University of Delaware
Department of Electrical and Computer Engineering
Computer Architecture and Parallel Systems Laboratory

Formalizing Causality as a Desideratum for Memory Models and Transformations of Parallel Programs

Chen Chen
Wenguang Chen
Vugranam Sreedhar
Rajkishore Barik
Vivek Sarkar
Guang Gao

CAPSL Technical Memo 089
July, 2009

Copyright © 2009 CAPSL at the University of Delaware

University of Delaware • 140 Evans Hall • Newark, Delaware 19716 • USA
http://www.capsl.udel.edu • ftp://ftp.capsl.udel.edu • capsladm@capsl.udel.edu

Abstract

It has been observed in previous work that it is desirable to avoid causal violations in any execution or transformation of a parallel program. In this paper, we formalize the notion of *causality* in memory consistency models and code transformations. For *memory models*, we introduce a framework of *causality graph* that can be used to analyze if a particular memory model violates causality. We show that a popular memory model as the Java memory model (JMM) [16], can lead to program executions that exhibit causality violations with respect to our definition of causality. The same analysis appears to also apply to a recent proposal of C++ specification [7] where the underline memory model may also lead to similar problems.

For *code transformations*, we identify transformations that are *causality-preserving* and those that are *potentially causality-violating*. We found that 10 of the 13 code transformation examples that were identified as causality-preserving with respect to the Java Memory Model fail our causality graph test and thus represent causality violations in our framework. Likewise, we also present examples to illustrate how the recently proposed C++ Memory Model can lead to potential causality violations.

Using our formalization, we establish causality as a desideratum for memory models and code transformations of parallel programs and define a Causal Memory Model (CMM) as the weakest memory model that preserves causality. We identify specific code transformations that are guaranteed to be causality-preserving. Finally, we present preliminary experimental results for a load elimination optimization to motivate the performance benefit of using the CMM model relative to the Sequential Consistency (SC) model. For the benchmark program studied, the number of getfield operations performed was reduced by 37.9% by using the CMM model instead of the SC model, and the execution time on a 16-core processor was reduced by 46.2%.

1 Introduction

A memory consistency model (or, memory model) is a contract between an application and a system that specifies the semantics of permissible values that a read operation of a memory location may return to a process or a thread of computation. Modern multiprocessor systems contain several levels of (nonuniform) memory that may cache the value of a memory location. There are fundamental challenges in specifying the memory model for a shared memory multiprocessor system which include the fact that several threads of computation may access the same memory location in parallel, and the fact that values corresponding to the same memory location may be resident in multiple physical locations in the memory hierarchy. It is desirable for the memory model to be intuitive for a programmer to understand, for compilers and software tools to work with, and to be portable across a wide-range of parallel systems. One of the most popular memory models is the Sequential Consistency (SC) model [13] in which memory operations must appear to execute as though they were performed one at a time in a serial order. SC is very easy for programmers to understand because it is a nature extension of uniprocessor memory consistency. However, it is well known that the requirement of serializability in SC limits performance optimizations that can be performed on applications in software and hardware. Modern day multi-core multiprocessor systems contain two major

sources of performance improvement: (1) multiple cores per processor for improving parallel processing, and (2) memory hierarchy for improving the latency of memory operations. Unfortunately, a strong memory model like SC can result in a performance bottleneck on both fronts. Several weaker memory models have been proposed to address these issues in the SC model, such as Weak Ordering model [5], Release Consistency model [11], and Location Consistency model [10].

For race-free programs, most of the weaker memory models guarantee the same result of an execution as if the execution was performed under the sequential consistency memory model. However, for programs with races it is still important to ensure some form of consistency. For instance the result of a read operation should be due to a prior write operation performed by some thread, and not be constructed "out of thin air". Ruling out such "out of thin air" reads is necessary but not sufficient for defining a useful memory model. The position taken by this paper is that it is desirable to ensure that the value of a read operation returned to a thread be the "effect" of some write that "caused" that value to be written to the location and not due to some "out of thin air" write, such as some write that may happen in the future.

In this paper, we introduce a *causal memory model* (CMM) that ensures that no reads are created out of thin air, and also ensures that a read value is the *effect* of some observable write that *caused* the value to be created. For race-free programs the causal memory model (CMM) also guarantees sequential consistency. For programs with race conditions, it guarantees a "cause-and-effect consistency". From the programmer's or hardware designer's point of view, CMM allows aggressive optimization without violating the underlying cause-and-effect consistency model. In this paper, we formalize the notion of *causality* in memory consistency models and code transformations. For *memory models*, we introduce a framework of *causality graph* that can be used to analyze if a particular memory model violates causality. We show that a popular memory model as the Java memory model (JMM [16]), can lead to program executions that exhibit causality violations with respect to our definition of causality. The same analysis appears to also apply to a recent proposal of C++ specification [7] where the underline memory model may also lead to similar problems.

For *code transformations*, we identify transformations that are *causality-preserving* and those that are *potentially causality-violating*. We found that 10 of the 13 code transformation examples that were identified as causality-preserving with respect to the Java Memory Model fail our causality graph test and thus represent causality violations in our notion. Likewise, we also present examples to illustrate how the recently proposed C++ Memory Model can lead to potential causality violations.

Using our formalization, we establish causality as a desideratum for memory models and code transformations of parallel programs and define the Causal Memory Model (CMM) as the weakest memory model that preserves causality. We identify specific code transformations that are guaranteed to be causality-preserving. Finally, we present preliminary experimental results for a load elimination optimization to motivate the performance benefit of using the CMM model relative to the Sequential Consistency (SC) model. For the benchmark program studied, the number of getfield operations performed was reduced by 37.9% by using the CMM

model instead of the SC model, and the execution time on a 16-core processor was reduced by 46.2%.

The rest of the paper is organized as follows: Section 2 introduces the motivating examples of CMM and causality graph. Section 3 introduces causality graph model and formally defines CMM. Section 4 demonstrates our analyses on the JMM and a recent proposal of the C++ memory model in our notion of causality. Section 5 discusses causality preserving transformations. Section 6 introduces the experimental results of load elimination. Section 7 introduces the related work. Finally, section 8 makes the conclusion.

2 Motivation

In this section we present two examples to motivate CMM. We will use a graph model, called the *causality graph* (CG). A CG consists of nodes representing operations and edges representing causality relation. Given a particular execution on a multiprocessor system supporting a particular memory model, and the corresponding output from the execution, we can use the CG to check if the execution and the corresponding output violates cause-and-effect consistency. The first example illustrates cause-and-effect inconsistencies that arise in the JMM and the second example illustrates a case that violates write atomicity while preserving causality.

