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

Order Free Consistency: Towards a Fully Asynchronous Memory Model

Chen Chen
Joseph B Manzano
Wenguang Chen
Guang R. Gao

CAPSL Technical Memo 080
November, 2007

Copyright © 2007 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

Computer architects are now studying a new generation of multi-core chip architectures that may integrate hundreds of processing cores and memory banks on a single chip - employing a shared memory organization. A system may consist of many such chips (nodes) and an increasing demand to support high bandwidth and shared-address space between nodes. Furthermore, the inter-chip and intra-chip interconnections are also fast progressing - with optical inter-chip and photonics intra-chip technology - promising unprecedented bandwidth as well as multi-channel reordering capabilities. This paper focuses on the following fundamental question: can we have a memory model that is truly *asynchronous* that is: (1) memory operations can be issued freely from the processors without blocking by any memory based data dependence, and (2) the memory transmissions can travel through the interconnection network freely without worrying that they may arrive at the destination out of order. Furthermore, such a memory model must be *realizable*: that is, we can define an operational model [1] (hence construct an abstract machine) that can fully explore the above features during program execution.

[1]The terms “operational model” and “operational semantics” can be exchangeable in this paper.

Contents

1 Introduction **1**

2 Problem Formulation **2**
2.1 The four properties that defines our memory model . . . 2
2.2 The role of an operational model in defining a realizable memory model . . . 6
2.3 Problem formulation . . . 7

3 The Order Free Consistency (OFC) Model **7**
3.1 The OFC program model . . . 7
3.2 The OFC abstract machine architecture model . . . 8
3.3 Handling synchronization operations . . . 11

4 Properties of The OFC Model **15**
4.1 Causal ordering property . . . 15
4.2 Equivalence property . . . 16
4.3 Monotonicity . . . 17
4.4 Non-intrusive reads . . . 17

5 Benefits of the OFC Model **17**
5.1 Causal ordering increases parallelism . . . 18
5.2 Non-speculation based lock-free mutual exclusion . . . 18
5.3 Ability to explore the bandwidth provided by multi-channel networks . . . 18
5.4 Opening up new opportunities for compiler / runtime optimizations . . . 19

6 Related Work **19**

7 Conclusion and Future Work **20**

List of Figures

1 A Reasonable Violation of Flow-dependence . . . 3
2 A Reasonable Violation of Anti-dependence . . . 3
3 A Reasonable Violation of Output-dependence . . . 4
4 A Pair of Examples Which Violates Monotonicity . . . 5
5 The OFC abstract machine architecture model . . . 8
6 The pomsets of location x at each step for the program in Figure 2 . . . 10
7 An example with synchronization operations . . . 13
8 The pomsets of location x at each step for the program in Figure 7 . . . 14

List of Tables

1 Introduction

Emerging future microprocessor chip technology unveils a new generation of many-core chip architectures that may contain 100 to 1,000 processing cores using a shared memory organization with a large number of on chip memory banks. A system may consist many such chips (nodes), an increasing demand for higher bandwidth and support for a shared-address space between nodes. Furthermore, the inter-chip and intra-chip interconnections are also fast progressing - with optical inter-chip and photonics intra-chip technology - promising unprecedented bandwidth as well as multi-channel reordering capabilities[4, 10]. In fact, under so-called "wavelength division multiplexing" (WDM) over a single physical channel one can have multiple communication/data channels by using different wavelengths for each!

However, some existing popular memory models place considerable constraints in reordering memory operations; both on their issuing order from the processors as well as their traveling order through the interconnection network. For instance, in the definition of the popular *release-consistency* (RC) model [8], as in most other SC-derived memory models[2] which are defined in [7], an implicit assumption is that uniprocessor data dependences should be respected. This assumption is rarely questioned, and has also been used in the definition of the *location consistency* (LC) model [7]. As a results, a compiler and architecture optimization must not violate the dependence ordering on a per process(or) basis. When working in a multiprocessor environment, the restrictions imposed by the above constraints (e.g. data dependencies) should be relaxed to allow more reordering opportunities - hence generating new opportunities for parallelism. We will illustrate this point in Section 2.1.

This will allow the most efficient use of new technologies (such as optical inter-chip and photonics intra-chip interconnects) technologies. Such technologies provide multiple channels along each path between each pair of processors and memory banks. There are ample opportunities for subsequent memory transmissions taking different channels on the same path (thus might arrive out of order) – to fully utilize the available bandwidth. In other words, enforcing such uniprocessor dependence ordering constraints (when it is not necessary) may causes several optimization opportunities to be lost and they are detrimental to the application performance.

This paper focuses on the following fundamental questions: Can we have a memory model that is truly *asynchronous*, that is: (1) Memory operations can be issued freely from the processors without blocking by any memory based data dependence, and (2) The memory transmissions can travel through the interconnection network freely without worrying that they may arrive at the destination out of order. Furthermore, such a memory model must be *realizable*: that is we can define an operational model (hence construct an abstract machine) that can fully explore the above features during program execution.

In this paper, we propose the Order Free Consistency (OFC) model to address the problem discussed above. The rest of the paper is organized as follows. Section 2 provides the problem formulation and the properties of a sound memory model. Section 3 introduces the OFC model.

[2]Memory Models which have the underlying assumption of memory coherence

Section 4 states the properties of the OFC model. Section 5 shows the benefits of the OFC model. Section 6 discusses the related work. The conclusions and future work are presented in Section 7.

2 Problem Formulation

This section presents the main challenges that this paper addresses. Moreover, the criterion for a good solution methodology is outlined here.

2.1 The four properties that defines our memory model

This section discusses the four desirable properties that defines our memory model. We believe that these properties are also the criteria for defining a reasonable and efficient memory model.

Property 1: Causal Ordering In a correct program execution, the instructions should be performed in an order which is consistent to the causal order – i.e. if a load is "caused" (returns a value that is written) by a store, the store must be performed before the load is performed. In a parallel program, a load may have more than one read values, as long as, the values are written by stores that such a load depends on (also called the "candidate stores" of this load). However, in an execution of a parallel program, only one "candidate store" causes a load. The causal ordering property only requires that this causal order is respected.

Nevertheless, most memory models applies extra restrictions that are not necessary under the causal order. For example, one of the necessary properties used widely in defining memory consistency models (in particular, SC and SC-derived models such as Release Consistency) is that data dependence (on a uni-processor level) must be respected [8]. This means that if there is data dependence between two instructions from the same processor, they cannot be reordered when they are performed. Although this does not violate the causal ordering, it may preclude some program executions that also satisfy casual ordering - as it will be explained by the following examples.

