LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

UCRL-CONF-221452

Formal Specification of the OpenMP Memory Model

G. Bronevetsky, B. R. de Supinski

May 18, 2006

Second International Workshop on OpenMP (IWOMP 2006)
Reims, France
June 12, 2006 through June 15, 2006

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 [1] 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 [1]) 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.
This paper focuses on the formal verification of OpenMP programs through a proposed formal memory model that is derived from the existing prose model [1]. Our formalization provides a two-step process to verify whether an observed OpenMP execution is conformant. 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 being increasingly 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 most easily and effectively programmed in a multi-threaded shared memory style.

OpenMP [1] 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 where a program or protocol is formalized 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 [9]. While formal verification is generally too complex to apply to real-world applications, it is feasible for the basic algorithms on which real applications are based.

Existing work on formally verifying shared memory algorithms [8] 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 [7] [3], none exists for OpenMP. Instead, the official description of OpenMP's memory model (section 1.4 of version 2.5 of the OpenMP specification [1]) 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 [6], this clarification is also informal.

This paper focuses on verification of OpenMP programs through a proposed formal memory model that we derived from the existing prose model [1]. Our formalization provides a two-step process to verify if an observed OpenMP execution is conformant. 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 present several examples that demonstrate a need for further refinement of the OpenMP memory model however it is specified. 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 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 [4]. 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 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 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 accesses 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. Thus, we can 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 in addition to reads, writes and flushes. These 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, we had several questions as we formulated our formal memory model based on it. 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 present 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? The memory model is definitely intended to allow them but only supports them with this sentence: "The flush operation restricts reordering of memory operations that an implementation might otherwise do." We read this to mean that the memory model imposes no other reordering restrictions. This would mean that compilers may reorder operations that access the same shared variable. In particular, they can reorder not only reads but also writes. In general, the compiler can reorder any accesses not separated by a `flush`, including conflicting accesses to the same variable, provided that it preserves the application's sequential semantics.

```
if(threadNum==0) {
    Barrier
    A=20;
    Barrier
} else {
    A=5;
    Barrier
    Barrier
    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, 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, can produce unexpected results. 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 [5].

In Section 2, we presented 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, what value 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 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 otherwise: "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 [1] 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, 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 section. In the very least, a forward reference is needed.

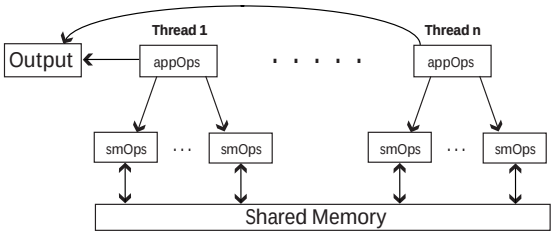
3.4 Captured Writes

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." We find this ambiguous and believe others will also. 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`? We believe that to be the intent but the actual wording could refer to writes on other threads that have been read by the given thread. Our point 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

The following sections describe the OpenMP memory model in formal, mathematical language. This specification takes as input an application and a trace that shows how this application executed on top of some implementation of OpenMP (a trace is a tuple of lists of executed shared memory operations, one list for each thread, with the operations stored in the order in which they were executed on that thread, along with their results, if any). It then uses a set of rules to judge if the application could have generated the trace and if a valid interleaving of thread operations exists under the OpenMP memory model that results in the values read in the trace.

Our OpenMP formalization is an operational model (outlined on the right). It defines a system state and valid transition rules for modifying the 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 val), 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 executed by its respective thread and any values they returned (e.g., the entry ($Read$ $var \mapsto val$) corresponds to a read of variable var that returned the value val). Traces do not specify the interleaving of smOps from different threads.

We break our operational model into two sub-models, the Compiler Phase and the Runtime Phase, so that we can reason independently about different aspects of the memory model. 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 that 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. Whenever an appOp uses values from shared memory (e.g., the value returned by a read), it looks them up in the sub-trace. The trace corresponds to the application's source code if the compiler phase independently verifies this for each sub-trace.

The runtime phase determines if the smOps in the individual threads' sub-traces correspond to each other. More specifically, it evaluates 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. Therefore, its state consists of:

- the writes, atomic updates and flushes that each thread performed (one list per thread);
- a partial order that relates those 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 for 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 or atomic update 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 operations to the above partial order. Since the runtime phase is non-deterministic, the trace is self-consistent if the exists some interleaving of the different threads' smOps such that all reads and atomic updates 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, respectively. Due to lack of space, we do not cover the

full mathematical details of the formalism, which are available elsewhere [2]. Instead, we express them in a more verbal style here.