2.1 Motivating Example 1

Consider the Java causality example from [16] shown in Figure 1. For this example, The question to address is whether a result of $r1 = r2 = r3 = 2$ violates cause-and-effect consistency. To construct the causality graph, we first create nodes corresponding to statements that produce the result set. For each result, we trace back to determine which statements caused the values in the result set. For instance, we insert a causality relation edge from node 1 to the result node r1=2. Similarly we also insert other causality relation edges, as illustrated in Figure 2. Notice that the resulting causality graph contains a cycle indicating that the an execution with output $r1 = r2 = r3 = 2$ is *not* cause-and-effect consistent. Let us further analyze why the behavior is not cause-and-effect consistent.

```
         Initially, a = 0, b = 1
Thread 1                     Thread 2
1: r1 = a;                   5: r3 = b;
2: r2 = a;                   6: a = r3;
3: if (r1 == r2)
4:    b = 2;
        result: r1 == r2 == r3 == 2
```

Figure 1: Example: A result violates cause-and-effect consistency.

The steps in the cycle are as follows:

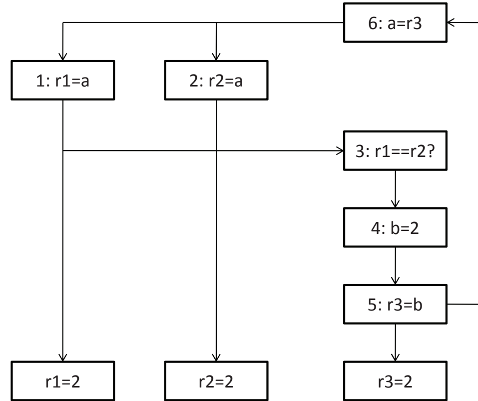

Figure 2: The causality graph corresponding to the example in Figure 1

1. Since in the result r1 equals 2, statement 6 must be finished before statement 1. Otherwise, r1 will get value 0.

2. For a similar reason, statement 6 must be finished before statement 2 too.

3. For a similar reason, statement 4 must be finished before statement 5.

4. Statements 1 and 2 must be finished before statement 3 because of the read-after-write data dependences on local variables r1 and r2 in Thread 1.

5. Statement 5 must be finished before statement 6 because of a read-after-write data dependence on local variable r3 in Thread 2.

6. Statement 3 must be finished before statement 4 because of local control dependence.

According to steps 1 and 2, statement 6 must be finished before statements 1 and 2. However, according to steps 3, 4, 5 and 6, statements 1 and 2 must be finished before statement 6. Hence, we establish a causality violation.

2.2 Motivating Example 2

```
          Initially, X=Y=0;
Thread 1   Thread 2   Thread 3
1: X=1;    2:r1=X;    4:r2=Y;
           fence;     fence;
           3:Y=1;     5:r3=X;
     result: r1==r2==1, r3=0
```

Figure 3: Example: A result violates write atomicity but preserves cause-and-effect consistency.

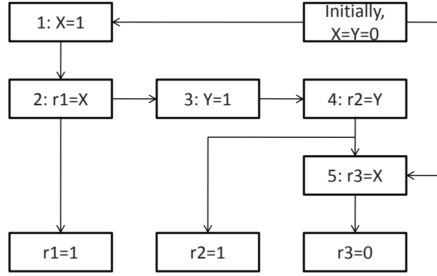

Figure 4: The causality graph corresponding to the example in Figure 3

Now, consider the second example shown in Figure 3 which was proposed as a counter-example in the recent proposal of the C++ memory model [7]. In the example, each fence ensures that the memory operation sequenced-before the fence will appear to execute before the memory operation sequenced after the fence. For this example, is it possible to obtain the result $r1 = r2 = 1$ and $r3 = 0$ at the end of the program's execution? One observation is that the result violates write atomicity [1] because statement 1 will overwrite the initial value of X if write atomicity is preserved. Therefore, if both r1 and r2 read value 1, r3 should also read value 1 under write atomicity. However, violation of write atomicity does not imply the violation of causality. Again, we first build CG (see Figure 4). From the CG we can see that there is no cycle in the graph. Therefore, it is possible to generate the result $r1 = r2 = 1$ and $r3 = 0$ along a time-line which goes through all of the statements without violating the causality relation. For example, it is possible that thread 3 owns a private cache which may not always be coherent with the shared memory. So it is possible that statement 1 overwrites the value of X in the shared memory but does not update the value in thread 3's cache. Thus statement 5 reads value 0 of X from thread 3's cache and writes it to r3. The following steps explain the details of constructing the CG.

1. Statement 1 must be finished before statement 2 because r1 gets value 1 in the final result. For a similar reason, statement 3 must be finished before statement 4, and initialization must be finished before statement 5.

2. Statement 2 must be finished before statement 3 due to the fence operation between them. For a similar reason, statement 4 must be finished before statement 5.

3. Initialization must be finished before any other statements. Because causality relation is transitive, we do not show the edges which can be derived from the transitivity. However, we keep the edge from the initialization to statement 5 to emphasize the causality relation for r3 to get value 0.

[1] The term "write atomicity" may have different meanings in different literature. In this paper, the meaning of "write atomicity" is the same as that in [4], i.e., writes to the same location should be serialized and a read cannot get the newly written value until the corresponding write becomes visible to all processors (threads). In some other literature such as [10], the term "coherence" has the same meaning as that of "write atomicity" in this paper.

3 Causality and Causal Memory Model

In this section, we introduce the notion of causality in memory models.

3.1 Background and Notation

In this section we present relevant background and notation that are needed to understand the rest of the paper. For simplicity we assume that a program P is made of a set of threads T, and each thread t contains a set of actions A_t. The program order for an execution is a partial order on the actions of the execution imposed by the program text [3, 19]. If $a \in A_t$ is executed before $a' \in A_t$ in t, then we say that a and a' are related by the program order relation (denoted as $a \stackrel{po}{\rightarrow} a'$). We assume that each action is atomic. For simplicity we also assume that each action can access no more than one shared variable.

In discussing memory models it is often desirable to classify actions into different classes, such as memory read and write actions, synchronization actions, and control flow actions. Without loss of generality, in this paper we assume that the synchronization actions are paired *acquire* and *release* synchronization operations as defined in [3] where *acquire* represents the semantic of entering a critical section and *release* represents an exit from a critical section. The synchronization order ($\stackrel{so}{\rightarrow}$) over synchronization actions specifies the order to complete the synchronization actions in an execution. We also define the happens-before relation $\stackrel{hb}{\rightarrow}$ as the reflexive transitive closure of $\stackrel{po}{\rightarrow}$ and $\stackrel{so}{\rightarrow}$.

Given a classification of actions, we can then define an execution E of a program P as an ensemble of a set of actions within a set of threads of P. Formally, an execution e of program P (denoted as $e \in E(P)$ where $E(P)$ is a set of all executions that P can generate) is defined as a tuple $e =< A, \stackrel{po}{\rightarrow}, \stackrel{so}{\rightarrow}, W, V >$ in which

- A is the set of all the actions that are executed in this execution. If an action is executed multiple times (e.g. an action in loop) – say n times, the action should be considered as n different actions in A where each one corresponds to an instance.
- $\stackrel{po}{\rightarrow}$ is the program order over all of the actions in A.
- $\stackrel{so}{\rightarrow}$ is the synchronization order over synchronization actions in A.
- W is a write-seen function. For each action r in A which reads a shared variable x, $W(r)$ is the write action in A which writes x and is seen by r. We require that happens-before consistency [16] is satisfied, that is, only the following two cases are prohibited in W: (1) $r \stackrel{hb}{\rightarrow} W(r)$; (2) $\exists w'(w'$ writes $x \wedge W(r) \stackrel{hb}{\rightarrow} w' \stackrel{hb}{\rightarrow} r)$.
- V is a value-written function. For each action w in A which writes a shared variable, $V(w)$ is the value written by w.