In Figure 1, a program is shown with two concurrent threads - namely T1 and T2. Both threads operate on a shared variable x which is initialized to zero before the program starts. Under SC and SC-derived models, the result {r1=2, r2=1} is illegal since it violates flow-dependence [3].

In contrast to the SC and SC-derived memory consistency models, we believe that the ordering constraints due to flow-dependence in this example are not always necessary and can be further relaxed. To illustrate our view, let us consider the following program execution.

The instructions are performed in the order 1, 4, 3, 2. (The order of all the instructions that are performed is called the "execution order" in the rest of this section). This execution

[3] Flow-dependence is also called true-dependence.

Initially x = 0

T1	T2
1: x = 1	3: x = 2
2: r1 = x	4: r2 = x

The result {r1 = 2, r2 = 1} violates flow-dependence under SC and SC-derived models. However it satisfies causal ordering.

Under SC and SC-derived models, the result {r1 = 2, r2 = 1} is impossible. Such a result implies that instruction 3 came before instruction 2 (because r1 = 2, causal ordering); instruction 1 came before instruction 4 (because r2 = 1, causal ordering); instruction 1 came before instruction 2 (due to uniprocessor flow-dependence); instruction 3 came before instruction 4 (due to uniprocessor flow-dependence), and so both instruction 1 and 2 came before both instruction 3 and 4. Thus it leads to either the result {r1 = r2 = 1} or the result {r1 = r2 = 2}, but not the result {r1 = 2, r2 = 1}. However the result {r1 = 2, r2 = 1} can be produced without violating causal ordering. Since instruction 1 and 3 are store candidates for instruction 2 and 4, the instructions can be performed in the order 1, 4, 3, 2, which satisfies causal ordering and produces the result {r1 = 2, r2 = 1}.

Figure 1: A Reasonable Violation of Flow-dependence

does not violate causal ordering. Moreover, it produces the result {r1=2, r2=1}. The reason is that both instructions 1 and 3 are store candidates for instruction 2. Thus, either of them can cause instruction 2. Similarly, either of them can cause instruction 4. Thus, the order 1, 4, 3, 2, in which 1 causes 4 and 3 causes 2, satisfies casual ordering even thought it violates flow-dependence.

Initially x = 0

T1	T2
1: r1 = x	3: r2 = x
2: x = 1	4: x = 2

The result {r1 = 2, r2 = 1} violates anti-dependence under SC and SC-derived models. However it satisfies causal ordering.

Under SC and SC-derived models, the result {r1 = 2, r2 = 1} is impossible. Such a result implies that instruction 4 came before instruction 1 (because r1 = 2, causal ordering); instruction 2 came before instruction 3 (because r2 = 1, causal ordering); instruction 1 came before instruction 2 (due to uniprocessor anti-dependence); instruction 3 came before instruction 4 (due to uniprocessor anti-dependence), and so instruction 1 came before itself. Such a cycle is prohibited by sequential consistency. However the result {r1 = 2, r2 = 1} can be produced without violating causal ordering. Since instruction 2, 4 are a store candidate for instruction 3, 1 respectively, the instructions can be performed in the order 2, 3, 4, 1, which satisfies causal ordering and produces the result {r1 = 2, r2 = 1}.

Figure 2: A Reasonable Violation of Anti-dependence

Figure 2 presents another example [4] which shares same initial conditions with the example in Figure 1 (Two threads working on a shared variable which is initialized to zero). Under SC and SC-derived models, the result {r1=2, r2=1} is also illegal as it violates anti-dependence. As with the example in Figure 1, we believe that the ordering constraints presented by this example are not always necessary and can be further relaxed. Consider the execution order 2,3,4,1 which produces the result {r1=2, r2=1}. Although it violates anti-dependence, it satisfies casual ordering. The reason is that instructions 2 and 4 are store candidates for instructions 3 and 1, respectively.

Initially x = 0

T1	T2
1: x = 1	3: r1 = x
2: x = 2	4: fence
	5: r2 = x

The result {r1 = 2, r2 = 1} violates output-dependence under SC and SC-derived models. However it satisfies causal ordering.

Under SC and SC-derived models, the result {r1 = 2, r2 = 1} is impossible. Such a result implies that instruction 2 came before instruction 3 (because r1 = 2, causal ordering); instruction 1 came before instruction 5 (because r2 = 1, causal ordering); instruction 1 came before instruction 2 (due to uniprocessor output-dependence); instruction 3 came before instruction 5 (due to the semantic of fence), and so instruction 1 came before instruction 2; instruction 2 came before instruction 3 and 5s. Thus it leads to the result {r1 = r2 = 2}, not the result {r1 = 2, r2 = 1}. However the result {r1 = 2, r2 = 1} can be produced without violating causal ordering. Since instruction 1 and 2 are store candidates for instruction 3 and 5, the instructions can be performed in the order 2, 3, 1, 4, 5, which satisfies causal ordering and produces the result {r1 = 2, r2 = 1}.

Figure 3: A Reasonable Violation of Output-dependence

Finally, in Figure 3, we present a program with a similar setup to our other two examples in Figure 1 and 2. However, in this case, this program exhibits ordering constraints due to output-dependence are not always necessary and can be further relaxed. As before, the result {r1=2, r2=1} is illegal under SC and SC-derived models since it violates the output-dependence. However, it can be produced by the execution order 2,3,1,4,5 which violates output-dependence but satisfies causal ordering.

Property 2: Equivalence Property For parallel programs which are data-race-free [1] or properly labeled [8], the model should be equivalent to the Sequential Consistency model. In Section 4.2, we will discuss the details.

Property 3: Monotonicity The model should be monotonic with respect to parallelism – if the memory model permits a certain mapping of values to dynamic instances of read instructions

[4] The example is one of the Java Causality Test Cases[15]

in the execution of a parallel program, it must permit the same mapping in an isomorphic legal execution of a more parallel version of the same program.[5, 7]

In [13], some examples are used to explain the properties and features of the Java memory model. Among those examples we found that two of them violate the monotonicity property. To better explain this property, we duplicate these two examples and put them in Figure 4. (The original version of the two examples are shown in Figure 11 and 12 in [13].)