5 Language Specification

5.1 Application Operations

Our 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 and locks. Section number references refer to the OpenMP 2.5 specification [1]. The while loop primitive makes the application language Turing-complete in its use of shared memory operations. As mentioned, these operations are sufficient for our examples; the complete language covers the remaining OpenMP synchronization operations such as barriers and ordered sections [2].

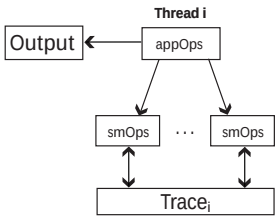
$var_A = var_B \otimes var_C$ • Represents any local computation performed by the application. • $\otimes$ is a Turing-complete binary operation that does not use shared memory. • var_A, var_B and var_C are shared variables. • Corresponds to (*Read* var_B), (*Read* var_C) and (*Write* var_A *val*) smOps.	*Lock lockVar* *Unlock lockVar* • Model the `omp_set_lock` and `omp_unset_lock` function calls [section 3.3]. • *lockVar* is a shared variable only accessed via *Lock* and *Unlock* operations. • Correspond to a *BlockSynch* smOp surrounded by ($Flush_{mm}$ *allVars*) smOps (*Lock* and *Unlock* correspond to different *BlockSynch* smOps)
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*.	*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.
Atomic var $\oplus =$ *updVal* • Models the atomic update construct [section 2.7.4]. • $\oplus$ may be one of the following operations: +, $*$, −, /, &, ^, |, <<, or >> (++ and - - are modeled via +=1 and -=1). • *var* is a shared variable. • *updVal* is a constant. • Corresponds to an $Atomic_{mm}$ smOp surrounded by ($Flush_{mm}$ (*var*)) smOps.	*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

We use a very simple shared memory operation language that is sufficient for the functionality needs of the higher-level appOps. The smOps include reads, writes, atomic updates, flushes and blocking synchronizations (from which higher-level synchronizations are built) and are detailed in Figure 1.

6 Compiler Phase

The compiler phase, diagrammed here, 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 and atomic writes are also looked up in the trace. This phase also defines a dependence order $\overrightarrow{DepO}$ on each thread's smOps, which



$Write\ var\ val$: writes val to variable var.	$BlockSynch\ blockF\ updF$: generic blocking synchronization operation.
• var is a shared variable. • val is a constant.	• Used to implement synchronization semantics of higher-level operations such as locks 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 $updF$ vary with each high-level synchronization construct. • The compiler phase (Section 6) defines $blockF$ and $updF$. • The runtime phase (Section 7), where synchronization state is defined, applies $blockF$ and $updF$.
$Read\ var \mapsto val$: read of variable var returns val.	
• var is a shared variable. • val is a constant.	
$Atomic_{mm}\ var \oplus =\ updVal \mapsto finalVal$: atomically updates variable var to $finalVal$.	
• var is a shared variable. • $updVal$ is a constant. • Reads current value, val, of var. • Computes $finalVal = val \oplus updVal$. • Writes $finalVal$ to var. • Actions are atomic: unsychronized atomic updates do not make the value of var indeterminate. • Does not have any flush semantics (unlike the *Atomic* appOp). • $\oplus$ may be: $+$, $*$, $-$, $/$, &, $\hat{}$, $\mid$, $<<$, or $>>$.	
$Flush_{mm}\ varList$: flushes this thread's temporary view of 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.	

Fig. 1. Types of shared memory operations

the evaluation in the runtime phase must not violate. The remainder of this section defines the state and transition function of the compiler phase.

This phase's operational model is applied to the sub-trace corresponding to each thread. During each transition it evaluates the next appOp of the *app* list 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 $\overrightarrow{DepO}$ is preserved after this compiler pass 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 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 null relationship.

6.2 Compiler Transitions

The valid state transitions are shown in Figure 2. 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. The transition used depends on the associated value of the loop variable, as described following 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.

