+1 >, < Readvar \rightarrow readVal, n+2 >>,$
    $<< Readvar \rightarrow readVal, n+2 >, < Write\ var \leftarrow (readVal \oplus updVal), n+3 >>,$
    $<< Write\ var \leftarrow (readVal \oplus updVal), n+3 >, < Flush_{mm}\ (var), n+4 >>,$
    $<< Flush_{mm}\ (var), n+4 >, < BlockSynch\ atomicExitBlock\ atomicExitUpd\ var, n+5 >>\}$
  // The flushes depend on all prior smOps that relate to var
  $\uplus \{< op_{prev}, < Flush_{mm}\ (var), n >> \mid RelatesBefore(op_{prev}, \{var\}, n, trace_{sub})\}$
  $\uplus \{< op_{prev}, < Flush_{mm}\ (var), n+2 >> \mid RelatesBefore(op_{prev}, \{var\}, n, trace_{sub})\}$
  // The $BlockSynchs$ depends on all prior $BlockSynch$ smOps
  $\uplus \{< op_{prev}, < BlockSynch\ atomicEntryBlock\ atomicEntryUpd\ var, n >> \mid$
    $BlockSynchBefore(op_{prev}, n, trace_{sub})\}$
  $\uplus \{< op_{prev}, < BlockSynch\ atomicExitBlock\ atomicExitUpd\ var, n+5 >> \mid$
    $BlockSynchBefore(op_{prev}, n, trace_{sub})\}$
  // all six smOps depend on the last read that was part of a while loop iteration test
  $\uplus \{< R^{while}_{prev}, < BlockSynch\ atomicEntryBlock\ atomicEntryUpd, n >> \mid R^{while}_{prev} = last\ while\ loop\ read\}$
  $\uplus \{< R^{while}_{prev}, < Flush_{mm}\ (var), n+1 >> \mid R^{while}_{prev} = last\ while\ loop\ read\}$
  $\uplus \{< R^{while}_{prev}, < Readvar \rightarrow readVal, n+2 >> \mid R^{while}_{prev} = last\ while\ loop\ read\}$
  $\uplus \{< R^{while}_{prev}, < Write\ var \leftarrow (readVal \oplus updVal), n+3 >> \mid R^{while}_{prev} = last\ while\ loop\ read\}$
  $\uplus \{< R^{while}_{prev}, < Flush_{mm}\ (var), n+4 >> \mid R^{while}_{prev} = last\ while\ loop\ read\}$
  $\uplus \{< R^{while}_{prev}, < BlockSynch\ atomicExitBlock\ atomicExitUpd, n+5 >> \mid R^{while}_{prev} = last\ while\ loop\ read\}$

$< n, app, trace_{sub}, \overrightarrow{DepO} > \Rightarrow < n+5, app', trace_{sub}, \overrightarrow{DepO}' >$

While Loop Iteration Step: $While(var = testVal)\ bodyList$

// The next operation in the source code is the while loop test condition
$app = (\textbf{While}(\textbf{var} = \textbf{testVal})\ \textbf{bodyList}) :: app'$

// The variable used in the test is an actual application variable
$var \in appVars$

// The *Read* smOp that makes up this appOpp appears in the sub-trace
$< Read\ var \rightarrow readVal, n > \in trace_{sub}$
// And the read returned a value $= testVal$
$readVal = testVal$
// Update $\overrightarrow{DepO}$ to contain new dependencies:
$\overrightarrow{DepO}' = \overrightarrow{DepO} \uplus$
// The read depends on all prior non-read smOps that relate to var
$\uplus \{< op^{var}_{prev}, < Read\ var \rightarrow readVal, n >> \mid$
$RelatesBefore(op^{var}_{prev}, \{var\} \wedge op^{var}_{prev}\ not\ a\ Read, n, trace_{sub})\}$
// And on the last read that was part of a while loop iteration test
$\uplus \{< R^{while}_{prev}, < Read\ var \rightarrow readVal, n >> \mid R^{while}_{prev} = last\ while\ loop\ read\}$

$< n, app, trace_{sub}, \overrightarrow{DepO} > \Rightarrow$
$< n+1, bodyList :: (While(var = testVal)\ bodyList) :: app', trace_{sub}, \overrightarrow{DepO}' >$

While Loop Termination Step: $While(var = testVal)\ bodyList$

// The next operation in the source code is the while loop test condition
$app = (\textbf{While}(\textbf{var} = \textbf{testVal})\ \textbf{bodyList}) :: app'$

// The variable used in the test is an actual application variable
$var \in appVars$

// The *Read* smOp that makes up this appOpp appears in the sub-trace
$< Read\ var \rightarrow readVal, n > \in trace_{sub}$
// The read returned a value $\neq testVal$
$readVal \neq testVal$
// Update $\overrightarrow{DepO}$ to contain new dependencies:
$\overrightarrow{DepO}' = \overrightarrow{DepO} \uplus$
// The read depend on all prior non-read smOps that relate to var
$\uplus \{< op^{var}_{prev}, < Read\ var \rightarrow readVal, n >> \mid$
$RelatesBefore(op^{var}_{prev}, \{var\} \wedge op^{var}_{prev}\ not\ a\ Read, n, trace_{sub})\}$
// And on the last read that was part of a while loop iteration test
$\uplus \{< R^{while}_{prev}, < Read\ var \rightarrow readVal, n >> \mid R^{while}_{prev} = last\ while\ loop\ read\}$

$< n, app, trace_{sub}, \overrightarrow{DepO} > \Rightarrow < n+1, app', trace_{sub}, \overrightarrow{DepO}' >$

Print Step: $Print\ var$

// The next operation in the source code is a print
$app = (\textbf{Print var}) :: app'$

// The variable being printed is an actual application variable
$var \in appVars$

// The *Read* smOp that makes up this appOpp appears in the sub-trace
$< Read\ var \rightarrow readVal, n > \in trace_{sub}$
// Update $\overrightarrow{DepO}$ to contain new dependencies:
$\overrightarrow{DepO}' = \overrightarrow{DepO} \uplus$
// The read depend on all prior non-read smOps that relate to var
$\uplus \{< op^{var}_{prev}, < Read\ var \rightarrow readVal, n >> \mid$
$RelatesBefore(op^{var}_{prev}, \{var\} \wedge op^{var}_{prev}\ not\ a\ Read, n, trace_{sub})\}$
// And on the last read that was part of a while loop iteration test
$\uplus \{< R^{while}_{prev}, < Read\ var \rightarrow readVal, n >> \mid R^{while}_{prev} = last\ while\ loop\ read\}$

$< n, app, trace_{sub}, \overrightarrow{DepO} > \Rightarrow < n+1, app', trace_{sub}, \overrightarrow{DepO}' >$

End Step: *End*

// The next operation in the source code is the *End* operation
$app = (\textbf{End}) :: app'$
 	
// All the smOps in the sub-trace have been processed already
$\forall < smOp, m > \ \in \ trace_{sub}, \ m \leq n$
 	
// No operations follow *End* in the source code
$app' = []$

$< n, app, trace_{sub}, \overrightarrow{DepO} > \ \Rightarrow \ < n+1, [], trace_{sub}, \overrightarrow{DepO} >$

7 Runtime Phase

The first pass verifies that the smOps from each thread's sub-trace could have come from the given application. The second pass, the runtime phase, verifies that the values returned by reads would occur with some OpenMP conformant interleaving of the smOp traces. It evaluates the traces from all the threads in parallel, interleaving operations from different threads, as diagrammed in Figure 8. The transition system below specifies this evaluation procedure.

During each transition we choose some thread and evaluate the next smOp from this thread's sub-trace. We then check that the value returned for any read could have been read under the OpenMP memory model. Conceptually, our runtime phase does not have a single shared memory. Instead, each write simply becomes available to reads on its own thread and other threads the moment it is evaluated. Overall, this phase determines the trace is valid if at least one interleaving of thread operations agrees with the trace, since the procedure is non-deterministic. As discussed in Section 7.5, we consider an interleaving of smOps to agree with the trace if:

- it verifies the values returned by all reads; and
- either all smOps have been evaluated or the remaining smOps correspond to a deadlock.