Initially, x == y == 0

Thread 1	Thread 2	Thread 3
1: r1 = x	4: r2 = x	6: r3 = y
2: if (r1 == 0)	5: y = r2	7: x = r3
3: x = 1		

Must not allow r1 == r2 == r3 == 1

(a)

Initially, x == y == 0

Thread 1	Thread 2
1: r1 = x	6: r3 = y
2: if (r1 == 0)	7: x = r3
3: x = 1	
4: r2 = x	
5: y = r2	

Compiler transformations can result
in r1 == r2 == r3 ==1

(b)

(a) The example which is duplicated from Figure 11 in [13]. (b) The example which is duplicated from Figure 12 in [13]. The example in (a) can be considered as a more parallel version of the example in (b). The reason is that the former example is obtained by partitioning a sequential thread (Thread1 in (b)) from the latter example into two parallel threads (Thread1 and Thread2 in (a)). However the result {r1=r2=r3=1} is allowed in (b) but prohibited in (a). Thus they violate monotonicity.

Figure 4: A Pair of Examples Which Violates Monotonicity

In figure 4 (a), a program is shown with three concurrent threads - namely Thread1, Thread2 and Thread3. The threads operate on shared variables x and y which are initialized to zero before the program starts. Under Java memory model, the result {r1=r2=r3=1} is illegal since it displays an unacceptable "bait-and-switch" circular reasoning [13].

In figure 4 (b), another program is shown with similar setup. The difference is that two concurrent threads are used. Under Java memory model, the result {r1=r2=r3=1} is allowed since it can result from well understood and reasonable compiler transformations. The details are explained in [13]

The reasons why these examples violates monotonicity are explained below. Both of these examples consist of the same stream of instructions which are labeled from 1 to 7. In the former example, instruction 1 to 5 are distributed – i.e. instruction 1 to 3 are in Thread1,

and instruction 4 and 5 are in Thread2. However, in the latter example instruction 1 to 5 are centralized in Thread1. Thus, it can be deduced that the former example is obtained by partitioning a sequential thread (Thread1 in Figure 4 (b)) from the latter example into two parallel threads (Thread1 and Thread2 in Figure 4 (a)). As it is defined in [7], the former example is a more parallel version of the latter example. So, the monotonicity property requires that any result which is allowed in the latter example should also be allowed in the former example. However, these two examples violate the requirement since the result {r1=r2=r3=1} is allowed in the latter example but prohibited in the former one.

We found that the Relaxed Atomic + Ordering (RAO) model [16] uses a very similar example and it allows the same result ({r1=r2=r3=1} as the example shown in Figure 4 (b) [5]. However, it does not show a similar example as that in Figure 4 (a). Nevertheless, it seems that a reasonable memory model should not allow the result {r1=r2=r3=1} for the example shown in Figure 4 (a). Thus, we believe that the RAO model also violates the monotonicity property.

Property 4: Non-intrusive Reads Reads should be non-intrusive in the model – the addition or removal of a read instruction in a parallel program cannot change the legality of values returned by dynamic instances of other read instructions in a given execution of the parallel program. [7]

This property is called "classical" in [5]. Moreover, that paper shows an example of memory models which violate this "classical" property in Section 3.5.

2.2 The role of an operational model in defining a realizable memory model

This paper wishes to emphasize the role of an operational model in defining a realizable memory model with the set of desirable properties. An example illustrating the role of it is presented in Figure 2.

In this example, the reader may recall that the result {r1=2,r2=1} should be legal due to the causal ordering property: i.e. the instructions can be performed in the order of 2,3,4,1 which will produce this result. To the best of our knowledge, no other memory models except the Java memory model[13] and the Relaxed Atomic + Ordering (RAO) model[16] allow this result. However, neither of the two memory models has a specification of an abstract machine model as a part of its definition. In other words, under such models only the legal pairings of input and output sequences are used to define memory consistency. They do not provide a specification on how a real parallel machine can execute the example program to produce this result. In Section 3, we will introduce an operational model that plays a critical role in defining our memory consistency model. We will also illustrate how the above example is executed under our model and how it produces the desired result. An operational model specification and its

[5]Although the name of variables are mismatched

memory consistency model serve as a contract between hardware/architecture and software, and can become a useful hint for both compiler and architecture design practitioners.

2.3 Problem formulation

Based on the above discussion, the problem presented here can be formulated as:

Open Problem: How to define an operational model that implements a reasonable and efficient memory model with all desirable properties as listed in Section 2.1 ?

There may be more than one way to define an operational model that can provide a solution to the above problem. However, beginning in Section 3 we will introduce our proposed solution to this problem. We will also show that some benefits are to be gained from our memory model in Section 5.

3 The Order Free Consistency (OFC) Model

This section presents our memory model, named the Order Free Consistency (OFC) Model, which represents our solution to the open problem illustrated in Session 2.3. In Section 3.1, we outline the program model used for the OFC model. Then, in Section 3.2 we present the basis of our corresponding operational model that is used to define our memory model. Finally, in Section 3.3, we present the extension of our operational model for handling with synchronization operations.

3.1 The OFC program model

In this section, we outline the program model assumed in this paper. We follow a similar style as used by Gao and Sarkar in their work on the Location Consistency model [6].

- **Memory Write:** If processor P_i needs to write value v in location L, it performs a $write(P_i, v, L)$ operation, which we also represent by the notation $L = v$ in processor P_i's instruction sequence.
- **Memory Read:** If processor P_i needs to read a value from location L, it performs a $read(P_i, L)$ operation, which we also represented by the notation *read* L in processor P_i's instruction sequence.
- **Signal-wait synchronization:** If processor P_2 needs to wait for processor P_1, the synchronization is accomplished by P_1 performing a $signal(P_2)$ operation and by P_2 performing a corresponding $wait(P_1)$ operation.

- **Sync (composite) synchronization:** if processor $P_1, ..., P_k$ all need to synchronize among each other, the synchronization is accomplished by each processor performing a $sync(\{P_1, ..., P_k\})$ operation. A sync operation performed on the entire set of processors is equivalent to a barrier synchronization. In our memory model the $sync(\{P_1, ..., P_k\})$ operation is not an atomic operation. Instead, it is composed by performing a serial of signal-wait instructions, $signal(P_1)$; ...; $signal(P_k)$; $wait(P_1)$; ...; $wait(P_k)$, in each processor. Therefore, in Section 3.3, we don't discuss the sync operation.
- **Acquire-Release synchronization:** If processor P_i needs exclusive access to a shared location L, the synchronization is accomplished by performing an $acquire(P_i, L)$ operation at the start of the critical section and by a $release(P_i, L)$ operation at the end. In general, a processor may use acquire-release to request exclusive access to a set of shared variables, $(L_1, ..., L_m)$, rather than a single memory location. It should be noted that it is prohibited to use the signal-wait synchronizations inside the acquire-release pairs.