An execution should be consistent with the program, that is, W and V must be consistent with the control flow actions (statements) of the program, and A must be consistent with the intention of the program text, i.e., A does not miss any action that must be executed according to the program text and does not contain any action that must not be executed.

Next we introduce the notion of causal order. For a given execution, and for all the real machines which can generate the execution, if all these machines have to finish an action a_1 before finishing another action a_2 to generate the execution, we say that a_1 is causally ordered before a_2, and is denoted as $a_1 \stackrel{co}{\rightarrow} a_2$ in the execution.

To simplify the presentation we focus on the following kinds of causal order:

- a_1 writes a value to a variable and a_2 reads that value from the same variable. Any real machine has to finish a_1 before a_2 because in the reverse order a_2 cannot read the value which will be written by a_1 in the future. Note that even a machine which is able to do value speculation cannot violate this order. The reason is that a_2 cannot be committed before a_1 is committed.

- a_2 is control dependent on a_1. It includes uniprocessor control dependence such as conditional branch as well as interprocessor control dependence such as fork and join. Note that a machine which is able to do control speculation may start a_2 before starting a_1. However, a_2 cannot be committed before the commitment of a_1.

- $a_1 \xrightarrow{so} a_2$.

- There exists an action a_3 where $a_1 \xrightarrow{co} a_3$ and $a_3 \xrightarrow{co} a_2$. This is the transitivity property of the causal order.

Next we discuss observable behavior of an execution. An output action is an action which outputs an observable result (e.g. a printf function in the C language). Without loss of generality, we assume that one output action only outputs the value of one local variable. The observable behavior of the execution is a multiset of the values that all of the output actions output, together with the output order over the values. Formally, the observable behavior of an execution e is a tuple $ob(e) =< V, \xrightarrow{vo}>$ where V is the multiset of the values that all of the output actions output and $\xrightarrow{vo}$ is the value order. The value order is defined as follows: Let a_1 and a_2 be two output actions, v_1 and v_2 be the outputted values of a_1 and a_2, respectively. $v_1 \xrightarrow{vo} v_2$ if and only if $a_1 \xrightarrow{po} a_2$. Note: it is possible that another output action outputs the same value as v_1 or v_2. But the value orders of that value are not affected by the value orders of v_1 or v_2. For example, suppose a_3 outputs v_3 where $v_3 = v_2$, since v_2 and v_3 are distinguishable in the multiset V even if v_2 and v_3 are equal, $v_1 \xrightarrow{vo} v_2$ does not imply $v_1 \xrightarrow{vo} v_3$.

Sometimes, the observable behavior of an execution is simply represented as the final values of some local variables. For example, as it is shown in Figure 1, the observable behavior of the execution is represented as $r1 = r2 = r3 = 2$. To handle this kind of simple representation, we view it as adding output actions of the local variables to the end of the threads. If multiple

output actions are added to the same thread, they are arranged in the alphabetical order of the variable symbols.

3.2 Definitions of Causality Preserving Execution and Memory Model

With the given notions in the last section, now we can define causality preserving executions and memory models in this section.

We say an execution $e =< A, \xrightarrow{po}, \xrightarrow{so}, W, V >$ violates causality when a part of its causal order is cyclic. Formally, e violates causality if and only if $\exists a_1 \exists a_2 (a_1 \in A \wedge a_2 \in A \wedge a_1 \xrightarrow{co} a_2 \wedge a_2 \xrightarrow{co} a_1)$. If an execution does not violate causality, we say that the execution preserves causality.

If an execution violates causality, it cannot be generated by hardware because it is impossible for hardware to finish two actions a_1 and a_2 where a_1 must be finished before a_2 and also a_2 must be finished before a_1.

A memory model M violates causality if and only if it allows some observable behavior which can only be outputted by the executions that violate causality. The formal definition is as follows: M violates causality if and only if $\exists e \exists P(e \in E(P) \wedge M$ allows $ob(e)$ on $P \wedge \forall e'(e' \in E(P) \wedge ob(e') = ob(e) \rightarrow e'$ violates causality $))$. If a memory model does not violate causality, we say that the memory model preserves causality.

If a memory model violates causality, it cannot be implemented exactly by hardware because the memory model allows the observable behavior which can only be outputted by causality violating executions, where no hardware can output the observable behavior. Note, although a compile time or runtime transformation may transform a causality violating execution to a causality preserving execution, such a transformation violates the intention of causality of the program and should be considered as unreasonable. The details of causality preserving and causality violating transformations (mainly from the compiler angle) will be discussed in Section 5.

3.3 Causality Graph Analysis

As we defined in Section 3.2, an execution violates causality when a part of its causal order is cyclic. Therefore, we can construct a graph in which the nodes represent the actions and the directed edges represent the causal order of the nodes. Because of the transitivity property of causal order, if the execution violates causality, the graph will contain a cycle. We call the graph "causality graph" and the process "causality graph analysis". Formally, the causality graph of an execution e is a directed graph $CG(e) =< V, E >$ where $V = A$ of e and $E = \xrightarrow{co}$. The causality graph analysis consists of two steps as follows:

1. To construct $CG(e)$ for the given execution e.

2. To check if $CG(e)$ contains a cycle. If so, e violates causality. Otherwise, e preserves causality.

In Section 4, by using the causality graph analysis, we will illustrate that the Java memory model and the recently proposed C++ memory model violate our notion of causality.

3.4 Definition of Causal Memory Model

In this section, we formalize the definition of causal memory model (CMM) which is the weakest memory model that still preserves causality. Our formalization follows the approach used by Lamport to formalize sequential consistency [13] and Adve et al. to formalize weak ordering and data-race-free-1 [5, 3]. The main idea includes the following formalizations:

- The definition of executions.
- The properties that the executions must obey under the memory model.
- The memory model only allows the observable behaviors of the executions that obey the properties. Formally, a memory model is a function M: *set of programs* $\mapsto$ *set of observable behaviors*. $\forall P(M(P) = \{ob(e)|e \in E(P) \wedge e$ obeys the defined properties $\})$.

In Section 3.2, we have formalized causality preserving executions. Now we define CMM as follows:

Definition 3-1: The causal memory model is a function M: *set of programs* $\mapsto$ *set of observable behaviors*. $\forall P(M(P) = \{ob(e)|e \in E(P) \wedge e$ preserves causality $\})$.

Definition 3-2: Hardware obeys CMM if and only if the observable behavior of every execution on the hardware can be obtained by an execution of the program which preserves causality.

