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

Establishing 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 092
January, 2010

Copyright © 2010 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

In this paper, we establish a notion of causality that should be used as a desideratum for memory models and code transformations of parallel programs. We introduce a Causal Acyclic Consistency (CAC) model which is weak enough to allow various useful code transformations, yet still strong enough to prevent any execution that exhibits "causal cycles" that may be caused by the Java Memory Model (JMM) [18].

For *memory models*, we introduce a graph model called *causality graph* that can be used to analyze if a particular program execution violates causality. By using *causality graph*, we show that a popular memory model (such as the Java memory model) can lead to program executions that exhibit causality violations with respect to our notion of causality.

For *code transformations*, we establish criteria to identify transformations that are *causality-preserving* which do not result in any execution that exhibits causality violation. We showed that the CAC model allows all the causality-preserving transformations. Finally, we present preliminary experimental results for a load elimination optimization to motivate the performance benefit of using the CAC model relative to the Sequential Consistency (SC) model which is the most basic memory model. For the benchmark program studied, the number of getfield operations performed was reduced by 37.9% by using the CAC 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 values of a memory location. There are fundamental challenges in specifying the memory model for a shared memory multiprocessor system because several threads of computation may access the same memory location in parallel, and 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 [15] 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 natural extension of uniprocessor memory consistency. However, it is well known that the requirement of serializability in SC restricts many common compiler and hardware optimizations [4]. Several relaxed memory models have been proposed to address these issues in the SC model, such as Weak Ordering model [5], Release Consistency model [14], and Location Consistency model [13].

1.1 A Desideratum

For race-free programs, most of the relaxed memory models guarantee the same result of an execution as if the execution was performed under the sequential consistency memory model. For program with races, some memory models, such as the C++ memory model [8], consider those programs as incorrect programs and leave the semantics undefined. However, there are still some important reasons to ensure some semantics for programs with races. For instance, (1) Java language rules out "out-of-thin-air" results for security and safety properties of the programming language [18], and (2) some algorithms do have data races and rely on the semantics of the memory model for data race programs (e.g. Dekker's algorithm of mutual exclusion [11]), and (3) data race is a part of life while debugging, where the restrictions on race cases can help debuggers.

The two common considerations about the semantics for race programs are *memory coherence* and *causality.* Memory coherence can be stated as follows [14]: "all writes to the same location are serialized in some order and are performed in that order with respect to any processor." However, Gao and Sarkar found that memory coherence may pose fundamental obstacles to defining a scalable and efficient view of memory consistency in computer systems [12]. They proposed the location consistency model [13] that is independent of memory coherence for data race programs. Modern popular memory models such as Java memory model [18] and OpenMP memory model [9] also independent of memory coherence for data race programs. The C++ memory model [8] only requires memory coherence for atomic operations. In this paper we will not further discuss memory coherence since our memory model is also independent of memory coherence.

The basic idea of *causality* is that the value of a read operation returned to a thread is the "effect" of some prior write that "caused" that value to be written to the location, and not due to some out-of-thin-air write or some write that may happen in the future. There are several variations of causality, for instance, Causal Consistency (CC) memory model uses "writes-into order" [6], and C++ memory model uses "write-to-read causality" [8]. These different variations of causality can lead to different interpretation of what value a read operation will return to a thread. Intuitively, we say that a memory model M_1 is "weaker than" another memory model M_2 when every read operation under M_1 can return more possible values than (or at least the same possible values as) the same read operation under M_2. Given that there are many variations of causality the question that we want to answer in this paper is whether there is a definition of causality that is the weakest one, and yet it is "consistent". By consistency we mean that a read operation does not return values out-of-thin-air or from a write operation that will be performed in future. It is important to notice that CC has an additional requirement that each thread should also satisfy its program order, which seems too strong to allow aggressive reordering optimizations. On the other hand, JMM only rules out out-of-thin-air results and as we will illustrate in this paper that it allows executions that can lead "causal cycles"[18]. It seems too weak because causal cycles will cause difficulties in understanding, programming and implementation of memory models. In this paper we present a new memory model called Causal Acyclic Consistency (CAC) model that is weak enough to allow various code transformations

and it does not allow any out-of-thin-air result or any execution that exhibits "causal cycles" (which we will describe in this paper).