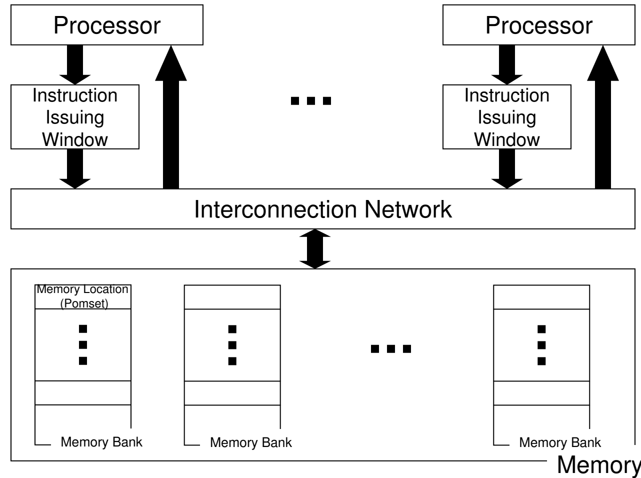


Figure 5: The OFC abstract machine architecture model

3.2 The OFC abstract machine architecture model

Our OFC operational model can be described by using a diagram for its underlying abstract machine architecture model as shown in Figure 5. The abstract machine has a number of processors and a shared memory connected by an interconnection network. Without loss of generality, in the figure, the memory is placed on the other side of the network. Furthermore, the memory itself may be consisted of many memory banks.

Now, let us describe how a (parallel) program is executed on this machine. Each processor proceeds to execute its portion of the parallel program. Memory operations (e.g. load/store) that are ready to be send to memory will be placed in the corresponding instruction issuing window. In addition, each ready instruction carries a tag that denotes the order the instruction

is fetched from the memory (i.e. the *fetch order*). Conceptually, all ready instructions from the same processor are totally ordered by their corresponding tags. However their issuing from the processor to the network, or their arrival to the memory, and their final completion can all be out of order - which will be further explained below.

It is important to note that there are several reasons - as listed below- that may cause the out-of-order effects mentioned above.

- **Instructions issuing:** This may become out-of-order from their fetch order.
- **Instructions transmission**: This may become out-of-order due network congestion.
- **Instructions arrival:** This may become out-of-order due to the existence of multi channels between the same source-destination port pairs.
- **Instruction completion:** This may become out-of-order due to (last minute) reordering opportunities by our abstract machine model.

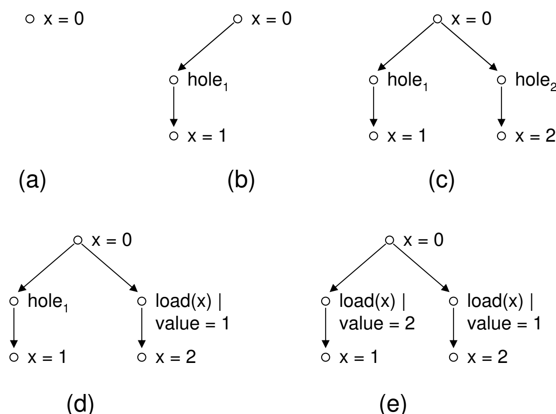
It is time to explain how memory operations are executed at the memory side. The instructions that have arrived at the output port(s) of the interconnection network will be placed in their corresponding memory locations. Each memory location is a partially ordered multiset called a pomset [7]. Each node in a pomset corresponds to a potential memory operation, while arcs between two nodes represent the direct ordering implied by their corresponding tags. It is important to note that in a pomset there may be some nodes that may denote memory operations that are to arrive in the future (we call them “holes” - a word derived from earlier data-driven architectures [18]). In this paper, we say that node A “precedes” node B if A and B access the same memory location and A’s tag is less than B’s tag. This means node A is less than node B in the partial order defined by their pomset.

A memory operation in the pomset can be performed if one of the following conditions is satisfied.

- Intuitively, a load operation can be performed if it is certain that there exists a store in the same pomset which writes the value that the load is eligible to read. More precisely, a load L is eligible to read a value which is written by a store S if and only if neither of the following conditions are satisfied.
  1. L precedes S in the pomset.
  2. S precedes L in the pomset and there exists a store (or a hole) S' which precedes L and succeeds S in the pomset.
- A store operation can always be performed since the pomset keeps all the arrived stores so that the stores never overwrite each other.

To help readers digest the operational model, we illustrate it by using an example which was already introduced in Figure 2. First, consider that at the processor side the following scenario happens.

- In processor P_1, instruction 1 is fetched, placed in the instruction issuing window and sent to the interconnection network. Then, instruction 2 is fetched, placed and sent.
- At the same time, in processor P_2, the same happens to instruction 3 and 4.



(a) Initial pomset. (b) The pomset after the arrival of instruction 2 (x = 1). (c) The pomset after the arrival of instruction 4 (x = 2). (d) The pomset after the arrival of instruction 3 (load x). The load returns value 1. (e) the pomset after the arrival of instruction 1 (load x). The load returns value 2.

Figure 6: The pomsets of location x at each step for the program in Figure 2

Then, consider at the memory side the following scenario happens step by step. The corresponding pomsets of memory location x in each step are shown in Figure 6

- **Initially:** The pomset of memory location x is initialized to contain a single node x = 0.
- **Step 1:** Instruction 2 (a store) arrives at the memory location. The other instructions have not arrived yet.

  Result: Instruction 2 is performed. And a hole (*hole1*) which precedes instruction 2 is generated.

  Explanation: According to the tag of instruction 2 it is known that there exists an instruction which is fetched earlier than instruction 2. However, it has not arrived yet. Therefore, we put a corresponding hole in the pomset.

- **Step 2:** Instruction 4 (a store) has arrived at the memory location.

  Result: Instruction 4 is performed and a hole (*hole2*) which precedes instruction 4 is generated.

  Explanation: The similar reason as step 1.

- **Step 3:** Instruction 3 (a load) has arrived at the memory location.

  Result: Instruction 3 replaces the *hole2*. Then it performs and reads the value which is written by instruction 2.

  Explanation: According to the tag of instruction 3 it is known that *hole2* is the corresponding hole to instruction 3. So instruction 3 replaces *hole2*. And in the current pomset, instruction 3 is eligible to read either the initial value or the value which is written by instruction 2. We assume that instruction 3 reads the value which is written by instruction 2.