We say an execution has data race if there exists two actions which access on a same location, at least one writes the location, and they are not ordered by the happens-before relation. We say a program is data-race-free with respect to CMM if all of its causality preserving executions has no data race. The formal definition is as follows:

Definition 3-3: A program P is data-race-free with respect to CMM if and only if $\forall e(e \in E(P) \wedge e$ preserves causality $\rightarrow e$ has no data race. $)$

CMM guarantees sequential consistency for the programs which are data-race-free with respect to CMM because for such a program each read action can only see one write action in an given execution which preserves causality. (Otherwise, the execution has data race.) Since we also require that an execution must be happens-before consistent, the execution should looks as if executed in some total order (which is consistent to the happens-before relation of the execution) where each individual thread is executed in its program order.

4 Causality Analysis of Memory Models

In this section, we discuss the causality analysis of the Java memory model (JMM) [16] and the recent proposal of C++ memory model (C++MM) [7]. JMM community has proposed a suite of 20 causality test cases [17] that can help compiler writers and JVM implementers to use to verify the consistency of their implementation with respect to JMM. Although these 20 test cases are not complete JMM compliance test suite, they provide valuable insight into the working JMM, especially for Java programs with data races. In this section we will show that ten of the JMM causality test cases which preserve the JMM notion of causality do indeed violate our notion of causality. In one case (i.e., case 12), it is suggested that the test case violates JMM notion of causality, but interestingly the same test case does not violate our notion of causality. We will also demonstrate that the recently proposed C++MM appears to violate our notion of causality.

4.1 Analyses of the Java Memory Model

# of case	Violation of causality	Forbidden under JMM
1	Y	N
2	Y	N
3	Y	N
4	Y	Y
5	Y	Y
6	Y	N
7	N	N
8	Y	N
9	Y	N
10	Y	Y

# of case	Violation of causality	Forbidden under JMM
11	N	N
12	N	Y
13	Y	Y
14	Y	Y
15	Y	Y
16	N	N
17	Y	N
18	Y	N
19	Y	N
20	Y	N

Figure 5: Summary of analyses on Java causality test cases

In this section, we introduce the result of using causality graph to analyze the Java Causality Test Cases. The summary of our analysis is shown in Figure 5. In the table, the label "# of case" represents the number of the cases. The label "Violation of causality" represents the result of using causality graph analysis on the case. Finally, the label "Forbidden under JMM" means whether JMM forbids the execution. For example, the second row of the left table means that for the Java causality test case-1, our causality graph analysis shows the execution violates causality. However, the Java memory model does not prohibit the execution of case-1.

The table shows that 10 of the 13 positive examples violate our notion of causality. In the following section, we analyze the Java causality test case 1 and show that the Java memory model violates our notion of causality. The analyses of the other cases are quite similar to the analysis of case1, thus we omit them due to space limitations.

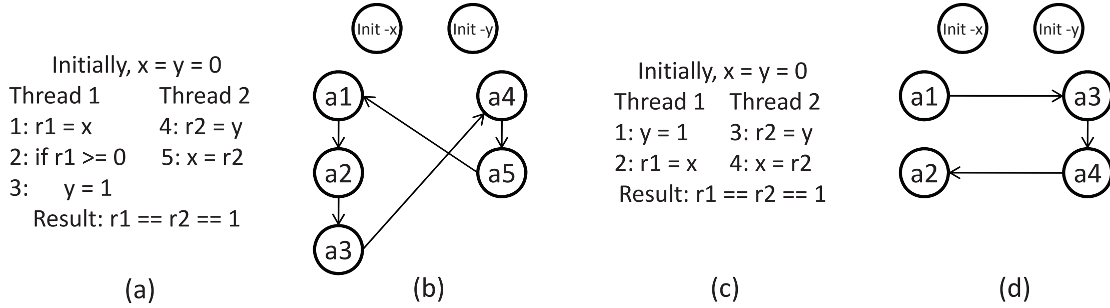

Figure 6: Analysis on Java causality test case-1. (a) The original program with an observable behavior. (b) Causality graph of the execution that generates the observable behavior in (a). (c) A possible transformed program with the same observable behavior. (d) Causality graph corresponds to (c).

4.1.1 Java Causality Test Case 1

The program and the observable behavior of case-1 are shown in Figure 6 (a), in which $r1$ and $r2$ are local variables and x and y are shared variables. Figure 6 (c) shows a possible transformation of the program which removes statement 2 because the condition is always true and then reorders statement 3 to the beginning of thread 1. The observable behavior {r1==r2==1} satisfies sequential consistency in (c). Since the JMM allows the transformation from the program of Figure 6 (a) to (c), JMM allows the observable behavior {r1==r2==1} in (a). It implies a strong connection between the concept of causality preserving executions and memory models and the concept of causality preserving transformations. In this section, we focus on the concept of causality preserving executions and memory models. The details about transformations will be discussed in Section 5.

Let $e =< A, \xrightarrow{po}, \xrightarrow{so}, W, V >$ be a execution with the observable behavior {r1==r2==1}. In the following analysis, we will show that such an execution is unique.

- The initial statements are two actions. One initializing x and the other initializing y. They are labeled as $init - x$ and $init - y$, respectively. The statements from 1 to 5 are five actions (labeled as a_1 to a_5), respectively.
- All of the actions except a_3 must be in A because they do not control dependent on any action. a_3 is also in A because it supplies the value “1” where all the other actions cannot. Therefore, A is unique because it must contain all of the actions.
- Since A is unique, $\xrightarrow{po}$ is also unique.
- $\xrightarrow{so}$ is unique because it must be empty.
- Now we discuss the write-seen function W. a_4 may see a_3 or a_{init-y}. However, $W(a_4) = a_3$ because the value of $r2$ must be “1” finally. For a similar reason, $W(a_1) = a_5$. So W is fixed and unique.

- V defines the value-written function. Since W is unique, V is also unique in which $V(a_{init-x}) = 0$, $V(a_{init-y}) = 0$, $V(a_3) = 1$, and $V(a_5) = 1$.

$CG(e)$ is shown in Figure 6 (b) with transitive reduction, i.e., any an edge that can be deduced by transitive property is not shown in the figure. It is clear that $CG(e)$ contains a cycle $a_1 \xrightarrow{co} a_2 \xrightarrow{co} a_3 \xrightarrow{co} a_4 \xrightarrow{co} a_5 \xrightarrow{co} a_1$. So e violates causality. As we analyzed above, e is the unique execution of which the observable behavior is {r1==r2==1}. So the Java memory model violates causality.

4.2 Analysis of the C++ Memory Model

In this section, we demonstrate that the recent proposal of C++ memory model [7] violates causality. First, we will analyze the data-race-free-0 model [5] and show that the data-race-free-0 model violates our notion of causality. Then we will demonstrate the strong connection between the data-race-free-0 model and the recently proposed C++MM, thus the recent proposal of C++MM appears to also violate our notion of causality.

Adve and Hill defined the data-race-free-0 model in [5]. The data-race-free-0 model requires the programs to be data-race-free. For the programs that contain data races, the semantics of the programs are undefined in data-race-free-0 model. Therefore, for a given program that contains data race, data-race-free-0 model allows any observable behavior of the program. Now let P be the program in Figure 6 (a). P contains data race because actions 1 and 5 can be executed concurrently where action 1 reads x and action 5 writes x. So data-race-free-0 should allow the result {r1==r2==1}. According to our analysis in Section 4.1.1, data-race-free-0 violates causality.