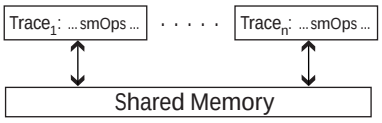

Fig. 8. Diagram of the runtime phase

7.1 Runtime State

The state of an application with r threads is:
$\sigma, \overrightarrow{FlshO}; < t_1 | subtrace_1, done_1, \overrightarrow{LclO_1} >, ..., < t_r | subtrace_r, done_r, \overrightarrow{LclO_r} >$
where:

- σ: The state of all synchronizations.
  - Contains one component for each type of synchronization in full model.
  - $\sigma.ActiveUpdates$: Set of pairs $< var, t_i >$, each corresponding to an atomic update of var currently being performed by thread t_i.
  - $\sigma.HeldLocks$: Set of pairs $< lockVar, t_i >$, each corresponding to a lock variable $lockVar$ currently being held by thread t_i. Initially $= \emptyset$.
  - $\sigma.HeldCrits$: Set of pairs $< critName, t_i >$, each corresponding to thread t_i executing inside of critical section $critName$. Initially $= \emptyset$.

  - $\sigma.OrderedPassed$: Set of $< loopID, i >$ pairs, each corresponding to a parallel for loop (uniquely identified by $loopID$) and the latest iteration i for which the ordered section has completed execution. Initially $= \emptyset$.
  - $\sigma.BarBlocked$: Mapping of threads to booleans that records whether each thread is currently blocked on a barrier. Initially, maps every thread to False.
- $\overrightarrow{FlshO}$: The flush order established so far; initially, the empty relationship.
- $subtrace_i$: The suffix of thread t_i's sub-trace with its smOps yet to be evaluated; initially t_i's full sub-trace.
- $done_i$: Set of smOps that have already been evaluated by thread t_i.
- $\overrightarrow{LclO_i}$: Thread t_i's local order established so far; initially, the empty relationship.

The partial orders $\overrightarrow{FlshO}$ and $\overrightarrow{LclO_i}$ are defined on the events that happen on different threads. $\overrightarrow{FlshO}$ applies to events on all threads. $\overrightarrow{LclO_i}$ applies to events on thread t_i. How these two orders relate events determines the values returned by reads.

$\overrightarrow{LclO_i}$ is the evaluation order of thread t_i in our runtime pass, the order in which it evaluates t_i's operations. If event E_1 is evaluated on thread t_i before event E_2 then we have $E_1\ \overrightarrow{LclO_i}\ E_2$. For any event E that happened on some thread t_i, "$\overrightarrow{LclO_i}\ \sqcup^i E$" is defined to be an order that is identical to $\overrightarrow{LclO_i}$, except that event E follows all events that have been completed on thread t_i.

$\overrightarrow{FlshO}$ is the global sequential flush order, defined by the relative times that different threads evaluate flushes. Let E and F be two events such that F is a flush of the form $Flush_{mm}\ varList$. These two rules relate E and F:

- If the *same* thread evaluates E and F and E is a $(Read\ var)$, $(Write\ var)$ or $< BlockSynch\ blockF\ updF\ var, m >$ and $var\ \in\ varList$ then if E was evaluated before F then $E\ \overrightarrow{FlshO}\ F$, otherwise $F\ \overrightarrow{FlshO}\ E$.
- If E is a flush of the form $Flush_{mm}\ varList2$ (on *any* thread) and $varList\ \cap\ varList2 \neq \emptyset$ then if E was evaluated before F then $E\ \overrightarrow{FlshO}\ F$, otherwise $F\ \overrightarrow{FlshO}\ E$.

The transitive closure of these rules defines $\overrightarrow{FlshO}$. For any smOp *op* that was evaluated on some thread t_i we define "$\overrightarrow{FlshO}\ \sqcup^j_{varList}\ op$" be an order that is identical to $\overrightarrow{FlshO}$, except that *op* follows *any* operation evaluated on t_j that relates to any variable in $varList$. $\mathbb{F}]^j_{varList}$ is a flush-specific variant of $\sqcup^j_{varList}$, where "$\overrightarrow{FlshO}\ \mathbb{F}]^j_{varList} op$" is defined to be an order that is identical to $\overrightarrow{FlshO}$, except that *op* follows *any flush* operation evaluated on t_j whose variable list overlaps with $varList$.

These orders are used in two key concepts: operation `races` and `eclipsing` operations. Two operations `race` if they are not related via $\overrightarrow{FlshO}$. A write W_{ecl} on thread t_i `eclipses` a write W on thread t_j from view by read R on thread t_k (all accessing the same variable) if W_{ecl} sits between W and R under the order $\overrightarrow{FlshO\ \uplus\ LclO_i\ \uplus\ LclO_k}$. Similarly, a read R_{ecl} on thread t_i `eclipses` a write W on thread t_j from view by read R on thread t_k (all accessing the same variable) if R_{ecl} sits between W and R under the order $\overrightarrow{FlshO\ \uplus\ LclO_i\ \uplus\ LclO_k}$ and R_{ecl} returns a value different from that written by W.

The notion of operation `races` is used to determine undefined behavior as a result of a lack of synchronization between writes and other operations. The notion of `eclipsing` operations is used to define the set of writes that are visible to a given read operation. Both notions are used to define the set of values that are available for reading by a given read.

7.2 Runtime Transitions

The runtime phase transition system contains one rule for each smOp. Each transition evaluates s_i, the first smOp in $subtrace_i$, provided that:

- No s'_i previously evaluated on thread t_i exists such that $s_i\ \overrightarrow{DepO}\ s'_i$
- The return value recorded in s_i is available for reading as defined below, if s_i is a read
- Its $blockF$ function evaluates to false and its $updF$ function would update the synchronization state σ to reflect s_i's evaluation, if s_i is a blocking synchronization operation

If these conditions are not satisfied for thread t_i, its next smOp will not be evaluated until they are.

For any s_i, its transition rule:

- removes s_i so $subtrace_i' = tail(subtrace_i)$ (recall that $s_i = head(subtrace_i)$);
- updates $\overrightarrow{FlshO}$ and $\overrightarrow{LclO_i}$ to include the ordering relationships between E_{s_i}, s_i's evaluation event, and those of all previously evaluated smOps, as discussed above;
- updates synchronization state to $\sigma' = updF(\sigma)$ if s_i is a $BlockSynch$ smOp.

Additional actions depend on the type of smOp, as detailed in the transitions in Section 7.4.

The runtime phase succeeds once $subtrace_i$ is empty on every thread t_i or there is a deadlock, as discussed in Section 7.5; otherwise the phase backtracks to examine other interleavings. If no interleavings succeed, the phase fails and the trace demonstrates non-conformance. This section addresses the safety properties of valid traces. Fairness is addressed in Section 7.5.

The values available for reading in $subtrace_i$ depend on the established $\overrightarrow{FlshO}$ and $\overrightarrow{LclO}$ orders and the writes that the transition system has previously evaluated. Specifically, let R be a read of variable var on thread t_i. Let $visibleWriteSet$ be the set of all un-eclipsed writes that precede R under $\overrightarrow{FlshO \uplus LclO_i}$ and let
$presentRemoteWriteSet$ be the set of writes that race R. Then a given value val is `available for reading` by R if:

- $presentRemoteWriteSet$ contains any writes (the writes race with RA, allowing it return any value); or
- $visibleWriteSet$ contains a write raced with some write in $visibleWriteSet$ (the race can leave the variable with an undefined value); or
- $visibleWriteSet$ contains a write that wrote val; or
- $visibleWriteSet$ is empty (R is not preceded by any writes to var and thus got its value from uninitialized memory).