1.2 Contributions of the Paper

The main contributions of the paper are as follows.

- We establish a notion of causality that should be used as a desideratum for memory models and code transformations of parallel programs. Based on our notion of causality, we introduce a Causal Acyclic Consistency (CAC) model which is weak enough to allow various useful code transformations, yet still strong enough to prevent any execution that exhibits causal cycles that may be caused by JMM. From the programmer's or hardware designer's point of view, CAC allows aggressive optimizations without violating causality.
- For *memory models*, we introduce a *causality graph* that can be used in conjunction with operational semantics for the parallel program to analyze if a particular program execution violates causality. We show that a popular memory model as the Java memory model (JMM [18]), can lead to program executions that exhibit causality violations with respect to our notion of causality.
- For *code transformations*, we establish criteria identify transformations that are *causality-preserving* which do not result in any execution that exhibits causality violation. We showed that CAC model allows all the causality-preserving transformations. Finally, we present preliminary experimental results for a load elimination optimization to motivate the performance benefit of using the CAC 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 CAC 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 an motivating example of causality and causality graph. Section 3 introduces the notion of causality and defines CAC. Section 4 demonstrates our analyses on the JMM 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 an example to motivate the notion of causality. We will use a graph model called the *causality graph* (CG) to illustrate the program execution that exhibits causal cycles.

A CG consists of nodes representing memory operations of a program execution and directed edges representing causality relations among the memory operations. Given a particular

execution of a program on a multiprocessor system that supports a particular memory model, we can use the CG to check if there exist cyclic causality relations among some of memory operations of the execution, i.e., some edges of the CG form a cycle. Since cyclic causality relations are impossible to happen in the real world, the execution violates causality in this case.

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: A program execution with the result of $r1 = r2 = r3 = 2$ will violate causality.

Consider the Java causality example from [18] shown in Figure 1, where a and b are shared variables, and $r1$, $r2$ and $r3$ are local variables. For this example, The question to address is whether there exists an execution with a result of $r1 = r2 = r3 = 2$ that does not violate causality.

To answer this question, we first assume that such an execution exists. Then we construct the causality graph of the execution. Finally, we will see that the causality graph contains cycles, which means that the execution violates causality. To construct the causality graph, we first create nodes corresponding to statements that produce the result. For each part of the result, we trace back to determine which statements caused the value in that part. For instance, we insert a causality relation edge from node 1 to node $r1 = 2$ because statement 1 is the only one that writes value to $r1$. Similarly we also insert other causality relation edges, as illustrated in Figure 2. Notice that the resulting causality graph contains cycles indicating that the execution with result $r1 = r2 = r3 = 2$ violates causality. Let us further analyze why the execution violates causality.

The steps that lead to causal cycles are as follows:

1. Since in the result $r1$ equals 2, statement 1 must get value of a from statement 6. Otherwise, $r1$ will equal 0 due to the initial value of a.

2. For a similar reason, statement 2 must get value of a from statement 6.

3. For a similar reason, statement 5 must get value of b from statement 4.

4. statement 6 must get value of $r3$ from statement 5 because $r3$ is a local variable which must satisfy semantics of sequential programs.