A simple way to view the recently proposed C++MM is as addressed in [7]: *"The model chosen for C++ is an adaptation of data-race-free-0; i.e., it guarantees sequential consistency for programs without data races and has undefined semantics in the presence of a data race."* Therefore, it appears that the recently proposed C++MM also violates our notion of causality.

5 Causality and Program Transformations

In previous section, we showed how to use causality analysis to study memory models and detect executions that violate causality. In this section, we continue to show that causality analysis can also be productively applied to compiler optimization. We show that, under the causal memory model (CMM) defined in Section 3.4, the (seemingly static and local in the uniprocessor sense) causality preserving property of a family of useful program transformations will not create causality violation in the global sense (of a parallel program as a whole) - a very desirable feature of our causality analysis framework. We also show that certain such causality preserving program transformations under CMM may not be permitted under sequential consistency model. Further experimental evidence of the usefulness of such findings will be provided in the next section (Section 6).

5.1 Legal Transformation in a Memory Model

In this section, we introduce the concept of legal transformation under a given memory model. For convenience, in the rest of Section 5, T denotes a transformation, P denotes the original program of T, and Q denotes the transformed program of T where $Q = T(P)$.

We say T is a legal transformation under a memory model M if and only if the set of possible observable behaviors of Q is a subset of the set of possible observable behaviors of P in M. This is known as the "subset correctness" criteria in [14], and we will use this criteria as the basis for understanding causality preserving transformations in the following section.

5.2 Causality Preserving Transformations

In this section, we introduce causality preserving transformation (*CT*). Informally, *CT*s are transformations which are safe with respect to causality for memory models, i.e., the permission of *CT*s will not cause a memory model to violate causality. We will also demonstrate that a *CT* is always a legal transformation under CMM, which helps the verification of legal transformations under CMM.

The intuition of *CT* is shown in Figure 6. In the figure, consider the transformation which transforms the program P in (a) to the program Q in (c). The causality graph in (d) shows that a causality preserving execution of Q can generate the observable behavior {r1==r2==1}. However, as we analyzed in Section 4.1, the execution of P which can generate the same observable behavior violates causality. The example of transformation illustrates a potential violation of causality. If the memory model allows the transformation and the observable behavior corresponds to the the causality preserving execution in Q, the memory model must also allow the observable behavior in P due to subset correctness. However, all of the executions in P that can generate the observable behavior violate causality. So the memory model violates causality.

A *CT* is a transformation which guarantees that any causality preserving executions in the transformed program is also causality preserving in the original program. The formal definition is as follows:

Definition 5-1: T is a *CT* if and only if $\forall e_Q(e_Q \in E(Q) \wedge e_Q$ preserves causality $\rightarrow \exists e_P(e_P \in E(P) \wedge e_P$ preserves causality $\wedge ob(e_P) = ob(e_Q)))$.

Note that in our definition, *CT* does not rely on a definition of memory model. (However, the definition of subset correctness does.) So the study of *CT* does not require the knowledge of memory model either. We have the following two claims on the properties of *CT*. The first claim implies that *CT* is safe with respect to causality for memory models. The second claim connects the study of *CT* to the study of legal transformation under CMM. The proof sketches can be found in Appendix.

Claim 5-1: If a memory model M only allows *CT*s, M preserves causality.

Claim 5-2: If T is a *CT*, T is a legal transformation under CMM.

5.3 Subgraph Analysis

In this section, we propose the subgraph analysis which can be used to verify causality preserving transformations. As we will see in Section 5.4 and 5.5, in many cases of compiler transformations, subgraph analysis is a local analysis because it only requires the analysis on the fragment of code which is involved in the transformation. It is significantly different from previous work on delay-set analysis [19, 14] in sequential consistency model as delay-set analysis normally requires a global analysis over the whole program.

Our subgraph analysis is based on two theorems. Informally, the first theorem can be explained as follows: Given an original program P and a transformed program Q, the corresponding transformation T is a causality preserving transformation (CT) if the following two conditions are satisfied: (1) For any execution e_Q in Q, we can always find a corresponding execution e_P in P which has the same observable behavior, and (2) The causality graph (CG) of e_P is a subgraph of the CG of e_Q. The formal theorem is as follows:

Theorem 5-1: If $\forall e_Q(e_Q \in E(Q) \rightarrow \exists e_P(e_P \in E(P) \wedge ob(e_P) = ob(e_Q) \wedge CG(e_P)$ is a subgraph of $CG(e_Q)))$, then T is a CT.

Proof sketch: Because $CG(e_P)$ is a subgraph of $CG(e_Q)$, if $CG(e_Q)$ has no cycle, $CG(e_P)$ has no cycle either. So if e_Q preserves causality, e_P preserves causality too. Therefore, T satisfies the definition of CT. ■

In practice, when we are applying theorem 5-1, we may hope to reduce the complexity of the causality graph without changing the cyclic property of the graph, where cyclic property means the assertion that whether the graph has a cycle. For example, if a node in CG has zero indegree (or outdegree), removing such a node and the edges which are connected to it will not change the cyclic property. Therefore, we have the following theorem to address this case:

Theorem 5-2: Let $F(CG(e_P))$ denote a graph which removes one zero indegree (or outdegree) node and the edges that are connected to $CG(e_P)$. If $\forall e_Q(e_Q \in E(Q) \rightarrow \exists e_P(e_P \in E(P) \wedge ob(e_P) = ob(e_Q) \wedge F(CG(e_P))$ is a subgraph of $CG(e_Q)))$ then T is a CT.

Proof sketch: Because $F(CG(e_P))$ is a subgraph of $CG(e_Q)$, if $CG(e_Q)$ has no cycle, both $F(CG(e_P))$ and $CG(e_P)$ have no cycle. So if e_Q preserves causality, e_P preserves causality. Therefore, T satisfies the definition of CT. ■

5.4 Load elimination

Load elimination is a classical compiler transformation that replaces a memory access by a read of a compiler generated temporary. This transformation is also known as *Scalar replacement* in past work [12]. Figure 7 depicts the two cases of load elimination transformation. In this section, we demonstrate that the load elimination transformations in Figure 7 are CTs. For the code fragments, x is shared, r, $r1$ and $r2$ are local, and *temp* is a compiler generated temporary local variable. For convenience, we label the statements as 1 (or 1a and 1b in the transformed code in (a)), 2, and 3. For both the cases in Figure 7 (a) and (b), before the transformation

```
Original code:        Transformed code:
Ti                    Ti
1: x = ...;           1a: temp = ...;
2: ...                1b: x = temp;
3: r = x;             2 : ...
                      3 : r = temp;
```

(a)

```
Original code:        Transformed code:
Ti                    Ti
1: r1 = x;            1: r1 = x;
2: ...                2: ...
3: r2 = x;            3: r2 = r1;
```

(b)

Figure 7: Load eliminations

is applied, statement 3 loads the value of a shared variable. After the transformation, the load of the shared variable is eliminated and replaced by a read of a local variable. Note that, it is possible that there are some statements program ordered before statement 1 in thread T_i.