- **Step 4:** Instruction 1 (a load) has arrived at the memory location.

  Result: Instruction 1 replaces the *hole1*. Then it performs and reads the value which is written by instruction 4.

  Explanation: The similar reason as step 3. We assume that instruction 1 reads the value which is written by instruction 4.

3.3 Handling synchronization operations

Now, we explain how we handle synchronization operations introduced earlier in our program model. We need to handle the new complexity introduced for programs that use synchronization operations.

First, the representation of global ordering of operations (from different processors) need to go beyond the tags used so far - that is: due to the ordering introduced by inter-processor synchronization, we need to introduce an additional dimension. Conceptually, for those who are familiar with terminology from distributed systems, we need to have something quite similar to vector clocks [12, 14].

Second, we need to handle the general out-of-order conditions as outlined earlier. Here further complexity arises - that a global time of a "hole" may not be known at a particular time. However, we can deduce the range of the time - hence the time is represented by such a range of possible time values.

Now, we describe one possible implementation of the global clock that will be used to handle the synchronization operations.

A vector clock is a vector of integers where the size of the vector is equal to the number of processors. We say clock C_i is smaller (or larger) than clock C_j if each element of C_i is numerically not larger (or not smaller) than the corresponding element of C_j, and at least one

element of C_i is numerically smaller (or larger) than the corresponding element of C_j. C_i is unrelated to C_j if it is neither smaller nor larger. In our memory model the minimum clock is denoted as $C_0 = (0, ..., 0)$ which is assigned to the initial node at each pomset and the maximum clock is denoted as $C_\infty = \{\infty, ..., \infty\}$.

A clock range is a two-tuples (C_L, C_U) where C_U is greater than C_L. We say clock range (C_{iL}, C_{iU}) is smaller (or larger) than clock range (C_{jL}, C_{jU}) if C_{iU} is smaller than C_{jL} (or C_{iL} is larger than C_{jU}). (C_{iL}, C_{iU}) is unrelated to (C_{jL}, C_{jU}) if it is neither smaller nor larger.

Then we define three functions on the clocks.

- Increase function: The increase function INC(C = $(c_1, ..., c_n)$, P_i) generates a new clock INC(C, P_i) = $(c_1, ..., c_{i-1}, c_i + 1, c_{i+1}, ..., c_n)$. The only exception is that INC(C_∞, P_i) = C_∞.
- Meet function: The meet function MEET($C_i = (c_{i1}, ..., c_{in})$, $C_j = (c_{j1}, ..., c_{jn})$) generates a new clock MEET(C_i, C_j) = (min$\{c_{i1}, c_{j1}\}$, ..., min$\{c_{in}, c_{jn}\}$).
- Join function: The join function JOIN($C_i = (c_{i1}, ..., c_{in})$, $C_j = (c_{j1}, ..., c_{jn})$) generates a new clock JOIN(C_i, C_j) = (max$\{c_{i1}, c_{j1}\}$, ..., max$\{c_{in}, c_{jn}\}$).

Now it is time to describe the method for calculating the clock range of each node in the pomset. For handling the signal-wait operations, we add two types of nodes to the pomset, i.e. signal and wait nodes. Similarly for handling acquire-release operations, we add acquire and release nodes.

We calculate the clock range of nodes in the following way.

- **Loads, Stores and Signals:** For such types of node N which is sent from processor Pi, let N_{prev} be the node which is fetched just before N in Pi (if N is the first operation in Pi, N_{prev} is the initial node). And the clock range of N_{prev} is (C_{prevL}, C_{prevU}). Then the clock range of N is (INC(C_{prevL}, Pi), INC(C_{prevU}, Pi)).
- **Wait:** For a wait node N which is sent from processor Pi, we define N_{prev} in the same way and let N_{sig} be the corresponding signal node. We calculate the clock range of N in the following way. Suppose the clock range of N_{prev} is (C_{prevL}, C_{prevU}). And the clock range of N_{sig} is (C_{sigL}, C_{sigU}). Then the clock range of N is (JOIN(INC(C_{prevL}, Pi), C_{sigL}), JOIN(INC(C_{prevU}, Pi), C_{sigU})).
- **Acquires and Releases:** For such types of node N, we calculate the clock range of N in the same way as Load, Store and Signal nodes. Then, we repeat the following steps if there exist an acquire node and a release node where their clock ranges are unrelated:

  Step 1: Modify the clock range of the acquire node as if the acquire node is a wait node and the release node is the corresponding signal node.

  Step 2: Modify the clock range of other nodes whose clock ranges are affected by the modification of the acquire node.

- **Holes:** A hole node N may be either a wait node or a non-wait node. Moreover, if it is a wait node, it may correspond to signal nodes in some processors. All of these possibilities are depends on the information which is carried by the instructions, i.e. the fetch order, the number of signals instructions which are fetched before, etc.

  We calculate the clock range of a hole in the following way. Firstly we consider all possible types of node N and get a serial of time ranges: (C_{1L}, C_{1U}), ..., (C_{kL}, C_{kU}). Then the clock range of N is (MEET(C_{1L}, ..., C_{kL}), JOIN(C_{1U}, ..., C_{kU})).

Finally for readers' further understanding of the operational model, we illustrate it by using an example which was shown in Figure 7.

Initially x = 0

T1	T2
1: x = 1	4: x = 2
2: sync({T1,T2})	5: sync({T1,T2})
3: r1 = x	6: r2 = x

(a)

Initially x = 0

T1	T2
1: x = 1	5: x = 2
2: signal(T2)	6: signal(T1)
3: wait(T2)	7: wait(T1)
4: r1 = x	8: r2 = x

(b)

(a) A program with sync operations. (b) The corresponding program with signal / wait operations. Sync({T1,T2}) is replaced by $signal(T_1)$; $signal(T_2)$; $wait(T_1)$; $wait(T_2)$. We omit $signal(T_1)$; $wait(T_1)$ in T1 and $signal(T_2)$; $wait(T_2)$ in T2 since they are redundant.