Computation

Current State:
$[n, (\mathbf{var_A} = \mathbf{var_B} \otimes \mathbf{var_C}) :: app, trace_{sub}, \overrightarrow{DepO}]$
Next State: $[n+3, app, trace_{sub}, \overrightarrow{DepO}']$
and the following are true:
- $< Read\ var_B \mapsto val_B, n > \ \in\ trace_{sub}$
- $< Read\ var_C \mapsto val_C, n+1 > \ \in\ trace_{sub}$
- $< Write\ var_A\ (val_B \otimes val_C), n+2 > \ \in\ trace_{sub}$
- $\overrightarrow{DepO}'$ extends $\overrightarrow{DepO}$ as follows:
  - The write depends on the reads.
  - The read from var_B, the read from var_C and the write to var_A, depend on the most recently evaluated writes or atomic updates to var_A, var_B or var_C, respectively (if any).
  - All three smOps depend on the most recent read that was part of a while loop iteration test (i.e., they depend on control flow).

While Loop

Current State:
$[n, (\mathbf{While(var = testVal)\ bodyList}) :: app, trace_{sub}, \overrightarrow{DepO}]$
Next State if $readVal = testVal$:
$[n+1, bodyList :: (While(var = testVal)\ bodyList) :: app, trace_{sub}, \overrightarrow{DepO}']$
Next State if $readVal \neq testVal$:
$[n+1, app, trace_{sub}, \overrightarrow{DepO}']$
and the following are true:
- $< Read\ var \mapsto readVal, n > \ \in\ trace_{sub}$
- $\overrightarrow{DepO}'$ extends $\overrightarrow{DepO}$ as follows:
  - The *Read* of *var* depends on the most recently evaluated write or atomic update of *var* (if any).
  - The read depends on the most recent read that was part of a while loop iteration test.

Atomic Update

Current State:
$[n, (\mathbf{Atomic\ var} \oplus = \ \mathbf{updVal}) :: app, trace_{sub}, \overrightarrow{DepO}]$
Next State: $[n+3, app, trace_{sub}, \overrightarrow{DepO}']$
and the following are true:
- $< Flush_{mm}\ (var), n > \ \in\ trace_{sub}$
- $< (Atomic_{mm}\ var \oplus = \ updVal \mapsto finalVal), n+1 > \ \in\ trace_{sub}$
- $< Flush_{mm}\ (var), n+2 > \ \in\ trace_{sub}$
- $\overrightarrow{DepO}'$ extends $\overrightarrow{DepO}$ as follows:
  - The smOps are ordered to place the atomic update between the two flushes.
  - The $Atomic_{mm}$ smOp depends on the most recently evaluated write to or atomic update of *var*.
  - The $Flush_{mm}$ smOps depend on all prior writes to or atomic updates of *var*.
  - All three smOps depend on the most recent read that was part of a while loop iteration test.

Print

Current State:
$[n, (\mathbf{Print\ var}) :: app, trace_{sub}, \overrightarrow{DepO}]$
Next State: $[n+1, app, trace_{sub}, \overrightarrow{DepO}']$
and the following are true:
- $< Read\ var \mapsto readVal, n > \ \in\ trace_{sub}$
- $\overrightarrow{DepO}'$ extends $\overrightarrow{DepO}$ as follows:
  - The read of *var* depends on the most recently evaluated write or atomic update of *var* (if any).
  - The read depends on the most recent read that was part of a while loop iteration test.

Lock Acquire

Current State:
$[n, (\mathbf{Lock\ lockVar}) :: app, trace_{sub}, \overrightarrow{DepO}]$
Next State: $[n+3, app, trace_{sub}, \overrightarrow{DepO}']$
and the following are true:
- $< Flush_{mm}\ allVars, n > \ \in\ trace_{sub}$
- $< BlockSynch\ lockBlock\ lockUpd, n+1 > \ \in\ trace_{sub}$
- $< Flush_{mm}\ allVars, n+2 > \ \in\ trace_{sub}$
- $\overrightarrow{DepO}'$ extends $\overrightarrow{DepO}$ as follows:
  - The smOps are ordered to place the lock acquisition between the two flushes.
  - The $Flush_{mm}$ smOps depend on all prior writes to or atomic updates of any variable and the lock acquire ($BlockSynch$) depends on the most recently evaluated acquire or release of $lockVar$.
  - All three smOps depend on the most recent read that was part of a while loop iteration test.
- *lockBlock* is a function that returns $True$ (blocked) if $lockVar$ is currently held by some thread and $False$ otherwise.
- *lockUpd* takes the current runtime state and returns one where $lockVar$ is recorded as being held.