In other words, val is available if it is the most recently written value to var, there were writes racing with R or if var is uninitialized or contains the result of racing writes (so R may return anything).

7.3 Formal Definitions

Definitions Used in Transitions:
// The order that results from appending smOp op to order $\overrightarrow{LclO_i}$.
$\overrightarrow{LclO_i} \sqcup^i op = \overrightarrow{LclO_i} \uplus \{< op', op > \mid op' \in done_i\}$

// The order that results from appending smOp op to order $\overrightarrow{FlshO}$,
// causally connecting it to all prior operations evaluated by thread
// t_j that refer to one or more variables in $varList$.
$\overrightarrow{FlshO} \sqcup^j_{varList} op = \overrightarrow{FlshO} \uplus$
$\{< op', op > \mid \exists m.$
// op' has been evaluated by thread t_j
$op' \in done_j \wedge$
(
// and op' is a flush
$(op' =< Flush\ flushVarList, m > \wedge$
// and one or more variables in $varList$ is in the flush's list
$(varList \cap flushVarList) \neq \emptyset$
$) \vee$
// or op' is a read or write
$((op' =< Read\ var \rightarrow readVal, m > \vee$
$op' =< Write\ var \leftarrow readVal, m > \vee$
$) \wedge$
// that refers to a variable in $varList$

$var \in varList$)
)
}

// The order that results from appending smOp op to order $\overrightarrow{FlshO}$,
// causally connecting it to flushes of one or more variables in $varList$
// on thread t_j.
$\overrightarrow{FlshO} \; \mathbb{E}^{j}_{varList} op = \overrightarrow{FlshO} \uplus$
$\{< op', op > \mid \exists m.$
// op' is a flush
$op' =< Flush \; flushVarList, m > \; \wedge$
// and op' has been evaluated by thread t_j
$op' \in done_j \; \wedge$
// and one or more variables in $varList$ is in the flush's list
$(varList \cap flushVarList) \neq \emptyset\}$

// Two events are racing under a given Flush Order if they are
// not related under it.
$Racing(op_1, op_2, \; \overrightarrow{FlshO} \;) = \neg(op_1 \; \overrightarrow{FlshO} \; op_2) \wedge \neg(op_2 \; \overrightarrow{FlshO} \; op_1)$

// Defines what it means for a given write W_{ecl} to `eclipse` the write
// W from the view of read R under a given ordering $\overrightarrow{Order}$.
$WriteEclipse(var, R, W, W_{ecl}, \overrightarrow{Order}) =$
$\wedge (W_{ecl} \; is \; a \; write \; to \; var)$
$\wedge W \; \overrightarrow{Ord} \; W_{ecl} \; \overrightarrow{Order} \; R$

// Defines what it means for a given read R_{ecl} to `eclipse` a write
// W from the view of read R under a given ordering $\overrightarrow{Order}$.
$ReadEclipse(var, R, W, R_{ecl}, \overrightarrow{Order}) =$
$\wedge (R_{ecl} \; is \; a \; read \; from \; var)$
$\wedge (R'_{ecl} s \; value \neq WA's \; value)$
$\wedge W \; \overrightarrow{Order} \; R_{ecl} \; \overrightarrow{Order} \; R$

// If R read of variable var on thread t_i then its $visibleWriteSet$ is the
// set of writes that precede the given event and were not eclipsed by other
// writes and reads to var under the given flush order and local orders.
$visibleWriteSet(R, var, t_i, \overrightarrow{FlshO}, \overrightarrow{LclO}) =$
$\{W \mid W \; is \; write \; to \; var \; on \; thread \; t_j \; \wedge$
// $\overrightarrow{activeO}$ is the active inter-thread order that will be used to
// determine which writes are visible to this read and which
// writes may be eclipsed by other reads and writes
$let \; \overrightarrow{activeO} = \overrightarrow{FlshO} \; \uplus \; \overrightarrow{LclO_i} \; \uplus \; \overrightarrow{LclO_j} \; in$
// $visibleWriteSet$ is the set of writes to
// var that:
// (i) Precede the read in $\overrightarrow{FlshO}$ or $\overrightarrow{LclO_i}$
$\wedge (W \; \overrightarrow{FlshO} \; RA) \vee (W \; \overrightarrow{LclO_i} \; RA)$
// (ii) And there are no other writes to var that eclipse W from
// R under $\overrightarrow{activeO}$
$\wedge \neg \exists W_{ecl}.$
$(W_{ecl} \; is \; a \; write \; to \; var) \wedge$
$\wedge WriteEclipse(var, R, W, W_{ecl}, \overrightarrow{activeO})$

// (iii) And there are no reads of var that eclipse W from R
// under $\overrightarrow{activeO}$
$\wedge \neg\exists R_{ecl}.$
$(R_{ecl}\ is\ a\ read\ of\ var) \wedge$
$\wedge\ ReadEclipse(var, R, W, R_{ecl}, \overrightarrow{activeO})$
}

// If R read of variable var on thread t_i then $presentRemoteWriteSet$ is the
// set of writes to var from another thread that could happen at the same
// time as the read according to the flush order.
$presentRemoteWriteSet(R, var, t_i, \overrightarrow{FlshO}) =$
$\{write\ or\ W\ to\ var\ |$
$(W\ is\ on\ thread\ j \neq i) \wedge Racing(W, R, \overrightarrow{FlshO})\}$

// Defines the set of values that are available for reading by read R of var
// that is evaluated on thread t_i, under the given orders $\overrightarrow{FlshO}$ and $\overrightarrow{LclO}$.
$availableForReading(R, var, t_i, \overrightarrow{FlshO}, \overrightarrow{LclO}) =$
$\{readVal|$
// The value $readVal$ could have been read if R is racing some write
// (in which case it may read any value)
$\exists W\ \in\ presentRemoteWriteSet(R, var, t_i, \overrightarrow{FlshO}).\ W\ is\ a\ write$
// Or some of the past writes that R could have read its value from
// were racing with each other
// (in which case the variable may contain value)
$\vee \exists W_1, W_2\ \in\ visibleWriteSet(R, var, t_i, \overrightarrow{FlshO}, \overrightarrow{LclO}).$
$W_1\ and\ W_2\ are\ writes \wedge Racing(W_1, W_2, \overrightarrow{FlshO})$
// Or $readVal$ is the value written by some past un-eclipsed write
$\vee \exists W\ \in\ visibleWriteSet(R, var, t_i, \overrightarrow{FlshO}, \overrightarrow{LclO}), m.$
$W =< Write\ var \leftarrow readVal, m >$
// Or the $visibleWriteSet$ is empty, meaning that R gets the
// variable's uninitialized value (which may be anything).
$\vee\ visibleWriteSet(R, var, t_i, \overrightarrow{FlshO}, \overrightarrow{LclO}) = \emptyset$
}

Definition of Valid Sequences:
$Trans_{runtime}$ = set of all runtime transitions

If $transition\ \in\ Trans_{runtime}$ is a specific transition then the application of that transition to thread t_i is denoted as: $transition_i$.

// The initial state of the runtime transition system for the given application and trace, running on r threads
$InitS_{runtime}(r, app, trace) = \sigma_{init}, \emptyset; < t_1|trace_1, \emptyset, \emptyset >, ..., < t_r|trace_r, \emptyset, \emptyset >$
// where σ_{init} is:
$\sigma_{init}.HeldLocks = \emptyset$
$\sigma_{init}.HeldCrits = \emptyset$
$\sigma_{init}.OrderedPassed = \emptyset$
$\sigma_{init}.BarBlocked = (\lambda thread.\ False)$

// Definition of a legal sequence of of runtime transitions for a given
// application and trace, running on r threads
$LegalRuntimeSeq(seq, app, trace, r) =$
// The sequence begins with the initial state
$seq[0] = InitS_{runtime}(r, app, trace)$ // And every pair of adjacent runtime states is related via some

valid

// compiler transition

$\forall n \in [0, seq.length).\ \exists transition \in Trans_{runtime}.$

$transition(seq[n] \Rightarrow seq[n+1])$

// Definition of a trace being verified by the runtime phase (ignoring Fairness concerns)

$ValidTraceRuntimeSafety(app, trace, r) =$

// For every thread there exists some legal sequence of runtime states

// (i.e. an interleaving of smOps) that validates the trace's runtime behavior

$\forall t_i.\ \exists seq_i.\ LegalRuntimeSeq(seq_i, app, trace, r)$

7.4 Formal Transition System

Write Step

// The next operation in thread t_i's sub-trace is a *BlockSynch*
$subtrace_i =< Write\ var \leftarrow val, n_i >:: subtrace'_i$

// Thread t_i evaluates the write operation and transitions to the corresponding
// new state if the conditions below are satisfied.