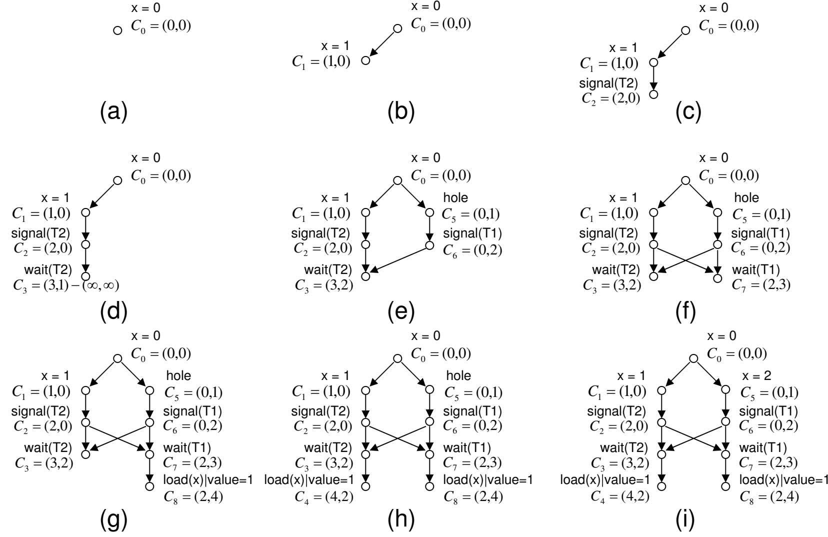
Figure 7: An example with synchronization operations

In Figure 7 (a) we show a program with two concurrent threads - namely T1 and T2. Both threads operate on a shared variable x which is initialized to zero before the program starts. In the program instruction 2 and 4 are sync (composite) operations which semantically require T1 and T2 to synchronize between each other. Note that sync is a composite operation. As we explained in Section 3.1 the sync operation is composed by performing a serial of signal-wait, instructions, $signal(T_1)$; $signal(T_2)$; $wait(T_1)$; $wait(T_2)$, in each thread. The corresponding program is shown in Figure 7 (b). [6]

For the program in Figure 7 (b) suppose instruction 1 has been performed and instruction 4 and 8 have arrived at the memory side. However instruction 5 has not arrived yet. An interesting question arises: Is it possible to perform instruction 4 and 8 before the arrival of instruction 5? Since instruction 1 is store candidate of both instruction 4 and 8, such a

[6]We omit $signal(T_1)$; $wait(T_1)$ in T1 and $signal(T_2)$; $wait(T_2)$ in T2 since they are redundant.

possibility does not violate causal ordering. Thus the OFC operational model should answer "yes" to the question. Now we explain how the possibility is achieved as the following scenario.



(a) Initial pomset. (b) The pomset after the arrival of instruction 1 (x = 1). (c) The pomset after the arrival of instruction 2 (signal(T2)). (d) The pomset after the arrival of instruction 3 (wait(T2)). (e) the pomset after the arrival of instruction 6 (signal(T1)). (f) the pomset after the arrival of instruction 7 (wait(T1)). (g) the pomset after the arrival of instruction 8 (load x). The load returns value 1. (h) the pomset after the arrival of instruction 4 (load x). The load returns value 1. (i) the pomset after the arrival of instruction 5 (x = 2). The clock range is denoted as a single vector clock if the lower bound and the upper bound are equivalent.

Figure 8: The pomsets of location x at each step for the program in Figure 7

Assume that at memory side instructions arrive in the order 1, 2, 3, 6, 7, 8, 4, 5. Although at processor side instructions may be issued in another order, it is beyond the following explanation. Then we explain the states of the memory location x on the arrivals of instructions step by step. The corresponding pomsets of memory location x in each step are shown in Figure 8. In follows the clock range is denoted as a single vector clock if the lower bound and the upper bound are equivalent.

- **Step (a):** The pomset of memory location x is initialized to contain a single node x = 0. The corresponding vector clock is (0, 0).

- **Step (b):** Instruction 1 (x = 1) arrives at the memory location. The corresponding vector clock is (1, 0) since it is a store node.

- **Step (c):** Instruction 2 (signal(T2)) arrives at the memory location. The corresponding vector clock is (1, 0) since it is a signal node.

- **Step (d):** Instruction 3 (wait(T2)) arrives at the memory location. The corresponding signal node is unknown at this time. However we know that the clock range of the signal node is (0, 1) to (∞, ∞). Thus the clock range of the wait node is (3, 1) to (∞). the signal node has not arrived yetit is a store node.

- **Step (e):** Instruction 6 (signal(T1)) arrives at the memory location. First of all according to the information carried by instruction 6, i.e. the fetch order, the number of signals instructions which are fetched before, etc, a hole which represents the instruction fetched before instruction 6 is created. And the hole must be either a load or a store. Thus the vector clock of the hole is (0, 1). So the vector clock of the signal(T1) node is (0, 2). Moreover the vector clock of the wait(T2) node is recalculated as (3, 2).

- **Step (f):** Instruction 7 (wait(T1)) arrives at the memory location. The corresponding vector clock is (2, 3) since the vector clock of signal(T2) node is (2, 0).

- **Step (g):** Instruction 8 (load x) arrives at the memory location. The corresponding vector clock is (2, 4). So instruction 8 is eligible to read the value written by instruction 1.

- **Step (h):** Instruction 4 (load x) arrives at the memory location. The corresponding vector clock is (4, 2). So instruction 4 is eligible to read the value written by instruction 1.

- **Step (i):** Instruction 5 (x = 2) arrives at the memory location. It replaced the hole without changing the corresponding vector clock.

4 Properties of The OFC Model

In this section we show that the OFC Model has all the properties that a sound memory model should have as listed in Section 2.1.

4.1 Causal ordering property

In this subsection we show that the OFC model has the causal ordering property.

In fact, under our abstract machine model a load operation from a processor does not need to obey any program order constraints - neither when it is issued from the processor nor when it travels in the interconnection network. When it arrives at the memory buffer,once it can be performed (i.e. there exists another candidate store operation which has been performed) it can bypass any other candidate store operations. In other words, the only constraint to a load operation for its completion is that the candidate store operation which causes the load operation is performed earlier. So the OFC model has the causal ordering property.

4.2 Equivalence property

In this subsection we show that the OFC model has the equivalence property. More precisely, we will define a way to convert a Sequential Consistency program to an OFC program with the guarantee that if the former program has no data race, it is equivalent to the latter one.

Before proving the property, it should be noted that the program models of the Sequential Consistency model and the OFC model are different. So for a parallel program $PROG_{SC}$ under the Sequential Consistency model, we should firstly define a way to convert $PROG_{SC}$ into another parallel program $PROG_{OFC}$ under the OFC model. Then we can prove that for any given $PROG_{SC}$ which has no data races, it generates the same result (or result sets) as the corresponding $PROG_{OFC}$ does.