Lock Release

Current State:
$[n, (\mathbf{Unock\ lockVar}) :: app, trace_{sub}, \overrightarrow{DepO}]$
Next State: $[n+3, app, trace_{sub}, \overrightarrow{DepO}']$
and the following are true:
- $< Flush_{mm}\ allVars, n > \ \in\ trace_{sub}$
- $< BlockSynch\ unlockBlock\ unlockUpd, n+1 > \ \in\ trace_{sub}$
- $< Flush_{mm}\ allVars, n+2 > \ \in\ trace_{sub}$
- $\overrightarrow{DepO}'$ extends $\overrightarrow{DepO}$ as follows:
  - The smOps are ordered to place the lock release between the two flushes.
  - The $Flush_{mm}$ smOps depend on all prior writes to or atomic updates of any variable and the lock release ($BlockSynch$) depends on the most recently evaluated acquire or release of $lockVar$.
  - All three smOps depend on the most recent read that was part of a while loop iteration test.
- *unlockBlock* always returns False (not blocked)
- *unlockUpd* updates the current runtime state s.t. $lockVar$ is recorded as being not held.

Flush

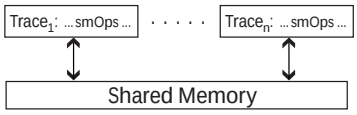
Current State:
$[n, (\mathbf{Flush\ varList}) :: app, trace_{sub}, \overrightarrow{DepO}]$
Next State: $[n+1, app, trace_{sub}, \overrightarrow{DepO}']$
and the following are true:
- $< Flush_{mm}\ varList, n > \ \in\ trace_{sub}$
- $\overrightarrow{DepO}'$ extends $\overrightarrow{DepO}$ as follows:
  - The $Flush_{mm}$ smOp depends on all previously evaluated writes to or atomic updates of variables in $varList$.
  - The $Flush_{mm}$ depends on the most recent read that was part of a while loop iteration test.

End

Current State:
$[n, (\mathbf{End}) :: app, trace_{sub}, \overrightarrow{DepO}]$
Next State: $[n+1, [], trace_{sub}, \overrightarrow{DepO}']$
and $\forall < smOp, m > \ \in\ trace_{sub},\ m \leq n$
(the sub-trace has no more smOps).

Fig. 2. Valid application state transitions

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 and atomic updates 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 here. 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* or *Atomic* update could have been read under the OpenMP memory model. Conceptually, our runtime phase does not have a single shared memory. Instead, each write or atomic update 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 7.3, we consider an interleaving of smOps to agree with the trace if:

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

7.1 Runtime State

The state of an application with r threads is:

$$\sigma, \overrightarrow{FlshO}; < t_1 | subtrace_1, \overrightarrow{LclO_1} >, ...$$
$$..., < t_r | subtrace_r, \overrightarrow{LclO_r} >$$

where:

- σ: The state of all synchronizations.
  - Contains one component for each type of synchronization in full model.
  - $\sigma.HeldLocks$: lock component (only component in abbreviated model)
    - Set of pairs $< lockVar, t_i >$, corresponding to lock variables $lockVar$ currently held by thread t_i.
    - Initially $= \emptyset$.
- $\overrightarrow{FlshO}$: The flush order established so far; initially, the null 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.
- $\overrightarrow{LclO_i}$: Thread t_i's local order established so far; initially, the null 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 program order of thread t_i in our runtime pass, the order in which it evaluates t_is 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, we define "$\overrightarrow{LclO_i}\ \sqcup^i E$" 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 $(Atomic_m m\ var \oplus =\ updVal)$ 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 event E that happened on some thread t_i we define "$\overrightarrow{FlshO}\ \sqcup^j_{var} E$" to be an order that is identical to $\overrightarrow{FlshO}$, except that event E follows any flush operation evaluated on t_j that has var in its variable list. (note that t_i may or may not be the same as t_j)

We use these orders in two key concepts: operation `races` and `eclipsing` operations. Two operations `race` if they are not related via $\overrightarrow{FlshO}$. A write or atomic update WA_{ecl} on thread t_i `eclipses` a write or atomic update WA on thread t_j from view by read R on thread t_k (all accessing the same variable) if WA_{ecl} sits between WA and R under the order $\overrightarrow{FlshO\ \cup\ LclO_i\ \cup\ LclO_k}$. Similarly, a read R_{ecl} on thread t_i `eclipses` a write or atomic update WA on thread t_j from view by read R on thread t_k (all accessing the

same variable) if R_{ecl} sits between WA and R under the order $\overrightarrow{FlshO \cup LclO_i \cup LclO_k}$ and R_{ecl} returns a value different from that written by WA.