// $\overrightarrow{FlshO}'$ is $\overrightarrow{FlshO}$ but updated to include the new write, with the write following
// all the flush operations relating to *var* that have been completed on this thread
// and included *var* in their *varList*
$\overrightarrow{FlshO}' = \overrightarrow{FlshO} \sqcup^i_{\{var\}} \ < Write\ var \leftarrow val, n_i >$

// $\overrightarrow{LclO_i}'$ is $\overrightarrow{LclO_i}$ but updated to include the new read, with the
// read following all events that have been completed on thread t_i.
$\overrightarrow{LclO_i}' = \overrightarrow{LclO_i} \sqcup^i < Write\ var \leftarrow val, n_i >$

// The write operation has not been evaluated after some other operation
// that depends on the write via $\overrightarrow{DepO}$.
$\forall smOp_{prev} \in done_i.\ \neg(< Write\ var \leftarrow val, n_i > \ \overrightarrow{DepO}\ smOp_{prev})$

// Thread t_i has a write operation as the next thing in its trace
$\sigma, \overrightarrow{FlshO}; ..., < t_i | subtrace_i, done_i, \overrightarrow{LclO_i} >, ..., < t_j | subtrace_j, done_j, \overrightarrow{LclO_j} >, ... \ \Rightarrow$
$\sigma', \overrightarrow{FlshO}'; ..., < t_i | subtrace'_i, done_i \cup head(subtrace_i), \overrightarrow{LclO_i}' >, ...,$
$< t_j | subtrace_j, done_j, \overrightarrow{LclO_j} >, ...$

Read Step

// The next operation in thread t_i's sub-trace is a *BlockSynch*
$subtrace_i =< Read\ var \rightarrow readVal, n_i >:: subtrace'_i$

// Thread t_i evaluates the read operation and transitions to the corresponding
// new state if the conditions below are satisfied.

// $\overrightarrow{FlshO}'$ is $\overrightarrow{FlshO}$ but updated to include the new read, with the read
// following all the flush operations relating to *var* that have been completed
// on this thread and included *var* in their *varList*
$\overrightarrow{FlshO}' = \overrightarrow{FlshO} \sqcup^i_{\{var\}} \ < Read\ var \rightarrow readVal, n_i >$

// $\overrightarrow{LclO_i}'$ is $\overrightarrow{LclO_i}$ but updated to include the new read, with the
// read following all events that have been completed on thread t_i.
$\overrightarrow{LclO_i}' = \overrightarrow{LclO_i} \sqcup^i < Read\ var \rightarrow readVal, n_i >$

// The value returned by this read, was actually available for reading at this
// point in time
$readVal \in availableForReading(< Read\ var \rightarrow readVal, n_i >, var, t_i, \overrightarrow{FlshO}', \overrightarrow{LclO}')$

// The read operation has not been evaluated after some other operation
// that depends on the read via $\overrightarrow{DepO}$.
$\forall smOp_{prev} \in done_i.\ \neg(< Read\ var \rightarrow readVal, n_i > \ \overrightarrow{DepO}\ smOp_{prev})$

// Thread t_i has a read operation as the next thing in its trace
$\sigma, \overrightarrow{FlshO}; ..., < t_i | subtrace_i, done_i, \overrightarrow{LclO_i} >, ..., < t_j | subtrace_j, done_j, \overrightarrow{LclO_j} >, ... \ \Rightarrow$
$\sigma', \overrightarrow{FlshO}'; ..., < t_i | subtrace'_i, done_i \cup head(trace_i), \overrightarrow{LclO_i}' >, ...,$
$< t_j | subtrace_j, done_j, \overrightarrow{LclO_j} >, ...$

Flush Step

// The next operation in thread t_i's sub-trace is a $Flush_{mm}$
$subtrace_i =< Flush_{mm}\ varList, n_i >:: subtrace'_i$

// Thread t_i evaluates the flush operation and transitions to the corresponding
// new state if the conditions below are satisfied.

// $\overrightarrow{FlshO}'$ is $\overrightarrow{FlshO}$ but updated to include the new flush, with the flush following
$\overrightarrow{FlshO}' = \overrightarrow{FlshO} \uplus$
$\overrightarrow{FlshO}' = \overrightarrow{FlshO}$
  // all smOps that have been evaluated on this thread and access a variable $\in\ varList$.
  $\sqcup^i_{varList} < Flush_{mm}\ varList, n_i >$
  // all flushes that have been completed on any thread this thread and have
  // variable lists that overlap $varList$.
  $\boxed{\text{F}}^j_{varList} < Flush_{mm}\ varList, n_i >\ \forall\ threads\ t_j$
// $\overrightarrow{LclO_i}'$ is $\overrightarrow{LclO_i}$ but updated to include the new flush, with the
// flush following all events that have been completed on thread t_i.
$\overrightarrow{LclO_i}' = \overrightarrow{LclO_i} \sqcup^i < Flush_{mm}, n_i >$

// The flush operation has not been evaluated after some other
// operation that depends on the flush via $\overrightarrow{DepO}$.
$\forall smOp_{prev}\ \in\ done_i.\ \neg(Flush\ \overrightarrow{DepO}\ smOp_{prev})$

// Thread t_i has a flush operation as the next thing in its trace
$\sigma, \overrightarrow{FlshO}; ..., < t_i | subtrace_i, done_i, \overrightarrow{LclO_i} >, ..., < t_j | trace_j, done_j, \overrightarrow{LclO_j} >, ...\ \Rightarrow$
  $\sigma', \overrightarrow{FlshO}'; ..., < t_i | subtrace'_i, done_i \cup head(trace_i), \overrightarrow{LclO_i}' >, ...,$
  $< t_j | trace_j, done_j, \overrightarrow{LclO_j} >, ...$

Blocking Synchronization Step

// The next operation in thread t_i's sub-trace is a $BlockSynch_{mm}$
$subtrace_i =< BlockSynch_{mm}\ blockF\ updF, n_i >:: subtrace'_i$

// Thread t_i evaluates the blocking synchronization operation and
// transitions to the corresponding new state if the conditions below
// are satisfied.