5. For a similar reason, statements 3 must get value of $r1$ from statement 1 and value of $r2$ from statement 2.

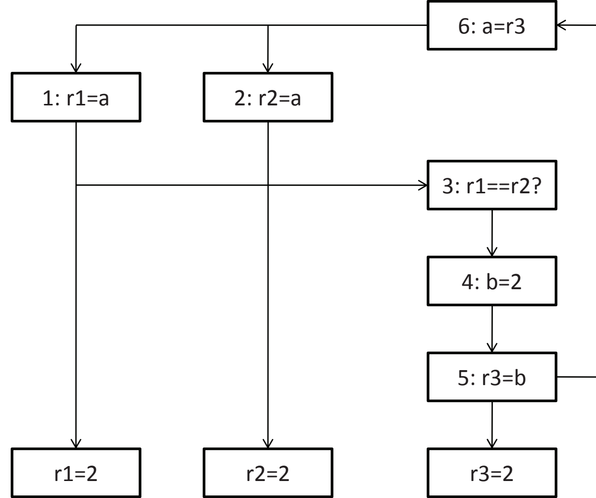

Figure 2: The causality graph corresponding to the example in Figure 1

6. There is a causality relation from statement 3 to statement 4 because the control statement 3 determines whether statement 4 will be executed.

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.

3 Causality and the Causal Acyclic Consistency Model

In this section, we propose our notion of causality. Based on the definition of causality, we define a Causal Acyclic Consistency (CAC) model as the weakest memory model that satisfies our notion of causality.

3.1 Causality Relation

To define our notion of causality, we firstly define causality relation. Given a program, the same line of the program may be instantiated (i.e., executed) many times during the execution. We call each of these instantiations an *action* (assuming one instruction per line in the program and each instruction accesses at most one shared variable). Given a program execution which is the trace of actions that are executed by hardware, a causality relation defines which actions are "causes" of actions that are "effects".

The same as JMM [18], we assume that program executions are "well-formed". Briefly, that means (1) Each read action should get value from a write action. (2) Synchronization actions are used correctly. For example, lock and unlock actions are correctly nested. We use synchronization order ($\stackrel{so}{\rightarrow}$) to represent the order to complete actions that is enforced by the

semantics of synchronization actions. For example, actions inside a lock region cannot complete unless the lock action owns the lock. (3) The actions performed by each thread are the same as that they are generated by that thread in program order in isolation, except that read actions on shared variables may get values from other threads, where the program order ($\stackrel{po}{\rightarrow}$) is the order imposed by the program text [3, 21]. (4) Let happens-before order [18] ($\stackrel{hb}{\rightarrow}$) be the transitive closure of program order and synchronization order. A read action r cannot get value from a write action w if $r \stackrel{hb}{\rightarrow} w$ or $w \stackrel{hb}{\rightarrow} w' \stackrel{hb}{\rightarrow} r$ where w' is another write action that accesses the same memory location as r and w.

Since a natural requirement of causality relation is that a "cause" should be completed earlier than the completion of any corresponding "effect", we use causal order ($\stackrel{co}{\rightarrow}$) to represent the causality relation among actions. If an action a_1 causes another action a_2, we say $a_1 \stackrel{co}{\rightarrow} a_2$ which means a_1 must be completed earlier than the completion of a_2. Our idea of establishing causal order is to have as less restrictions on action reordering as possible and still prevent out-of-thin-air results or causal cycles. Based on this idea, we require the causal order to satisfy the following requirements. (1) A read operation can only get value which was written by a prior write operation. (2) The order of accesses on shared variables should not be restricted by uniprocessor data dependencies on those shared variables when a read action is possible to get value from a write action in another thread. (3) Accesses on local variables should satisfy uniprocessor dependencies. (4) Synchronization order and the order enforced by fences [1] must be respected. (5) Causal order is transitive. Based on all those requirements, an action $a_1 \stackrel{co}{\rightarrow}$ another action a_2 if they satisfy one of the following conditions.

- a_1 writes a value to a variable and a_2 reads that value from the same variable. (based on requirements 1,2 and 3)
- a_2 is control dependent on a_1. (based on requirement 3)
- $a_1 \stackrel{so}{\rightarrow} a_2$. (based on requirement 4)
- There exists a local fence action a_3 where $a_1 \stackrel{po}{\rightarrow} a_3$ and $a_3 \stackrel{po}{\rightarrow} a_2$. (based on requirement 4)
- There exists an action a_3 where $a_1 \stackrel{co}{\rightarrow} a_3$ and $a_3 \stackrel{co}{\rightarrow} a_2$. (based on requirement 5)