7.2 Transition System

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 in $subtrace_i$ is available for reading as defined below, if s_i is a read or an atomic update;
- 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. The phase succeeds once $subtrace_i$ is empty on every thread t_i or there is a deadlock, as discussed in Section 7.3; otherwise the phase backtracks to examine other interleavings. If no interleavings succeed, the phase fails and the trace demonstrates non-conformance.

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

- $presentRemoteWriteSet$ contains any writes; or
- $presentRemoteWriteSet$ contains an atomic update the final value of which is *val*; or
- $pastWriteSet$ contains a pair of writes that race each other; or
- $pastWriteSet$ contains a write that wrote *val* or an atomic update the final value of which is *val*; or
- $pastWriteSet$ is empty (i.e. RA 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* or if *var* is uninitialized or racing writes exist to it (so RA can return anything).

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 Figure 3.

7.3 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. An interleavings in which some smOp of some thread never executes is 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. In particular, *BlockSynch* is only enabled in states where its *blockF* returns false, reads and atomic updates 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

Blocking synchronization

Current State:

$\sigma, \overrightarrow{Flsh O}; ..., < t_i | <$ **BlockSynch blockF updF**$, m >::$
$subtrace_i, \overrightarrow{LclO_i} >, ...$

Next State:

$\sigma', \overrightarrow{Flsh O}; ..., < t_i | subtrace_i, \overrightarrow{LclO_i}' >, ...$

and the following are true:

- The function $blockF(\sigma)$ returns $False$, meaning that this thread does not need to block.
- $\sigma' = updF(\sigma)$, meaning that that synchronization state is transformed to reflect the fact that thread t_i is unblocked.
- $\overrightarrow{Flsh O}' = \overrightarrow{Flsh O} \sqcup^i_{var} E_{s_i}$ for all variables var.
- $\overrightarrow{LclO_i}' = \overrightarrow{LclO_i} \sqcup^i E_{s_i}$.

Atomic Update

Current State:

$\sigma, \overrightarrow{Flsh O}; ...,$
$< t_i | <$ **Atomic$_{\mathbf{mm}}$ var** $\oplus =$ **updVal** $\mapsto$ **finalVal**$, m >::$
$subtrace_i, \overrightarrow{LclO_i} >, ...$

Next State:

$\sigma', \overrightarrow{Flsh O}; ..., < t_i | subtrace_i, \overrightarrow{LclO_i}' >, ...$

and the following are true:

- $\overrightarrow{Flsh O}' = \overrightarrow{Flsh O} \sqcup^i_{var} E_{s_i}$.
- $\overrightarrow{LclO_i}' = \overrightarrow{LclO_i} \sqcup^i E_{s_i}$.

Read

Current State:

$\sigma, \overrightarrow{Flsh O}; ..., < t_i | <$ **Read var** $\mapsto$ **readVal**$, m >::$
$subtrace_i, \overrightarrow{LclO_i} >, ...$

Next State:

$\sigma', \overrightarrow{Flsh O}; ..., < t_i | subtrace_i, \overrightarrow{LclO_i}' >, ...$

and the following are true:

- The value $readValue$ is `available for reading`.
- $\overrightarrow{Flsh O}' = \overrightarrow{Flsh O} \sqcup^i_{var} E_{s_i}$.
- $\overrightarrow{LclO_i}' = \overrightarrow{LclO_i} \sqcup^i E_{s_i}$.

Write

Current State:

$\sigma, \overrightarrow{Flsh O}; ...,$
$< t_i | <$ **Write var val**$, m >::$
$subtrace_i, \overrightarrow{LclO_i} >, ...$

Next State:

$\sigma', \overrightarrow{Flsh O}; ..., < t_i | subtrace_i, \overrightarrow{LclO_i}' >, ...$

and the following are true:

- $\overrightarrow{Flsh O}' = \overrightarrow{Flsh O} \sqcup^i_{var} E_{s_i}$.
- $\overrightarrow{LclO_i}' = \overrightarrow{LclO_i} \sqcup^i E_{s_i}$.