// Thread t_i is not currently blocked and may proceed with its execution
$blockF(\sigma) = False$
// The synchronization state is transformed to reflect the fact that thread
// t_i is unblocked
$\sigma' = updF(\sigma)$

// $\overrightarrow{FlshO}'$ is $\overrightarrow{FlshO}$ but updated to include the new synchronization operation, with the
// synchronization following all flush operations that have been completed on thread t_i.
$\overrightarrow{FlshO}' = \overrightarrow{FlshO}\ \sqcup^i_{allVars}\ < BlockSynch_{mm}\ blockF\ updF, n_i >$
// $\overrightarrow{LclO_i}'$ is $\overrightarrow{LclO_i}$ but updated to include the synchronization operation,
// with the synchronization following all events that have been completed on thread t_i.
$\overrightarrow{LclO_i}' = \overrightarrow{LclO_i} \sqcup^i < BlockSynch_{mm}\ blockF\ updF, n_i >$

// The synchronization operation has not been evaluated after some other
// operation that depends on it via $\overrightarrow{DepO}$.
$\forall smOp_{prev}\ \in\ done_i.\ \neg(< BlockSynch_{mm}\ blockF\ updF, n_i >\ \overrightarrow{DepO}\ smOp_{prev})$

$\sigma, \overrightarrow{FlshO}; ..., < t_i | subtrace_i, done_i, \overrightarrow{LclO_i} >, ..., < t_j | n_j, trace_j, done_j, \overrightarrow{LclO_j} >, ...\ \Rightarrow$
  $\sigma', \overrightarrow{FlshO}'; ..., < t_i | subtrace'_i, done_i \cup head(trace_i), \overrightarrow{LclO_i}' >, ...,$
  $< t_j | n_j, trace_j, done_j, \overrightarrow{LclO_j} >, ...$

Non-Blocking Synchronization Step

// The next operation in thread t_i's sub-trace is a $NonBlockSynch_{mm}$
$subtrace_i =< NonBlockSynch_{mm}\ blockF\ updF\ \rightarrow\ successFlag, n_i >:: subtrace'_i$

// Thread t_i evaluates the non-blocking synchronization operation and
// transitions to the corresponding new state if the conditions below
// are satisfied.

// If this is a successful synchronization, $NonBlockSynch_{mm}$ acts like
// $BlockSynch_{mm}$, only being able to proceed if $blockF$ returns True.
$successFlag = True\ \Rightarrow$
    // Thread t_i is not currently blocked and may proceed with its execution
    $\wedge\ blockF(\sigma) = False$
    // The synchronization state is transformed to reflect the fact that thread
    // t_i is unblocked
    $\wedge\ \sigma' = updF(\sigma)$
// If this is an unsuccessful synchronization, $NonBlockSynch_{mm}$ acts as a noop
// and neither $blockF$, nor $updF$ need to be evaluated.

// $\overrightarrow{FlshO}'$ is $\overrightarrow{FlshO}$ but updated to include the new synchronization operation, with the
// synchronization following all flush operations that have been completed on thread t_i.
$\overrightarrow{FlshO}' = \overrightarrow{FlshO}\ \sqcup^i_{allVars}\ < NonBlockSynch_{mm}\ blockF\ updF\ \rightarrow\ successFlag, n_i >$
// $\overrightarrow{LclO_i}'$ is $\overrightarrow{LclO_i}$ but updated to include the synchronization operation,
// with the synchronization following all events that have been completed on thread t_i.
$\overrightarrow{LclO_i}' = \overrightarrow{LclO_i} \sqcup^i < NonBlockSynch_{mm}\ blockF\ updF\ \rightarrow\ successFlag, n_i >$

// The synchronization operation has not been evaluated after some other
// operation that depends on it via $\overrightarrow{DepO}$.
$\forall smOp_{prev}\ \in\ done_i.\ \neg(< NonBlockSynch_{mm}\ blockF\ updF\ \rightarrow\ successFlag, n_i >\ \overrightarrow{DepO}\ smOp_{prev})$

$\sigma, \overrightarrow{FlshO}; ..., < t_i | subtrace_i, done_i, \overrightarrow{LclO_i} >, ..., < t_j | n_j, trace_j, done_j, \overrightarrow{LclO_j} >, ...\ \Rightarrow$
    $\sigma', \overrightarrow{FlshO}'; ..., < t_i | subtrace'_i, done_i \cup head(trace_i), \overrightarrow{LclO_i}' >, ...,$
        $< t_j | n_j, trace_j, done_j, \overrightarrow{LclO_j} >, ...$

7.5 Fairness and Deadlocks

The transition rules verify that a trace conforms with the OpenMP memory model if an interleaving of operations exists that agrees with the outcomes of the trace's smOps. Interleavings in which some smOp of some thread never executes are not sufficient since the phase will not validate that thread's sub-trace. Thus, our model has a basic fairness guarantee on valid traces that we now make explicit.

A trace is `Fair` if an interleaving of thread transitions exists such that no thread's current smOp is `enabled` for evaluation an infinite number of times without being evaluated (this is known as Strong Fairness [9]). In particular, *BlockSynch* is only enabled in states where its *blockF* returns false, reads are enabled when their values are `available for reading` and writes and flushes are always enabled for execution. For finite traces this fairness condition guarantees that every smOp on every thread will eventually be evaluated unless there is a deadlock or the ordering of smOps on a thread's sub-trace violates the application's dependence order. For infinite traces it ensures no thread may be enabled for unblocking an infinite number of times without actually unblocking. In particular, if a thread is waiting to acquire a lock that periodically becomes available, it will eventually acquire it.

However, OpenMP does not guarantee deadlock freedom. A poorly written OpenMP program can contain a deadlock. Thus, our fairness guarantee also allows applications that deadlock. If the application reaches a point where every thread's next smOp is a *BlockSynch* whose *blockF* returns true, then the proposed

interleaving deadlocks. Ordinarily, our transition system would reject the interleaving since each thread's last smOp (the *BlockSynch*) would not be validated against the trace. In order to allow (poorly written) applications that may deadlock, we explicitly accept deadlocked interleavings if every thread's last smOp is a *BlockSynch* for which *blockF* returns true.

A situation similar to deadlocks can occur when the sub-traces of one or more threads violate the dependence order established during the compiler phase. The problem is that the next smOp on such threads will never be evaluated since its evaluation would follow the evaluation of an smOp that should have preceded it according to the dependence order. Such traces are illegal and are rejected by the above model.

7.6 Formal Fairness

// Defines what it means for an smOp to be `enabled` for evaluation
$EnabledOp(\sigma, op) =$
    // A *BlockSynch* smOp is enabled if it is unblocked
    $(op =< BlockSynch\ blockF\ updF, n_i > \wedge blockF(\sigma) = False) \vee$
    // all other smOps are always enabled
    $op \neq< BlockSynch\ blockF\ updF, n_i >$

// Defines what it means for a sequence of runtime states to be Fair
$FairSeq(seq) =$
    // *seq* is Fair if for any time step m the next smOp on any thread
    // t_i is enabled for execution
    $\forall t_i.$
        // If operations on t_i are enabled for execution infinitely often
        $\forall m < seq.length.\ \exists n < \infty.$
            $EnabledOp(seq[m+n].\sigma, head(seq[m+n].t_i.trace_i))$
        $\Rightarrow$
        // Then they are actually evaluated infinitely often
        $\forall m < seq.length.\ \exists n < \infty,\ transition\ \in\ Trans_{runtime}.$
            $transition_i(seq[m+n] \Rightarrow seq[m+n+1])$

// Definition of a trace being verified by the runtime phase, including
// the Fairness guarantee
$ValidTraceRuntime(app, trace, r) =$
    // There exists some sequence of runtime states s.t.
    $\exists seq.$
        // The the sequence satisfies all the safety properties that relate it
        // to the application and its trace
        $ValidTraceRuntimeSafety(app, trace, r) \wedge$
        // And the sequence satisfies Fairness
        $FairSeq(seq)$

8 Examples

In the examples below we use the following shorthand:

- $var_A = const$ corresponds to $var_A = var_{const} + var_{zero}$ where var_{const} and var_{zero} are variables that are initialized to *const* and 0 and never modified.
- *Barrier* corresponds to the smOps that make up the *Barrier* appOp:
  $Flush_{mm}\ allVars$,

$BlockSynch\ barEntrBlock\ barEntrUpd$,
$BlockSynch\ barExitBlock\ barExitUpd$ and
$Flush_{mm}\ allVars$.

8.1 Uninitialized Read

Thread 0	Thread 1
Flush	var=1
print var	Flush

Fig. 9. Uninitialized read example

Thread 0	Thread 1
var=0	Barrier
Barrier	var=1
Flush	Flush
print var	

Fig. 10. Initialized read example

Figure 9 contains an example code where the read on thread 0 may return any value. The reason is that if the read executes before the write, its $visibleWriteSet$ will be empty. Therefore, the read may return any value since the value would come from uninitialized memory. In order to avoid such uninitialized reads we can transform this program into the one in Figure 10.