Now assume that statement 2 (in both (a) and (b)) satisfies the following constraints:

- No action in statement 2 changes the value of x.
- No action jumps into statement 2 without going through statement 1.
- Statement 2 may introduce synchronization order using either *acquire* or *release*, but not both.
- Statement 2 may introduce side-effects for shared variables other than x.

If statement 2 satisfies the above constraints, the read in statement 3 can see the write in statement 1 without violating the happens-before consistency. Now we claim that the load elimination in Figure 7 (a) preserves causality. The reason is as follows: For any execution $e_Q \in E(Q)$, let e_P be an execution of which the only difference from e_Q is that r sees the write of x in statement 1 instead of *temp* and in statement 1 x gets value from "..." directly. Therefore, $e_P \in E(P)$. The different parts of $CG(e_P)$ and $CG(e_Q)$ is shown in Figure 8. Because of transitive property of causal order, for each edge from node a_{1b} to a node c in $CG(E_Q)$, there is also an edge from node a_{1a} to c. Therefore, $CG(e_P)$ is a subgraph of $CG(e_Q)$ because we can rename node a_{1a} by a_1 in $CG(e_Q)$. According to theorem 5-1, the transformation preserves causality and thus it is legal under CMM. For a similar reason, the load elimination in Figure 7 (b) preserves causality too.

Although the above load eliminations preserve causality, they may not be permitted by all of the causality preserving memory models. For example, the load eliminations shown in Figure 9 match the load eliminations in Figure 7. However, they are prohibited by the sequential consistency because the transformations are not subset correct under SC.

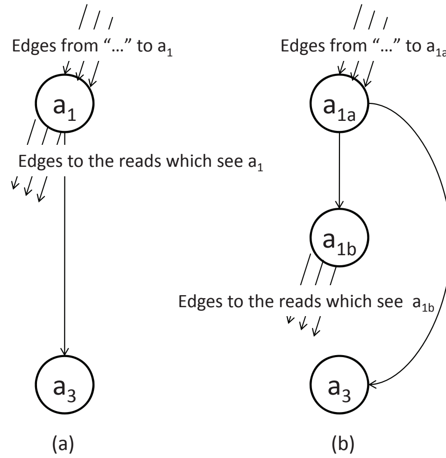

Figure 8: Causality Graphs (with transitive reduction) of Program P (a) and Q (b). Only different parts are shown.

5.5 Other Transformations

In this section, we discuss a number of transformations that include reordering, constant propagation, redundant store elimination and partially redundant store elimination. We will demonstrate that all of them are legal (subset correct) under CMM.

Original code:

```
Initially r = x = y = 0;
T1
1 : x = 1
2 : if (y == 0)
3:     goto 5;
4 : r = x;
5 : ...
T2
6 : x = 2
7 : y = 1;

The result r == 1 is disallowed under SC.
```

Transformed code:

```
Initially r = x = y = 0;
T1
1a: temp = 1
1a: x = temp
2: if (y == 0)
3:     goto 5;
4 : r = temp;
5 : ...
T2
6 : x = 2
7 : y = 1;
The result r == 1 is allowed under SC.
```

(a)

Original code:

```
Assume compiler does not know p==q
Initially p.x = q.x = 0
T1
1 : r1 = p.x
2 : r2 = q.x
3 : r3 = p.x
T2
4 : p.x = 1
T3
5: q.x = 2;

The result r1 == 1, r2 == 2, r3 == 1 is
disallowed under SC.
```

Transformed code:

```
Assume compiler does not know p==q
Initially p.x = q.x = 0
T1
1 : r1 = p.x
2 : r2 = q.x
3 : r3 = r1
T2
4 : p.x = 1
T3
5: q.x = 2;

The result r1 == 1, r2 == 2, r3 == 1 is
allowed under SC.
```

(b)

Figure 9: The load eliminations are prohibited by sequential consistency because the transformations introduce new results.

Reordering

We discuss the most common case of reordering, i.e., two actions (statements) that do not have uniprocessor data dependency, uniprocessor control dependency, and neither is a synchronization action. We also assume that the two actions are next to each other. The more general case is discussed in the end of Section 5.5. In the above case, the reordering is a *CT*. The reason is as follows: Let a_1, a_2 be the two actions that are reordered in the transformation. Before transformation, $a_1 \xrightarrow{po} a_2$; after transformation, $a_2 \xrightarrow{po} a_1$. Now for a given execution e_Q of the transformed program, let e_P be an execution of the original program of which the only difference from e_Q is that a_1 and a_2 have no causal order in e_P. Such e_P always exists because a_1 and a_2 have neither dependency nor synchronization order in original program. $CG(e_P)$ is a subgraph of $CG(e_Q)$ because $CG(e_P)$ has less (or the same) edges in comparison to $CG(e_Q)$. According to theorem 5-1, the reordering is a *CT* and thus it is legal under CMM.

Constant propagation

We show that constant propagation in Figure 10 is a *CT*. Again, the more general case is discussed in the end of Section 5.5.

Assuming r is local, x is shared.

Before transformation:	**After transformation:**
Ti	Ti
...	...
k : x = 1;	k : x = 1;
k+1: r = x;	k+1: r = 1;
...	...

Figure 10: Constant propagation

For a given execution e_Q of the transformed program, let e_P be the execution of the original program of which the only difference from e_Q is that action a_{k+1} gets value from a_k in e_P. Let F be a function which removes node a_k in $CG(e_P)$. $F(CG(e_P))$ is equivalent to $CG(e_Q)$. So $F(CG(e_P))$ is a subgraph of $CG(e_Q)$ because a graph is a subgraph of itself. Note that the indegree of node a_k in $CG(e_P)$ is zero. According to theorem 5-2, the transformation is a *CT* and thus it is legal under CMM.

Redundant store elimination

We show that the redundant store elimination in Figure 11 is a *CT*.

We only discuss the case that the two actions (or statements) are next to each other. The case that the two actions do not next to each other is discussed in the end of Section 5.5.

For a given execution e_Q of the transformed program, let e_P be an execution of the original program of which the only difference from e_Q is that it has one more action a_k which does not be read by any other read actions. So the only difference between $CG(e_Q)$ and $CG(e_P)$ is that

Assuming r1 and r2 are local, x is shared.

Before transformation:	**After transformation:**
Ti	Ti
...	...
k : x = r1;	k+1: x = r2;
k+1: x = r2;	...
...	

Figure 11: Redundant store elimination

$CG(e_P)$ has one more node a_k in e_P and some edges connected to it. Let F be a function which removes node a_k in $CG(e_P)$. $F(CG(e_P))$ is equivalent to $CG(e_Q)$. So $F(CG(e_P))$ is a subgraph of $CG(e_Q)$. Note that the outdegree of node a_k in $CG(e_P)$ is zero. According to theorem 5-2, the transformation is a CT and thus it is legal under CMM.