Flush

Current State:

$\sigma, \overrightarrow{Flsh O}; ...,$
$< t_i | <$ **Flush$_{\mathbf{mm}}$ varList**$, m >::$
$subtrace_i, \overrightarrow{LclO_i} >, ...$

Next State:

$\sigma', \overrightarrow{Flsh O}'; ..., < t_i | subtrace_i, \overrightarrow{LclO_i}' >, ...$

and the following are true:

- $\overrightarrow{Flsh O}' = \overrightarrow{Flsh O} \sqcup^j_{var} E_{s_i}$ for all variables var and threads t_j.
- $\overrightarrow{LclO_i}' = \overrightarrow{LclO_i} \sqcup^i E_{s_i}$.

Fig. 3. Valid shared memory state transitions

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.

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 a barrier synchronization (not explicitly defined due to lack of space) and a $Flush_{mm}$ of all variables.

8.1 Uninitialized Read

Figure 4 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 *pastWriteSet* 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 5.

Thread 0	Thread 1
Flush	var=1
print var	Flush

Fig. 4. Uninitialized read example

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

Fig. 5. Initialized read example

In the modified program the barrier ensures that thread 0's read must follow some write to *var*, meaning that its $pastWriteSet$ 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 6 comes directly from example A.2 from the OpenMP 2.5 specification [1], converted from the original C/C++ and Fortran into our simplified language. Figure 7 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. 6. Example A.2

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

Fig. 7. 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 $pastWriteSet$. With no other available writes, this means that both reads must return 5, the value written by thread 0. Our formalism is consistent with the explanation of example A.2 [1].

8.3 Faulty Spinlock

Initially, $flag = 0$

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

Fig. 8. Example of a faulty spinlock

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

Fig. 9. Sample faulty spinlock interleaving

Initially, $flag = 1$

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

Fig. 10. Correct Spinlock

Figure 8 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 9.

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. Another iteration of the while loop on thread 1 will produce a flush call, which will cause thread 0's write to 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, suppose that we have 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. Indeed, the only way to prevent this situation is to ensure that the write to the flag is atomic, something that only the `atomic` construct can provide.

Given this new knowledge we can augment the program above to use an atomic update, as shown in Figure 10. In this case the above interleaving produces the expected behavior since even when the reads on thread 1 race with the atomic update on thread 0 (i.e. the atomic update is in their $presentRemoteWriteSet$), they do not get garbage values but rather either 0 or 1. (atomic update appOps contain their own $Flush_{mm}$ smOps)

8.4 Flush-free Spinlock

The example in Figure 11 is the same as the one above except that the flushes have been removed. This program must either print a sequence of zero of more 0's, followed by a 1 or an infinite sequence of 0's. To understand why this is, lets examine the smOp interleaving shown in Figure 12.

Initially, $flag = 0$

Thread 0	Thread 1

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

Fig. 11. Flush-free spinlock example

Thread 0	Thread 1
$Write\ flag$ 0 [*]	
Barrier	Barrier
	$Read\ flag \mapsto 0$ (while)
	$Read\ flag \mapsto 0$ (print)
	...
$Flush_{mm}$ $(flag)$	
$Atomic_{mm}$ $flag+ = 1 \mapsto 1$	
$Flush_{mm}$ $(flag)$	
	...
	$Read\ flag \mapsto 0$ (while)
	$Read\ flag \mapsto 0$ (print)
	...
	$Read\ flag \mapsto 1$ (print) [**]
	$Read\ flag \mapsto 1$ (while)
	$Read\ flag \mapsto 1$ (print)

Fig. 12. Sample flush-free spinlock interleaving

Before thread 0 executes the atomic update, the fact that reads on thread 1 have empty $presentRemoteWriteSets$ and $pastWriteSets$ that contain only the initialization write [*], causes them to return 0. When thread 0's atomic update does occur, thread 1 may not update its temporary view - ever. The atomic update is in the $presentRemoteWriteSet$ of its reads. Thus, the value may never be observed by thread 0, which can iterate its loop forever, printing out 0's. In the trace above, the view is eventually updated and some read [**] returns 1. Therefore, all subsequent reads of $flag$ on thread 1 must also read 1 because read [**] eclipses write [*] under order $\overrightarrow{FlshO \cup LclO_0 \cup LclO_1}$.