In the modified program the barrier ensures that thread 0's read must follow some write to var, meaning that its $visibleWriteSet$ cannot be empty. In future examples, whenever we make a statement about variables' initial value, we mean that the example's operations were preceded by a barrier, which was itself preceded by writes that initialized those variables. Equivalently, we could assume that the initialization occurs prior to the first parallel construct; we construct our examples with existing threads for notational simplicity.

8.2 Example A.2

The example in Figure 11 comes directly from example A.2 from the OpenMP 2.5 specification [2], converted from the original C/C++ and Fortran into the simplified language. Figure 12 shows a typical operation interleaving of this code (All other interleavings produce the same results).

Initially, $x = 2$

Thread 0	Thread 1
x=5	print(x)
Barrier	Barrier
print(x)	print(x)

Fig. 11. Example A.2

Thread 0	Thread 1
$Write\ flag \leftarrow 2$	
Barrier	Barrier
$Write\ x \leftarrow 5$	
	$Read\ x \rightarrow$??? (print)
Barrier	Barrier
$Read\ x \rightarrow 5$ (print x)	
	$Read\ x \rightarrow 5$ (print)

Fig. 12. Example A.2 sample execution

This interleaving features three reads. The first read is evaluated on thread 1 before the barriers. As such, in any possible interleaving it must race the write to x on thread 0. Since the write is in the first read's $presentRemoteWriteSet$, the read may return any value, regardless of x's initial value. The two other reads are in a different situation. The barriers force them to follow the write in any interleaving. Because of the $Flush_{mm}$ inside each barrier, both reads follow the write on thread 0 in $\overrightarrow{FlshO}$. As such, the write is in their $visibleWriteSet$. With no other available writes, this means that both reads must return 5, the value written by thread 0. The formalism is consistent with the explanation of example A.2 [2].

8.3 Faulty Spinlock

Initially, $flag = 0$

Thread 0	Thread 1
flag=1	Flush
Flush	while(flag=0){
	print(flag)
	Flush
	}
	print(flag)

Fig. 13. Example of a faulty spinlock

Thread 0	Thread 1
$Write\ flag \leftarrow 0$	
Barrier	Barrier
$Write\ flag \leftarrow 1$	
	$Flush_{mm}\ allVars$
	$Read\ flag \rightarrow$??? (while)
	$Read\ flag \rightarrow$??? (print)
	...
$Flush_{mm}\ allVars$	
	$Flush_{mm}\ allVars$
	$Read\ flag \rightarrow 1$ (while)
	$Read\ flag \rightarrow 1$ (print)

Fig. 14. Sample faulty spinlock interleaving

Figure 13 shows a basic spinlock. At first it appears that this program will print a finite sequence of 0's, followed by a 1. However, despite the abundance of flushes there is a race between the write on thread 0 and the reads on thread 1. The smOp interleaving that reveals this race is shown in Figure 14.

The problem here is that the reads on thread 1 may happen before the flush on thread 0. Thus, the values read by these reads are unspecified, meaning that the values printed may be garbage. Fortunately, our fairness assumption guarantees the flush on thread 0 will eventually be evaluated. The following iteration of the while loop on thread 1 will execute a flush. Since this flush will follow thread 0's flush, thread 0's write will now precede subsequent reads on thread 1 under $\overrightarrow{FlshO \uplus LclO_1}$. This in turn causes them to read 1, terminating the while loop.

While this seems to be a contrived example, consider the case of a shared memory implementation where 64-bit writes are broken up into multiple 16-bit messages and the write on thread 0 actually writes some large 64-bit value. In this case the reads on thread 1 may read $flag$ while it is only partially updated with only some of the 16-bit messages, causing the prints to output garbage. Despite the erroneous output, it is still true that the while loop on thread 1 will eventually terminate, making this the only way to write a working spinlock in OpenMP: use a loop that waits until a variable is written to but doesn't care about the actual value written. Since *Write-Read* races result in undefined read output, other spinlock variants will not work.

Initially, $flag = 1$

Thread 0	Thread 1
Atomic flag+=1	Flush
	while(flag=0){
	print(flag)
	Flush
	}
	print(flag)

Fig. 15. Correct Spinlock

Thread 0	Thread 1
$Write\ flag \leftarrow 0$	
Barrier	Barrier
$BlockSynch\ atomicEntryBlock$	
$atomicEntryUpd\ flag$	
$Flush_{mm}\ \{flag\}$	
	$Flush_{mm}\ allVars$
	$Read\ flag \rightarrow 0$ (while)
	$Read\ flag \rightarrow 0$ (print)
$Read\ flag \rightarrow 0$	
$Write\ flag \leftarrow 1$	
	$Flush_{mm}\ allVars$
$Flush_{mm}\ \{flag\}$	
	$Read\ flag \rightarrow$??? (while)
	$Read\ flag \rightarrow$??? (print)
	$Flush_{mm}\ allVars$
	$Read\ flag \rightarrow 1$ (while)
	$Read\ flag \rightarrow 1$ (print)
$BlockSynch\ atomicExitBlock$	
$atomicExitUpd\ flag$	

Fig. 16. Sample faulty spinlock interleaving

Indeed, consider the example code in Figure 15, which is identical to Figure 13, except that the write is replaced with an atomic update. While atomic updates are atomic relative to other atomic updates due to their flushes and synchronization, they do not look atomic to regular reads that may be racing with them. Figure 16 shows what happens.

An atomic update consists of a read and a write surrounded by flushes of *var*, which are themselves surrounded by *BlockSynchs* that ensure that no two atomic updates may execute at the same time. The first iteration of thread 1's wait loop executes at the beginning of thread 0's atomic update, after its initial flush but before its read and write. As such, the two loop reads both return 0, since they are only preceded by the initialization write. The next iteration of the while loop happens after thread 0's write. However, because thread 1's flush happens before thread 0's flush, thread 1's reads are not properly ordered relative to thread 0's write. As such, their return values are undefined. The last loop iteration happens after thread 0's atomic update has performed its final flush (though, not the final *BlockSynch*). Because thread 1's flush now properly follows thread 0's flush, the subsequent reads on thread 1 return 1.

8.4 Correct Use of Atomic Updates

The example in Figure 17 shows an example of how atomic updates are to be used correctly. In this code threads 0 and 1 execute atomic updates while thread 2 tries to read their intermediate results. All threads then execute a barrier and print the variable.

Initially, $x = 0$

Thread 0	Thread 1	Thread 2
Atomic x+=1	Atomic x+=1	Flush
Barrier	Barrier	print(x)
print(x)	print(x)	Flush
		print(x)
		Barrier
		print(x)

Fig. 17. Example of the correct use of atomic updates

Figure 18 shows a sample execution of this code. Thread 0 starts first, by executing its atomic update. It reads 0 and writes 1, performing appropriate flushes before allowing thread 1 to begin its atomic update. Thread 1's atomic update does the same, reading 1 and writing 2, because appropriate synchronization and flushing were performed relative to thread 0's write. Meanwhile thread 2 executes its two read operations. Because there is no synchronization relative to the writes on threads 0 and 1, the values returned by the reads are undefined. After all threads have performed their barriers (and thus, performed both synchronization and flushes) their subsequent reads are guaranteed to be properly ordered relative to the preceding writes. As such, when each thread tries to read the variable, the write on thread 1 is the most recent uneclipsed write for all of them, meaning that each thread reads 2 as the value of x.