3.2 Definitions of Causality and the CAC Model

Given a program execution, each "cause" action should always be completed before its corresponding "effect" actions. It is always possible to do so if there is no cyclic causal orders among the actions of the execution. So we say the execution preserves causality in this case. However, when there are cyclic causal orders among actions, it is impossible to complete the actions in

[1] For simplicity, we only consider a coarse grain fence which disallows any following action to start unless all the actions before the fence has been completed. However, there is no difficulty to extend the definition of causal order to support finer grain fences.

an order that each "cause" action is always completed before its corresponding "effect" actions. Therefore, we say the execution violates causality in this case. We say a memory model violates causality if it allows executions that exhibit causality violations. So we have the following definitions.

Definition 3-1: An execution violates causality if and only if a part of its causal orders is cyclic. An execution preserves causality if and only if it does not violate causality.

Definition 3-2: A memory model violates causality if and only if it allows some result r of some program p where no causality preserving execution of p can get the result r. A memory model preserves causality if and only if it does not violate causality.

Now we can define the Causal Acyclic Consistency (CAC) memory model. Since we expect it to be the weakest memory model that satisfies our notion of causality, it must allow and only allow the results of all the executions that preserve causality. So we have the following definitions.

Definition 3-3: A memory model is the causal acyclic consistency memory model if and only if for any given program, it allows and only allows the results of all executions of the program that preserve causality.

Definition 3-4: A hardware satisfies causal acyclic consistency if it only generates results that are allowed by the CAC model.

4 Causality Analysis of the Java Memory Model

In this section, we discuss the causality analysis of the Java memory model (JMM) [18]. JMM community has proposed a suite of 20 causality test cases [19] that can help compiler writers and Java Virtual Machine (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. We will show that ten of the JMM causality test cases which preserve the JMM's notion of causality do indeed violate our notion of causality. Moreover, in one case (i.e., case 12), it is suggested that the test case violates JMM's notion of causality, but interestingly the same test case does not violate our notion of causality.

4.1 An Overview of Analyses on the Java Causality Test Cases

In this section, we introduce the result of analyzing the Java Causality Test Cases. The summary is shown in Figure 3. 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 ("Y" means the test case violates our notion of causality). Finally, the label "Forbidden under JMM" means whether JMM disallows the test case ("Y" means JMM disallows the test case).

# 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
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 3: Summary of analyses on Java Causality Test Cases

The table shows that 10 of the 13 JMM's positive cases violate our notion of causality. Therefore, JMM violates our notion of causality. The process of analyzing those 10 cases is quite similar to the process that we used for analyzing the motivating example in Section 2, thus we omit them. The case 12 which violates JMM's notion of causality but satisfies CAC's notion of causality will be discussed in Section 4.2.

4.2 Java Causality Test Case 12

Java Causality Test Case 12 (as shown in Figure 4a) is a case that is disallowed by JMM. The argument presented by JMM is that the program in Figure 4a should be equivalent to the program in Figure 4b, where the result $r1 = r2 = r3 = 1$ in Figure 4b is out-of-thin-air. Therefore, JMM disallows the result in Figure 4a.

However, it is debatable that the program in Figure 4a and the program in Figure 4b are equivalent. Comparing the difference between the two programs, we see that the implicit data dependence between statements 2 and 3 in (a) are converted to the combination of the explicit data dependence between statement 1 and 3 and explicit control dependencies among statements 3, 4 and 5 in (b). Therefore, for the program in (b), statement 1 always indirectly causes statement 4 or 5. However, for the program in (a), statement 1 indirectly causes statement 3 only when r1 equals 0. Therefore, a may-happened causal relation in (a) becomes an always-happened causal relation in (b). (Note that statement 3 in (a) is equivalent to the combination of statements 4 and 5 in (b).) So we conclude that the two programs in Figure 4 are not equivalent. Therefore, the undesirable out-of-thin-air result of the program in (b) is not a direct consequence that the program in (a) should disallow the result.