Partially redundant store elimination

In this section, we will show that partially redundant store elimination in Figure 12 can be applied under CMM. The more general case is discussed in the end of Section 5.5.

Assuming r1 and r2 are local, x is shared.

Before transformation:
```
Ti
...
x = r1;
if (cond.) //"x = r1" and
//"cond." have no dependency.
     x = r2;
else
   {...}
```

After transformation:
```
Ti
...
if (cond.)
    x = r2;
else
    {x = r1; ...}
```

Figure 12: Partially redundant store elimination

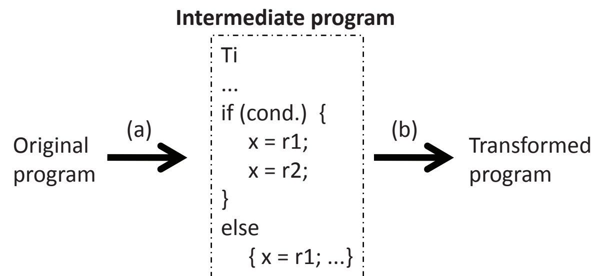

Figure 13: The two steps of partially redundant store elimination. (a) Reordering. (b) Redundant store elimination.

The partially redundant store elimination can be done by applying two transformations in sequence. As it is shown in Figure 13 (a), the first transformation moves the statement $x = r1$ into both paths of the conditional branch. No matter which path the conditional branch will

# getfield (original)	# getfield after SC load elim.	# getfield after CMM load elim.	No load elim. exec. time in sec	SC load elim. exec. time in sec	CMM load elim. exec. time in sec
1.19E10	7.91E09	4.91E09	22.3	18.38	12.68

Table 1: Moldyn Benchmark: reduction in dynamic counts of GETFIELD operations and execution time on a 16-core system.

go, the transformation matches the reordering as we discussed earlier in this section because $x = r1$ and the computation of the condition have no dependency in the original code. So it is legal under CMM. Figure 13 (b) shows the second transformation which is a redundant store elimination in one path of the conditional branch. As we discussed earlier in this section, this redundant store elimination is also legal under CMM.

Transformations in More General Cases

In the above of Section 5.5, we only discuss about the transformations on the actions that are next to each other. For the more general cases that the actions which will be transformed are far apart, we can firstly apply reorderings to move the actions next to each other (of course, the reorderings must be legal transformations) and then apply the transformation on the actions that are next to each other. Finally, it may be necessary to apply reorderings to move the transformed actions back to satisfy the original program order of the actions.

6 Preliminary Experimental Results

While the major contribution of our paper lies in the formalization of causality as a desirable property for memory models and program transformations, we present some preliminary experimental results for the JGF Moldyn benchmark [1] in this section to illustrate the potential performance benefits of using the Causality Memory Model (CMM) instead of the Sequential Consistency (SC) model. For the CMM version, the transformation is summarized in Section 5.4. For the SC version, the transformation was performed by using *delay set analysis* [19] to restrict opportunities for load elimination. Note that, the load elimination transformation under CMM model preserves all the constraints described in Section 5.4.

The JGF Moldyn benchmark was written using a subset of the `X10` language that includes of the `async`, `finish` and `isolated` parallel constructs [9]. The construct `async S` is used for creating a light-weight asynchronous task that executes `S`; the `finish S` construct causes the executing task to wait for the termination of all tasks created within `S`; and finally `isolated S` is intended to be executed by a task as if in a single step during which all other concurrent tasks are suspended.

The performance results were obtained using Jikes RVM [2] version 3.0.0 on a 16-core system that has four 2.40GHz quad-core Intel Xeon processors running Red Hat Linux (RHEL 5), with 30GB of main memory. For our experimental evaluation, we use the *production* configuration of Jikes RVM with the following options:`-X : aos : initial_compiler=opt -X : irc : O0`. By default, Jikes RVM does not enable SSA based HIR optimizations like load elimination at optimization level O0. We modified Jikes RVM to enable the SSA and load elimination phases at O0. However, since the focus of this paper is on optimizing application classes, the boot image was built with load elimination turned off and the same boot image was used for all execution runs reported in this paper. Load elimination transformation was extended to perform load elimination for CMM and SC memory models.

All results were obtained using the `-Xmx2000M` JVM option to limit the heap size to 2GB, thereby ensuring that the memory requirement for our experiments was well below the available memory on the 16-core Intel Xeon SMP. The `PLOS_FRAC` variable in `Plan.java` was set to $0.4f$ for all runs, to ensure that the Large Object Size (LOS) was large enough to accommodate all benchmarks.

The summary of the results for the Moldyn benchmark are summarized in Table 1. We observe that the load elimination algorithm using the CMM memory model results in a 37.94% reduction of getfield operations compared to the SC memory model. The execution time reduction on a 16-core processor is 46.2%, confirming the importance of this transformation for scalable parallelism. *The final version of the paper will include performance comparisons for additional benchmarks.*

7 Discussion and Related Work

In this paper we introduced a new notion of causality in which the value returned by a read from a memory location is due to some causal write to the same location. We also introduced a graph model, called the causality graph, for analyzing the causality of an execution. We showed that if the causality graph contains a cycle the corresponding execution is causally inconsistent. Interestingly, existing literature contains many different notions of causality. Lamport presented a very simple consistency model, the sequential consistency (SC) model, that is causally consistent, even according to our notion of causality [13]. One drawback that is very well known in the literature is that the SC model restricts many useful compiler and hardware optimizations. Subsequently weak models were proposed that allowed certain kinds of compiler and hardware optimizations. The weaker models guaranteed SC for data race free programs, but had very few guarantees for programs with races. There are several other work on memory and consistency model. Due to limited space we will not dwell into each one of them. In this section we focus on those models that focus on the causality aspects of memory model.

In [19], Shasha and Snir proposed a delay-set analysis that computes the minimal ordering between shared variable accesses that is required to guarantee sequential consistency. The delay-set analysis is based on the construction of a conflict graph and (minimal) cycle detection

in the graph. Delay set analysis was probably one of the earliest work that paved way for other works in understanding the kind of code transformations that are possible and do not violate SC. Our causality analysis, especially for determining causality preserving transformation is influenced by delay-set analysis.

Ahamad et al. defined causal memory in [6]. In their work, causal order is defined as the transitive closure of write-in order and program order. Due to the need to fully respect program order when performing optimization, their causal memory model may restrict on the kinds of optimizations that can be performed under our causal memory model. In [15], Linder and Harden defined the "access graph" graph model which represents the causal relationships between load, store, and synchronization events. In their notion of causality, program order should also be respected. Therefore, their causality model hinders many compiler and runtime optimizations either.

Manson et al. defined the Java memory model that also uses some notion of causality [16]. The main purpose of their definition of causality is to allow as many compiler transformations as possible and still prevent out-of-thin-air behavior for data race programs. However, under their definition of causality, a hardware may execute one operation more times than programmer desired. For example, for the program in Figure 6 (a), statement 1 must be executed exactly once according to programmer's desire. However, to achieve the result {r1==r2==1} under JMM, statement 1 must be executed multiple times in the JMM commit process as defined in Section 5.4 of [16]. This might violate the programmers' intention of the program thus we feel that the semantic of their causality is not very intuitive from programmers' perspective.