This example portrays an important lesson. Although fairness is an important condition and critical for avoiding infinite loops, it does not prevent them. Programs without appropriate flushes may still loop infinitely because a thread's temporary view may not be updated.

8.5 Multi-thread Writer Race

The example in Figure 13 shows the effect of a race between writes. Suppose that the above application has smOp interleaving as in Figure 14. 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 $\overrightarrow{FlshO \cup LclO_2}$. 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 $\overrightarrow{FlshO \cup LclO_2}$. However, the two writes are not related to each other under $\overrightarrow{FlshO \cup LclO_2}$, meaning that they race. This means that the third read on thread 2 may also return an unspecified value.

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.

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. 13. Multi-thread writer race example

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

Fig. 14. Sample multi-thread writer race interleaving

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 15 shows how writes on one thread that were placed in a given order by the program's source code will be seen to occur in this order by any reads on other threads that have ordered themselves correctly relative to the writes (via flushes). However, 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. 15. Example of a writes from the same thread

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

Fig. 16. Properly ordered interleaving

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

Fig. 17. Uordered interleaving

Figure 16 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 $\overrightarrow{FlshO\ \cup\ LclO_1\ \cup\ LclO_2}$ and write [**] follows write [*] under to the same ordering. As a result, the write [*] is eclipsed by write [**] under the definition of $WriteEclipse(flag, R, Write\ [*], W\ [**],$ $\overrightarrow{FlshO\ \cup\ LclO_1\ \cup\ LclO_2})$. Thus, the read only has write [**] in its past, no writes in its present and therefore returns 2.

Figure 17 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 $\overrightarrow{FlshO\ \cup\ LclO_0\ \cup\ LclO_1})$.

8.7 Atomic Updates Racing with Reads

Figure 18 shows a code example where atomic updates to a given variable may not be seen in a linear order to a reader thread that has not performed the appropriate flushes. This behavior is shown in Figure 19. In this trace the reads on thread 1 are preceded by the initialization write on thread 0 and two atomic updates on thread 1. Thus, the first read [*] has the initialization write in its $pastWriteSet$ and the two atomic updates in its $presentRemoteWriteSet$. Therefore, the read is free to return any of the three available values: 0, 1 or 2. In this trace it returns 2.

Now examine the other reads. Although they do follow read [*], the absence of flushes on thread 1 means that under the ordering $\overrightarrow{FlshO\ \cup\ LclO_0\ \cup\ LclO_1}$ read [*] does not eclipse any of the writes or atomic updates on thread 0. As such, their $pastWriteSets$ and $presentRemoteWriteSets$ are identical to those of read [*] and so they are free to return any of the same values: 0, 1 or 2.

Initially, $flag = 0$

Thread 0	Thread 1
Atomic flag+=1	print flag
Atomic flag+=2	print flag
	print flag

Fig. 18. Atomic values racing with reads example

Thread 0	Thread 1
$Write\ flag\ 0$	
Barrier	Barrier
$Flush_{mm}\ (flag)$	
$Atomic_{mm}\ flag\ +=\ 1) \mapsto 1$	
$Flush_{mm}\ (flag)$	
$Flush_{mm}\ (flag)$	
$Atomic_{mm}\ flag\ +=\ 1) \mapsto 2$	
$Flush_{mm}\ (flag)$	
	$Read\ flag \mapsto 2$ (print() [*]
	$Read\ flag \mapsto 1$ (print)
	$Read\ flag \mapsto 0$ (print)

Fig. 19. Sample interleaving for the Atomic Updates Racing with Reads example

9 Conclusion

The OpenMP 2.5 specification includes a section that details the OpenMP memory model [1]. 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. OpenMP Architecture Review Board. OpenMP application program interface, version 2.5.
2. Greg Bronevetsky and Bronis de Supinski. Fully formal specification of the OpenMP memory model. Cornell Computer Science, 2005. In Preparation.
3. William W. Collier. *Reasoning About Parallel Architectures*, 1992.
4. 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.
5. J.R. Goodman. Cache consistency and sequential consistency. Technical Report 61, SCI Committee, 1989.
6. Jay Hoeflinger and Bronis de Supinski. The openmp memory model. In *International Workshop on OpenMP (IWOMP)*, 2005.
7. William Pugh Jeremy Manson and Sarita V. Adve. The java memory model. In *Symposium on Principles of Programming Languages (POPL 2005)*.
8. 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.
9. 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.