Next we will show that the program in (a) can indeed get the result $r1 = r2 = r3 = 1$

Initially, x = y = 0; a[0] = 1; a[1] = 2;

Thread 1	Thread 2
1: r1 = x;	5: r3 = y;
2: a[r1]=0;	6: x = r3;
3: r2=a[0];	
4: y=r2;	

(a) Should the result $r1 = r2 = r3 = 1$ be allowed?

Initially, x = y = 0; a[0] = 1; a[1] = 2;

Thread 1	Thread 2
1: r1 = x;	7: r3 = y;
2: a[r1]=0;	8: x = r3;
3: if (r1==0)	
4: r2=0;	
5: else r2=1;	
6: y=r2;	

(b) The result $r1 = r2 = r3 = 1$ is out-of-thin-air.

Figure 4: Java causality test case-12. The result in (a) is disallowed by JMM because JMM considers the program in (a) and the program in (b) to be equivalent, where the result in (b) is out-of-thin-air.

without violating our notion of causality. Suppose the program is executed in the order of statements as 3,4,5,6,1,2 and get the result $r1 = r2 = r3 = 1$. Then statement 2 accesses $a[1]$ and statement 3 accesses $a[0]$, where these two statements have no dependence and the program execution does not cause any causal cycle. In conclusion, Java Causality Test Case 12 satisfies our notion of causality.

5 Causality and Program Transformations

In this section, we introduce causality preserving transformations which guarantee that the transformations do not result in any causality violating executions. Various code transformations are causality preserving transformations and dependence-breaking optimizations can be converted into causality preserving transformations by applying fences. We prove that the CAC model allows all causality preserving transformations, which indicates that the CAC model supports various code transformations. We also show that certain such causality preserving program transformations under CAC 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 of Parallel Programs

A legal transformation of a parallel program means that any result that the transformed program can generate can also be generated by the original program. Many past works, such as [16, 18, 7], use this definition, and we will use this criteria as the basis for understanding causality preserving transformations in the following section.

5.2 Causality Preserving Transformations

The intuition to define causality preserving transformation is that we want to guarantee that code transformations do not result in any execution that exhibits causality violation. It has been observed in [18] that some dependence-breaking optimizations potentially result in executions containing apparent "causal cycles". Therefore, our definition of causality preserving transformation is helpful for applying dependence-breaking optimizations as well as other optimizations without causing "out-of-thin-air" results or "causal cycles".

We requires a causality preserving transformation to guarantee that any result that can be generated by a causality preserving execution in the transformed program can also be generated by a causality preserving execution in the original program. Otherwise, suppose a result r can be generated by a causality preserving execution in the transformed program but cannot be generated by a causality preserving execution in the original program. Then it is risky to allow the result r of the transformed program under a memory model. The reason is that the original program should also allow r due to the definition of legal transformation, where only causality violating executions may generate r in the original program. So we define causality preserving transformation as follows.

Definition 5-1: A code transformation is a causality preserving transformation if and only if any result that can be generated by a causality preserving execution in the transformed program can also be generated by a causality preserving execution in the original program.

5.3 An Example of Causality Preserving Transformations

Figure 5 shows examples of causality preserving transformations and potentially causality violating transformations. The transformation that transforms the program in Figure 5a to the program in Figure 5b is a potentially causality violating transformation. The reason is as follows. Firstly, the program in Figure 5b must allow the result $r1 = r2 = 1$ because the result satisfies sequential consistency. Therefore, according to definition of legal transformation (Section 5.1), the program in Figure 5a should also allow $r1 = r2 = 1$. However, the program in Figure 5a is indeed the Java Causality Test Case 1. As we showed in Section 4.1, this case violates our notion of causality. Therefore, this transformation results in execution that exhibits causality violation for the program in Figure 5a. By using a local fence operation, Figure 5c shows a way to safely apply the dependence-breaking transformation without violating causality.

Initially, x=y=0;

Thread 1	Thread 2
1: r1 = x;	4: r2 = y;
2: if r1>=0	5: x = r2;
3: y=1;	

(a) Allowance of $r1 = r2 = 1$ will result in causality violating executions. So the result should be avoided.