8.5 Multi-thread Writer Race

The example in Figure 19 shows the effect of a race between writes. Suppose that the above application has smOp interleaving as in Figure 20. Before threads 0 and 1 do their flushes, the reads on thread 2 are racing with the writes on threads 0 and 1 under the order $\overrightarrow{FlshO}$. This is still true after thread 0 performs its flush since the reads on thread 2 are still racing with thread 1's write. The problem persists even after thread 1's flush. At this point both writes are in the past of all subsequent reads on thread 2 according to $\overrightarrow{FlshO \uplus LclO_0 \uplus LclO_2}$ and $\overrightarrow{FlshO \uplus LclO_1 \uplus LclO_2}$. However, the two writes are not related to each other under $\overrightarrow{FlshO}$, meaning that they race. Thus, the third read on thread 2 may also return an unspecified value.

Thread 0	Thread 1	Thread 2
$Write\ x \leftarrow 0$		
Barrier	Barrier	Barrier
$BlockSynch\ atomicEntryBlock$ $atomicEntryUpd\ x$		
$Flush_{mm}\ \{x\}$		
$Read\ x \rightarrow 0$		
		$Flush_{mm}\ allVars$
		$Readx \rightarrow$??? (print)
$Write\ x \leftarrow 1$		
$Flush_{mm}\ \{x\}$		
$BlockSynch\ atomicExitBlock$ $atomicExitUpd\ flag$		
	$BlockSynch\ atomicEntryBlock$ $atomicEntryUpd\ x$	
	$Flush_{mm}\ \{x\}$	
	$Read\ x\ 1$	
	$Write\ x \leftarrow 2$	
		$Flush_{mm}\ allVars$
		$Readx \rightarrow$??? (print)
	$Flush_{mm}\ \{x\}$	
	$BlockSynch\ atomicExitBlock$ $atomicExitUpd\ flag$	
Barrier	Barrier	Barrier
$Readx \rightarrow 2$ (print)		
	$Readx \rightarrow 2$ (print)	
		$Readx \rightarrow 2$ (print)

Fig. 18. Atomic updates sample execution

Initially, $flag = 0$

Thread 0	Thread 1	Thread 2
flag=1	flag=42	Flush
Flush	Flush	print(flag)
		Flush
		print(flag)
		Flush
		print(flag)

Fig. 19. Multi-thread writer race example

Thread 0	Thread 1	Thread 2
$Write\ flag \leftarrow 0$		
Barrier	Barrier	Barrier
$Write\ flag \leftarrow 1$		
	$Write\ flag \leftarrow 42$	
		$Flush_{mm}\ allVars$
		$Read\ flag \rightarrow$??? (print)
$Flush_{mm}\ allVars$		
		$Flush_{mm}\ allVars$
		$Read\ flag \rightarrow$??? (print)
	$Flush_{mm}\ allVars$	
		$Flush_{mm}\ allVars$
		$Read\ flag \rightarrow$??? (print)

Fig. 20. Sample multi-thread writer race interleaving

In reality, this example can happen in the aforementioned implementation where 64-bit writes are broken up into 16-bit messages and no filtering is done to tell which 16-bit message comes from which 64-bit write. Since the writes on threads 0 and 1 are unrelated by any synchronization, their individual messages may arrive in memory in arbitrary order, causing the resulting stored value to contain pieces from both writes.

8.6 Writes from Same Thread

The example in Figure 21 again highlights the importance of enforcing a proper order on the reads and writes on different threads. In this case, we have two writes executed on one thread and a read executed on another (with appropriate flushes). If the read is properly ordered to be executed after the writes, it is guaranteed to see them in their program order: it will return the value of the last write. In the absence of proper ordering, anything can happen.

Initially, $flag = 0$

Thread 0	Thread 1
flag=1	Flush
flag=2	print(flag)
Flush	

Fig. 21. Example of writes from the same thread

Thread 0	Thread 1
$Write\ flag \leftarrow 0$	
Barrier	Barrier
$Write\ flag \leftarrow 1$ [*]	
$Write\ flag \leftarrow 2$ [**]	
$Flush_{mm}\ allVars$	
	$Flush_{mm}\ allVars$
	$Read\ flag \rightarrow 2$ (print)

Fig. 22. Properly ordered interleaving

Thread 0	Thread 1
$Write\ flag \leftarrow 0$	
Barrier	Barrier
	$Flush_{mm}\ allVars$
$Write\ flag \leftarrow 1$ [*]	
$Write\ flag \leftarrow 2$ [**]	
$Flush_{mm}\ allVars$	
	$Read\ flag \rightarrow$??? (print)

Fig. 23. Unordered interleaving

Figure 22 shows a properly ordered trace. Thread 0 goes first, issues both writes and performs a flush. Note that since both writes were to $flag$, they were related via $\overrightarrow{DepO}$ and had to be evaluated in that order. Furthermore, when the read on thread 1 was evaluated, both writes precede it according to order $\overrightarrow{FlshO \uplus LclO_0 \uplus LclO_1}$ and write [**] follows write [*] under to the same ordering. As a result, the write [*] is eclipsed by write [**] under the definition of $WriteEclipse(flag, R, Write\ [*], W\ [**],$ $\overrightarrow{FlshO \uplus LclO_0 \uplus LclO_1})$. Thus, the read only has write [**] in its past, no writes in its present and therefore returns 2.

Figure 23 shows what happens when the read is not properly ordered relative to the writes. In this case both writes are in the read's present since they are not ordered relative to the read via $\overrightarrow{FlshO}$. Thus, the read may return any value. Indeed, any later read is also free to return any value until thread 1 calls a $Flush_{mm}$, placing the two writes on thread 0 into the past under order $\overrightarrow{FlshO \uplus LclO_0 \uplus LclO_1})$.

8.7 Local Reads Eclipse Writes

Figure 24 presents an example code where a read on one thread can eclipse prior writes on another thread from all subsequent reads on the same thread. The smOp interleaving in Figure 25 shows how this can

Initially, $flag = 0$

Thread 0	Thread 1
flag=0	
Barrier	Barrier
flag=1	flag=2
Flush	Flush
	print(flag)
	print(flag)

Fig. 24. Example of local reads eclipsing writes

Thread 0	Thread 1
$Write\ flag \leftarrow 0$	
Barrier	Barrier
	$Write\ flag \leftarrow 2$ [@]
$Write\ flag \leftarrow 1$ [@@]	
$Flush_{mm}\ allVars$	
	$Flush_{mm}\ allVars$
	$Read\ flag \rightarrow 1$ (print) [*]
	$Read\ flag \rightarrow 1$ (print) [**]

Fig. 25. Sample interleaving showing eclipsing behavior

happen.
In this trace threads 0 and 1 perform a writes to flag, followed by flushes. When thread 1 performs read [*], it has two writes that are in its *visibleWriteSet* and can choose to read either of their values. It chooses 1. When thread 2 evaluates read [**] it finds that read [*] eclipses write [@] via the definition $ReadEclipse(flag, Read\ [**], Write\ [@], Read\ [*], \overrightarrow{FlshO\ \uplus\ LclO_1\ \uplus\ LclO_1})$ because it reads 1 rather than 2 and appears between write [@] and read [**] under ordering $\overrightarrow{FlshO\ \uplus\ LclO_1\ \uplus\ LclO_1}$. However, write [@@] is not eclipsed by read [*] because it writes value 1, the same as read [*].

Alternately, note that if the non-deterministic choice at read [*] was to read 2 rather than 1, then the reverse eclipse would occur.

8.8 Remote Reads Eclipse Writes

Initially, $flag = 0$

Thread 0	Thread 1
flag=1	flag=2
Flush	Flush
print(flag)	print(flag)
Flush	

Fig. 26. Example of remote reads eclipsing writes

The example in Figure 26 shows how a read on one thread can eclipse prior writes on the same thread from subsequent reads on another thread. The following smOp interleaving from Figure 27 gives us an idea of how this can happen.

In this trace threads 0 and 1 both perform writes to flag, followed by flushes. Thread 0 then performs read [*], which has two writes in its *visibleWriteSet*. As such, it can read any value, in this case 42. When thread 2 performs read [**], both writes in its past. However, read [*] eclipses both writes under order $\overrightarrow{FlshO\ \uplus\ LclO_0\ \uplus\ LclO_1}$ since it reads a different value from what either write write. Thus, in this trace read[**] may only read 42.