In [18], Saraswat et al. proposed a family of memory models (Relaxed Atomic + Ordering family) which preserves sequential consistency for data-race-free programs. They discussed the Java causality test cases and showed that their family of memory models satisfies the cases. For example, it allows the observable behavior of the Java causality test case 1. It implies that their family of memory models violates our notion of causality.

Boehm and Adve discussed causality by using three examples (Figure 5,6, and 7 in [7]) which they claim to be violating causality. We used one of them as a motivating example in Figure 3. However, they do not explicitly define the notion of causality in their work. Since all of the three examples violate write atomicity. A reasonable conjecture is that their idea of causality requires that write atomicity must be satisfied. It seems that it is too expensive to implement write atomicity in large scale multi-core or many-core architectures.

The CMM defined in this paper is not a unique memory model that does not satisfy write atomicity. A typical memory model in the previous work that does not satisfy write atomicity is the location consistency memory model which is proposed by Gao and Sarkar [10], in which each memory location is viewed as a partially ordered multiset (pomset). A read operation may arbitrarily get one of the values in the pomset of the memory location. The OpenMP memory model as defined in [8] by Bronevetsky and de Supinski is another example because the OpenMP memory model is strictly weaker than the location consistency memory model.

8 Conclusion

In this paper, we introduced the notion of *causality* in memory consistency models and code transformations of parallel programs. A reasonable memory consistency model should not violate causality. This paper showed that the state-of-the-art memory models as the JMM and the recent proposal of the C++MM, violate causality. We also defined using the notion of causality, a Causal Memory Model (CMM) that is the weakest memory model that preserves causality. We identified specific code transformations that are causality preserving and those that are not. Our preliminary evaluation of load elimination transformation (which is shown to be a causality preserving transformation) in Jikes RVM results in a 37.94% reduction in dynamic memory load operations by using CMM as opposed to SC memory model and the execution time on a 16-core processor was reduced by 46.2%.

References

[1] Java Grande. http://www.epcc.ed.ac.uk/research/activities/java-grande/.

[2] Jikes RVM. http://jikesrvm.org/.

[3] Sarita Adve and Mark D. Hill. A unified formalization of four shared-memory models. *IEEE Transactions on Parallel and Distributed Systems*, 4:613–624, 1993.

[4] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29(12):66–76, 1996.

[5] Sarita V. Adve and Mark D. Hill. Weak ordering—a new definition. In *ISCA '90: Proceedings of the 17th annual international symposium on Computer Architecture*, pages 2–14, New York, NY, USA, 1990. ACM.

[6] Mustaque Ahamad, Phillip W. Hutto, Gil Neiger, James E. Burns, and Prince Kohli. Causal memory: Definitions, implementation and programming. Technical Report GIT-CC-93/55, Georgia Institute of Technology, 1994.

[7] Hans-J. Boehm and Sarita V. Adve. Foundations of the c++ concurrency memory model. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 68–78, New York, NY, USA, 2008. ACM.

[8] Greg Bronevetsky and Bronis R. de Supinski. Complete formal specification of the openmp memory model. *Int. J. Parallel Program.*, 35(4):335–392, 2007.

[9] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 519–538, New York, NY, USA, 2005. ACM.

[10] Guang R. Gao and Vivek Sarkar. Location consistency-a new memory model and cache consistency protocol. *IEEE Transactions on Computers*, 49(8):798–813, 2000.

[11] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *ISCA '90: Proceedings of the 17th annual international symposium on Computer Architecture*, pages 15–26, New York, NY, USA, 1990. ACM.

[12] K. Kennedy and J. R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[13] L. Lamport. How to make a multiprocessor that correctly executes multiprocess programs. *IEEE Trans. on Computers*, C-28(9):690–691, September 1979.

[14] Jaejin Lee, David A. Padua, and Samuel P. Midkiff. Basic compiler algorithms for parallel programs. *SIGPLAN Not.*, 34(8):1–12, 1999.

[15] D. H. Linder and J. C. Harden. Access graphs: A model for investigating memory consistency. *IEEE Trans. Parallel Distrib. Syst.*, 5(1):39–52, 1994.

[16] Jeremy Manson, William Pugh, and Sarita V. Adve. The java memory model. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 378–391, New York, NY, USA, 2005. ACM.

[17] W. Pugh. *Java Memory Model Causality Test Cases.* Technical Report, University of Maryland, 2004. http://www.cs.umd.edu/ pugh/java/memoryModel.

[18] Vijay A. Saraswat, Radha Jagadeesan, Maged Michael, and Christoph von Praun. A theory of memory models. In *PPoPP '07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 161–172, New York, NY, USA, 2007. ACM.

[19] Dennis Shasha and Marc Snir. Efficient and correct execution of parallel programs that share memory. *ACM Trans. Program. Lang. Syst.*, 10(2):282–312, 1988.

A Proof Sketch of Claim 5-1

Firstly we need to define single execution program (SEP) as follows: For a given execution e, suppose $ob(e) =< V, \xrightarrow{vo} >$, the single execution program $P = SEP(e)$ is a program which only contains output actions where each action outputs a constant. Moreover, let A be the set of actions in P. A is defined as follows: $A = \{output(value) | value \in V\}$ and $\forall output(value_1)$ $\forall output(value_2)$ $(output(value_1) \xrightarrow{po} output(value_2) \leftrightarrow value_1 \xrightarrow{vo} value_2)$.

For example, suppose $ob(e) =< \{1, 2\}, \{1 \xrightarrow{vo} 2\} >$, $SEP(e) =< \{a_1 = output(1), a_2 = output(2)\}, \{a_1 \xrightarrow{po} a_2\} >$. A single execution program $SEP(e)$ has two important properties:

(1) $E(SEP(e))$ only contains one execution and the execution preserves causality. The reason is that all actions in $SPE(e)$ do not access any variable. (2) Let e' be the execution in $E(SEP(e))$, then $ob(e') = ob(e)$.

Now we prove the claim 5-1. Note that it is equivalent to prove that if M violates causality, M allows at least one transformation which is not a CT. Let b denote observable behavior. When M violates causality, $\exists P \exists b(b \in M(P) \wedge \forall e(e \in E(P) \wedge ob(e) = b \rightarrow e$ violates causality)). Now let $Q = SEP(e)$ and let T be a transformation that $Q = T(P)$. T is subset correct under M because $M(Q) = \{b\} \subseteq M(P)$. However, T is not a CT for the following reason: For the execution $e_Q \in E(Q)$, $ob(e_Q) = b \wedge e_Q$ preserves causality, but $\forall e_P(e_P \in E(P) \wedge ob(e_P) = b \rightarrow e_P$ violates causality). So T does not satisfy the definition of CT. Therefore, M allows transformation T which is not a CT. ■

B Proof Sketch of Claim 5-2

The causality graph of any legal execution under CMM must have no cycle. So if T is a CT, any possible observable behavior of Q is also a possible observable behavior of P. Therefore, T is subset correct. ■