For the first step, we define the rules on the conversion from $PROG_{SC}$ to $PROG_{OFC}$ as follows,

- **Conversions of load, store, signal and wait instructions:** Such instructions are directly used in $PROG_{OFC}$ without any change.
- **Conversions of barrier instructions:** Such instructions are replaced by sync($\{P_1,...,P_n\}$) instructions in $PROG_{OFC}$.
- **Conversions of critical Sections:** The start of a critical section is replaced by an acquire(all memory locations) instruction. The end of a critical section is replaced by a release(all memory locations) instruction.

Now it is time to prove the equivalence property of the OFC model. For easier understanding of the proof, we describe the equivalence property in another way, i.e. for a given parallel program $PROG_{SC}$ which has no data race under the Sequential Consistency model, any result that is generated by $PROG_{SC}$ can also be generated by the corresponding parallel program $PROG_{OFC}$ under the OFC model, and vice versa. Here a result means the correspondences between all load instructions and all store instructions so that each load instruction reads the value which is written by the corresponding store instruction. The proof is as follows with the condition that $PROG_{SC}$ has no data race.

- **Any Sequential Consistency result can be generated by $PROG_{OFC}$:** In a Sequential Consistency result, if a load reads the value that is written by a store, the store is the store candidate of the load. So for $PROG_{OFC}$ the load is also eligible to read the value that is written by the store. Thus the correspondences between loads and stores in the sequential consistency result is also legal for $PROG_{OFC}$. In other words, any Sequential Consistency result which is generated by $PROG_{SC}$ can also be generated by $PROG_{OFC}$.
- **Any OFC result can be generated by $PROG_{SC}$:** Through the rules on the conversion from $PROG_{SC}$ to $PROG_{OFC}$ it is known that all the synchronization operations in $PROG_{OFC}$ operate on the whole memory (all memory locations). So the partial orders

in the pomsets of all memory locations are defined by the same constraints – i.e. same program orders and same inter-thread orders. In other words, all the load and store instructions in $PROG_{OFC}$ are ordered by a partial order, which is also known as the happened-before order [12]. So in an execution of $PROG_{OFC}$, which implies that the happened-before order is determined, each load operation only has one candidate store operation, and moreover, the correspondences between loads and stores are legal in both $PROG_{OFC}$ and $PROG_{SC}$. Otherwise the corresponding execution of $PROG_{SC}$ causes either data race or that some load cannot read any value.

From above we conclude that we defined a way to convert a Sequential Consistency program to an OFC program with keeping the equivalence for the programs that have no data race. In other words, the OFC model has the equivalence property.

4.3 Monotonicity

In this subsection we show that the OFC model has the monotonicity property.

The monotonicity property says that if a result can be generated by a program, it can also be generated by a legal more parallel version of the same program. Here the more parallel version can be achieved by removing some arcs in some pomsets of memory locations. In fact, under our abstract machine model a candidate store operation to a load operation in a program is still the candidate to that load operation in a legal more parallel version. So that any result of the original program is also legal for the more parallel one. Thus the OFC model has the monotonicity property.

4.4 Non-intrusive reads

In this subsection we show that the OFC model has the non-intrusive reads property.

In fact, under our abstract machine model a load operation with respect to a memory location never changes the partial order in the pomset of the memory location. Moreover, a load operation never changes the state of a write – i.e. become eligible/ineligible for another loads. So a read never changes the state of the memory. Thus the OFC model has the non-intrusive reads property.

5 Benefits of the OFC Model

This section illustrates the benefits of the OFC model compared to popular memory models such as Release Consistency. We believe that the benefits presented in Section 5.2 and 5.3 are unique to our model. Nevertheless, the benefits presented in Sections 5.1 and 5.4 can also be (partially) achieved under a few other memory models (e.g. Location Consistency and the Java Memory Model). Note that our listed benefits below may not be complete. However, they

should help architects, compiler writers, and application developers to understand the features of the OFC model and explore them fully.

5.1 Causal ordering increases parallelism

The causal ordering property avoids unnecessary reorder restrictions, and thus brings more parallelism. More precisely the benefits are gained from the following two aspects.

- **Relax data dependence restrictions:** The causal ordering only requires that a store which causes a load is performed earlier than the load. As we showed in Section 2.1, the flow-, anti- and output-dependences are not always necessary to restrict instruction orders. And it is clear that the OFC operational model does not restrict any order in instruction issuing and transmission.
- **Relax synchronization restrictions:** Causal ordering does not require the instructions to be waiting for synchronizations. Thus instructions can be freely issued and transmitted without the restriction of synchronization operations.

5.2 Non-speculation based lock-free mutual exclusion

In the OFC model we implement the semantic of exclusive accesses without using lock, which gains benefits in the following aspects.

- As it is pointed out in [9], common problems associated with conventional locking techniques, such as priority inversion, convoying and deadlock, are avoided.
- Exclusive accesses can be performed concurrently if it guarantees that the result looks as if the accesses are performed exclusively. Thus more parallelisms are gained.
- As it is shown in Section 3.3, we achieve exclusion without speculation (as proposed in many implementations of Transactional Memory) hence avoiding the speculation overhead.

5.3 Ability to explore the bandwidth provided by multi-channel networks

As pointed out by experts in the field, fiber will displace copper sooner than you think. Using optical fiber technology at short distances as the main technology for inter-chip connection has already been recognized as an inevitable trend. Most recently, important research activities are underway to investigate the transition from copper to fiber on a single chip. Although the production level solution of the latter is still several years away, it is a very promising path to meet the demand on on-chip bandwidth up to a terabyte per second or beyond - projected for future generation of many-core/multi-core chip architectures.

An important feature of optical technology based interconnection networks is the capability to provide multi channels on each communication path - due to wavelength division multiplexing. This will provide ample opportunities to explore the OFC model since instructions issued from a processor toward a memory module can take any channel based on the traffic situation.

5.4 Opening up new opportunities for compiler / runtime optimizations

It is clear that OFC model has opened up new opportunities for compiler optimizations. A number of such opportunities are briefly listed below - although a detailed study is beyond the scope of this paper.

Portability of some uniprocessor compiler optimizations

One important challenge is how to ensure simple portability of many existing uniprocessor compiler optimization when moving to a multiprocessor (and multi-core) environment. Many such existing instruction reordering optimizations and their derivations cannot be simply enabled under the sequential consistency model or some of its derivations. The readers are referred to some excellent examples documented in [13].

For example, under the OFC model, if compiler reschedules instruction without violating uniprocessor data dependence the result code should be obviously correct during the execution under a multiprocessor (or multi-core) environment.