Initially, x=y=0;

Thread 1	Thread 2
1: y=1;	3: r2 = y;
2: r1=x;	4: x = r2;

(b) $r1 = r2 = 1$ must be allowed because it satisfies sequential consistency. So the transformation from (a) to (b) may result in executions that exhibits causality violation.

Initially, x=y=0;

Thread 1	Thread 2
1: r1=x;	3: r2 = y;
2: fence;	4: x = r2;
3: y=1;	

(c) $r1 = r2 = 1$ is not allowed because "r1=x" is enforced to be completed before the completion of "y=1". So the transformation from (a) to (c) will not result in executions that exhibits causality violation.

Figure 5: Examples of causality preserving transformations and potentially causality violating transformations. (a) to (b) is a potentially causality violating transformation. (a) to (c) is a causality preserving transformation.

5.4 Legal Transformations Under CAC

In this section, we demonstrate that CAC allows various code transformations. Firstly, we will prove that all causality preserving transformations are legal under the CAC model. Then we show that various code transformations can be categorized into causality preserving transformations.

Theorem 5-1: Any causality preserving transformation is a legal transformation under the CAC model.

Proof: Suppose a causality preserving transformation T transforms a program P to a program Q. According to the definition of CAC (Definition 3-3 in Section 3.2), for any result r of Q that is allowed under CAC, r can be generated by a causality preserving execution in Q. Then according to the definition of causality preserving transformation (Definition 5-1 in Section 5.2), the result r can also be generated by a causality preserving execution in P. Then again, according to the definition of CAC, r of P is allowed under CAC. So according to the

definition of legal transformation in Section 5.1, T is a legal transformation under the CAC model. □

Now we demonstrate that the following three classes of code transformations are causality preserving transformations and thus are legal under the CAC model.

- If a transformation does not break any dependence, any synchronization order, or any order that is enforced by fence, the transformation is a causality preserving transformation. The reason is that a transformation that satisfies such restrictions will not remove any causal order from a causality graph, thus it does not transform any causality violating execution to a causality preserving execution.

  Many commonly used reordering transformations belong to this class. Note that the above restriction can be relaxed to allow breaking of write-after-read and write-after-write data dependencies because they do not indicate causality relations according to the definition of causal order in Section 3.1. Therefore, the code transformations that remove such dependencies (such as renaming) are also causality preserving transformations.

- A dependence-breaking transformation can always add fences to enforce the order of instructions that was enforced by dependencies in the original program. In this way, the transformation is a causality preserving transformation because the fences preserve the causal order that was indicated by the dependencies in the original program. One example is shown in Figure 5 of Section 5.3. Note that the transformed program may get more opportunities in reordering if fine-grain fences are used.

- Other cases that the transformation satisfies the definition of causality preserving transformations. Figure 6 shows one example of load elimination in this case. Note that the transformation in Figure 6 is disallowed under SC.

6 Preliminary Experimental Results

While the major contribution of our paper lies in the establishment 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 Causal Acyclic Consistency (CAC) model instead of the Sequential Consistency (SC) model. There are two reasons to compare the CAC model and the SC model. (1) SC is an ideal baseline because most of the memory models guarantee SC for data-race-free programs and have weaker semantics than SC for data race programs. (2) Each relaxed memory model has its own position to define its semantics. The performance issue is just one aspect of the tradeoff of a relaxed memory model. For example, if a memory model A is weaker than another memory model B, A always allows more transformations than B because a read operation under A may see more possible values than the same read operation under B but the programmability of A is not as good as B for the same reason.

Assume that the compiler does not know p==q.
Initially, p.x=q.y=0;

Thread 1	Thread 2	
1: r1=p.x;	4: p.x=1;	5: q.x=2;
2: r2=q.x;		
3: r3=p.x;		

(a) The original program. $r1 = r3 = 1, r2 = 2$ violates sequential consistency.