Thread 0	Thread 1
$Write\ flag \leftarrow 0$	
Barrier	Barrier
$Write\ flag \leftarrow 1$ [@]	
	$Write\ flag \leftarrow 2$ [@@]
$Flush_{mm}\ allVars$	
Read flag $\rightarrow$ 42 (print) [*]	
$Flush_{mm}\ allVars$	
	$Flush_{mm}\ allVars$
	$Read\ flag \rightarrow 42$ (print) [**]

Fig. 27. Sample interleaving showing eclipsing behavior

8.9 Lock Usage

Initially, $varZero = 0$, $varOne = 1$, $counter = 0$

Thread 0	Thread 1
while(varZero=0){	while(varZero=0){
Lock lockVar	Lock lockVar
print(counter)	print(counter)
counter = counter+varOne	counter = counter+varOne
print(counter)	print(counter)
Unlock lockVar	Unlock lockVar
}	}

Fig. 28. Lock usage example

The example in Figure 28 shows how locks can be used to enforce mutual exclusion. Any execution of the above program must print out the infinite sequence 1, 2, 3, ... The smOp interleaving in Figure 29 shows why.

In this example thread 0 begins its execution by entering its while loop and locking $lockVar$. The *Lock* appOp is made up of a *BlockSynch* smOp, surrounded by flushes of all variables. *BlockSynch lockBlock lockUpd* blocks if $< lockVar, t_0 > \ \in \ \sigma.HeldLocks$. Since initially $\sigma.HeldLocks = \emptyset$, this means that thread 0 does not block and continues executing, changing $\sigma.HeldLocks$ to $\{< lockVar, t_0 >\}$. Meanwhile thread 1 also begins its execution and while it can enter the while loop and call the first flush of the *Lock* appOp, it's *BlockSynch lockBlock lockUpd* cannot continue because $< lockVar, t_0 > \ \in \ \sigma.HeldLocks$. Thus, it blocks until this changes.

After acquiring the lock, thread 0 increments *counter*. *counter*'s value must be read in as 0 because the $presentRemoteWriteSet$ for the read of counter is empty (due to the mutual exclusion provided by the locks) and the $visibleWriteSet$ contains only the initialization write. Thus, the value of counter is written out as 1 and then printed out as 1. Finally, thread 0 evaluated the $Unlock\ lockVar$ appOp. This consists of a *BlockSynch unlockBlock unlockUpd*, surrounded by flushes of all variables. *unlockBlock* never makes the thread block and *unlockUpd* removes $< lockVar, t_0 >$ from $\sigma.HeldLocks$.

Since $\sigma.HeldLocks$ is now empty, thread 1 is able to proceed. It adds $< lockVar, t_1 >$ to $\sigma.HeldLocks$ and proceeds to increment *counter*. This time the only value that can be read for *counter* is 1 because $presentRemoteWriteSet$ is empty and the only un-eclipsed write in $visibleWriteSet$ is the write from *counter*'s previous increment on thread 0. (the initialization write is eclipsed by thread 0's increment write under $\overrightarrow{FlshO \ \uplus \ LclO_0 \ \uplus \ LclO_1}$) Thus, the following write saves *counter*'s value as 2, which is the value printed by $print(counter)$. Finally, $Unlock\ lockVar$ performs the flushes and removes $< lockVar, t_1 >$ from $\sigma.HeldLocks$.

This pattern repeats itself an infinite number of times. The important thing is that the mutual exclusion provided by the *Lock* appOps, together with their internal flushes ensures that the updates performed in one

Thread 0	Thread 1
$Write\ varZero \leftarrow 0$	
$Write\ varOne \leftarrow 0$	
$Write\ counter \leftarrow 0$	
Barrier	Barrier
$Read\ varZero \rightarrow 0$ (while)	
$Flush_{mm}\ allVars$ (Lock)	
$BlockSynch\ lockBlock\ lockUpd$ (Lock)	
$Flush_{mm}\ allVars$ (Lock)	
$Read\ counter \rightarrow 0$ (counter = ...)	
$Read\ varOne \rightarrow 1$ (counter = ...)	
$Write\ counter \leftarrow 1$ (counter = ...)	$Read\ varZero \rightarrow 0$ (from while(varZero=0))
Read counter $\rightarrow$ 1 (Print)	
$Flush_{mm}\ allVars$ (Unlock)	$Flush_{mm}\ allVars$ (Lock)
$BlockSynch\ unlockBlock\ unlockUpd$ (Unlock)	
$Flush_{mm}\ allVars$ (Unlock)	$BlockSynch\ lockBlock\ lockUpd$ (Lock)
	$Flush_{mm}\ allVars$ (Lock)
	$Read\ counter \rightarrow 1$ (counter = ...)
	$Read\ varOne \rightarrow 1$ (counter = ...)
	$Write\ counter \leftarrow 2$ (counter = ...)
	$Read\ counter \rightarrow 2$ (Print)
	$Flush_{mm}\ allVars$ (Unlock)
	$BlockSynch\ unlockBlock\ unlockUpd$ (Unlock)
	$Flush_{mm}\ allVars$ (Unlock)
...	...

Fig. 29. Lock usage sample interleaving

locked code region are seen in another locked code region and the locked code regions execute in a sequential fashion.

9 Conclusion

The OpenMP 2.5 specification includes a section that details the OpenMP memory model [2]. This section significantly improves previous specifications – the previous C/C++ specifications did not address the issue directly at all. Instead, users and implementers had to synthesize a model as best they could from several disparate sections. However, the memory model is still described in informal prose, which lacks precision by definition.

This paper presents a formal OpenMP memory model, derived from the model in the current specification. We tried to faithfully adhere to that prose description. However, as we have discussed, it has several ambiguities, which we resolve in our formal model by relying on our understanding of the intent of the language committee. Our operational model supports the verification of the conformance of OpenMP implementations. It consists of two phases: a compiler phase that extracts the constituent operations of the application and a runtime phase that verifies that a compliant execution could produce the values that appear in the trace. We have applied this model to several examples. Overall, our work demonstrates the need for the OpenMP community to adopt further refinements of the OpenMP memory model. Ideally those changes will lead to a formal model in later OpenMP specifications.

References

1. Posix 1003.1c-1995.
2. OpenMP Architecture Review Board. OpenMP application program interface, version 2.5.

3. Greg Bronevetsky and Bronis de Supinski. Fully formal specification of the OpenMP memory model. Cornell Computer Science, 2005. In Preparation.
4. William W. Collier. *Reasoning About Parallel Architectures*, 1992.
5. Scheurich C. Dubois, M. and F Briggs. Memory access buffering in multiprocessors. In *In Proceedings of the 13th Annual International Symposium on Computer Architecture (ISCA)*, pages 434–442, 1986.
6. J.R. Goodman. Cache consistency and sequential consistency. Technical Report 61, SCI Committee, 1989.
7. Jay Hoeflinger and Bronis de Supinski. The openmp memory model. In *International Workshop on OpenMP (IWOMP)*, 2005.
8. William Pugh Jeremy Manson and Sarita V. Adve. The java memory model. In *Symposium on Principles of Programming Languages (POPL 2005)*.
9. Leslie Lamport. Fairness and hyperfairness. Technical report.
10. John Matthews Serdar Tasiran Mark Tuttle Rajeev Joshi, Leslie Lamport and Yuan Yu. Checking cache-coherence protocols with tla+. *Formal Methods in System Design*, 22(2):125–131, 2003.
11. Alan Robinson and Andrei Voronkov eds. *Handbook of Automated Reasoning Volume*, 2000.

This work was performed under the auspices of the U.S. Department of Energy by University of California, Lawrence Livermore National Laboratory under Contract W-7405-Eng-48.