Assume that the compiler does not know p==q.
Initially, p.x=q.y=0

Thread 1	Thread 2	
1: r1=p.x;	4: p.x=1;	5: q.x=2;
2: r2=q.x;		
3: r3=r1;		

(b) The transformed program by load elimination on statement 3. $r1 = r3 = 1, r2 = 2$ satisfies sequential consistency.

Figure 6: An examples of load elimination that is a causality preserving transformation. It is not hard to examine that any result that can be generated by a causality preserving execution in (b) can also be generated by a causality preserving execution in (a). Note that the transformation is disallowed under SC because it introduces new result $r1 = r3 = 1, r2 = 2$.

In our experiments, for the CAC version, the transformation is applied according to Definition 5-1, where one example is shown in Figure 6. For the SC version, the transformation was performed by using *delay set analysis* [21] to arise opportunities for load elimination.

The JGF Moldyn benchmark was written using a subset of the `X10` language that includes of the `async`, `finish` and `isolated` parallel constructs [10]. 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 CAC and SC memory models.

All results were obtained using the `-Xmx2000M` JVM option to limit the heap size to 2GB,

# getfield (original)	# getfield after SC load elim.	# getfield after CAC load elim.	No load elim. exec. time in sec	SC load elim. exec. time in sec	CAC 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.

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 is shown in Table 1. We observe that the load elimination algorithm using the CAC 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.

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 prior write on 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 violates causally. 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 [15]. One drawback that is well known in the literature is that the SC model restricts many useful compiler and hardware optimizations. Subsequently relaxed models were proposed to allow certain kinds of compiler and hardware optimizations. The relaxed memory models guaranteed SC for data race free programs, but had very few guarantees for programs with races. There are several other works on memory models. Due to limited space we will not dwell into each one of them. In this section we discuss the models that focus on causality aspects.

In [21], 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 idea of the causality preserving transformation is influenced by delay-set analysis.

Ahamad et al. defined causal consistency in [6]. In their work, they defined a causality order which is the transitive closure of writes-into order and program order. Due to the need to fully respect program order when performing optimizations, their model may restrict on the kinds of optimizations that can be performed under our CAC model. In [17], Linder and Harden defined the "access 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 [18]. The main purpose of their definition of causality is to allow as many compiler transformations as possible and still prevent out-of-thin-air result for data race programs. However, it is observed that JMM allows executions that appear "causal cycles" (e.g. Figure 5 in [18]). This might violate the programmers' intention of the program thus we feel that the semantics of their causality is not very intuitive from programmers' perspective.

In [20], 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 result 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 [8]) which they claim to be violating causality. However, they do not explicitly define the notion of causality in their work. Since all of the three examples violate memory coherence [2]. A reasonable conjecture is that their idea of causality requires that memory coherence must be satisfied. It seems that it is too expensive to implement memory coherence in large scale multi-core or many-core architectures.

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 model as the JMM , violates causality. We also defined using the notion of causality, a Causal Acyclic Consistency (CAC) memory model that is the weakest memory model that preserves our notion of causality.

We identified specific code transformations that are causality preserving. 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 CAC as opposed to SC memory model and the execution time on a 16-core processor is reduced by 46.2%.

[2] In their work[8], the term "write atomicity" is used to represent the same meaning as memory coherence.

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] Rajkishore Barik and Vivek Sarkar. Interprocedural load elimination for dynamic optimization of parallel programs. In *PACT '09: Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, pages 41–52, Washington, DC, USA, 2009. IEEE Computer Society.

[8] 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.

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

[10] 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.

[11] Edsger W. Dijkstra. Cooperating sequential processes. pages 65–138, 2002.

[12] Guang R. Gao and Vivek Sarkar. On the importance of an end-to-end view of memory consistency in future computer systems. In *ISHPC '97: Proceedings of the International Symposium on High Performance Computing*, pages 30–41, London, UK, 1997. Springer-Verlag.

[13] 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.

[14] 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.

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

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

[17] 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.

[18] 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.

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

[20] 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.

[21] 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.