New opportunities for compiler-steered adaptive runtime reordering

As we described earlier, OFC model has eliminated the uniprocessor reordering constraints due to uniprocessor data dependence. In other word, instructions from a processor can be issued free of such constraints and the architecture model ensures correct results at the memory side. It is possible during a phase of execution of some application the memory side may become a bottleneck. We can imagine that compiler may perform static analysis and classify and label certain instructions so their issuing from a processor should subject to the runtime load situation. If memory side becomes congested - such annotated instruction can be delayed until the situation improve.

6 Related Work

In this section, we discuss the related work. We list representative memory models which are relevant to the OFC model. Then, we discuss them by the comparison with the OFC model. Such memory models include: Gharachorloo et al. the Release Consistency (RC) [8], Keleher et al. the Lazy Release Consistency (LRC) [11], Bershad et al. the Entry Consistency (EC) [2], Gao et al. the Local Consistency (LC) [6], Shen et al. the Commit-Reconcile & Fences (CRF) model [17], Manson et al. the Java memory model [13] and Saraswat et al. the Relaxed Atomic + Ordering (RAO) model [16].

- **The SC-derived models:** [7] defines SC-derived models as the memory models which assume memory coherence. Memory coherence can be stated as follows [8]: all writes to the same location are serialized in some order and are performed in that order with respect to any processor. As the definition, RC, LRC and EC are included. The fundamental difference between the SC-derived models and the OFC model is that the former assumes memory coherence but the latter not.

- **The LC model:** The distinguishing property of LC model is that it does not rely on coherence, thus dispensing the need for cache snooping and directories in a multiprocessor implementation. [16] Like LC, the OFC model do not relay on the coherence assumption. However LC model assumes that "all uniprocessor control and data dependences are satisfied." [6, 7] So the LC model introduces more reordering restrictions than the OFC model.

- **The CRF model:** The CRF model exposes a notion of semantic cache (sache), and decomposes load and store instructions into two finer-grain operations [17], which can freely control the moment for writing back a value from sache to the memory or purging a stale value in the sache. However as it is said in the conclusion of [17], "instruction reordering is constrained only by data dependences and memory fences." Thus the CRF model introduces more reordering restrictions than the OFC model.

- **The Java memory model:** The Java memory model well defines a notion of causality which "is strong enough to respect the safety and security properties of Java and weak enough to allow standard compiler and hardware optimizations." [13] However as we discussed in Section 2.1 the Java memory model violates the monotonicity property. Moreover as it is pointed out in [16]: "In contrast, the methodological stance of [13] is that a trace must be given beforehand; the memory model is then specified in terms of which traces are correct." Thus it is hard to know how a real parallel machine can execute a program under the specification of the Java memory model.

- **the RAO model:** The RAO model can be considered as an improvement of the Java memory model since "RAO is generative, given a source program it generates all possible sequences of executions." [16] However as we discussed in Section 2.1 the RAO model also violates the monotonicity property. Moreover we feel that "generative" cannot explain how a real parallel machine works under the specification of the RAO model.

7 Conclusion and Future Work

In this paper we have proposed a novel memory model which is truly asynchronous in the following ways:

- Memory operations can be issued freely from the processors without being blocked by any memory-based data dependence.

- The memory transmissions can travel through the interconnection network freely without worrying that they may arrive at the destination out of order.

Moreover, this paper presents four desired properties of memory models (i.e. causal ordering, monotonicity, equivalence and non-intrusive reads). All of these were explained and applied to our memory model, by a series of examples and explanations. In this way, we show that our model displays all four desired properties.

Furthermore, we defined an operation model (hence construct an abstract machine) that can fully explore the above features during program execution. We argued that our memory model satisfies the new generation of multi-core chip architectures and takes the advantage of the optical inter-chip and photonics intra-chip technologies on the interconnection networks. Besides this, it also allows the exploitation of many reordering opportunities that were hidden before thanks to unnecessary constrains due to data dependences.

As the abstract machine model has been defined, an implementation of our memory model cannot be far behind. This implementation is far out of the scope of this paper, but it is the logical next step. We are planning to use a simulator to test the memory model. Current plans points to using a simulator for the Cyclops-64 cellular architecture, which is introduced in [19, 3]. This architecture is characterized by having 160 thread units, around 4.7 Megabytes of on chip SRAM and a high bandwidth crossbar interconnect. After that the optical inter-chip and photonics intra-chip interconnections will be used in the simulation.

Thanks to this implementation, we can test selected benchmarks and how this memory models, measures against the canonical SC-derived memory models.

Acknowledgement

The authors would like to thank Professor Vivek Sarkar for his feedback on earlier discussions of the OFC model.

This work was conducted at Computer Architecture and Parallel System Lab (Direcor: Professor G.R. Gao,at University of Delaware) that is, in part, supported by sponsors, such as NSF (e.g. grant #: CSR-AES-720531, CSR-0708856, CSR-0702244, CNS-0509332) and others.

References

[1] S. V. Adve and M. D. Hill. A unified formalization of four shared-memory models. *IEEE Trans. on Parallel and Distributed Systems*, 4(6):613–624, 1993.

[2] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon. The Midway distributed shared memory system. In *Proc. of the 38th IEEE Int'l Computer Conf. (COMPCON Spring'93)*, pages 528–537, February 1993.

[3] L. Chen and Z. Hu. Optimizing fast fourier transform on a multi-core architecture. In *Workshop on Performance Optimization for High-Level Languages and Libraries in the 21st IEEE International Parallel and Distributed Processing Symposium (IPDPS 2007).* University of Delaware, March 26 2007.

[4] Schares et al. Terabus: Terabit/second-class card-level optical interconnect technologies. *IEEE Journal of Selected Topics in Quantum Electronics*, 12(5):1032–1044, October 2006.

[5] M. Frigo. The weakest reasonable memory. Master's thesis, Department of Electrical Engineering and Computer Science, MIT, 1998.

[6] G. R. Gao and V. Sarkar. Location consistency: Stepping beyond the barriers of memory coherence and serializability. Technical Report ACAPS Technical Memo 78, 1993.

[7] 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.

[8] 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.

[9] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the Twentieth Annual International Symposium on Computer Architecture*, 1993.

[10] J. A Kash. Intrachip optical networks for a future supercomputer-on-a-chip. *Photonics in Switching*, pages 55–56, August 2007.

[11] Pete Keleher, Alan L. Cox, and Willy Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proc. of the 19th Annual Int'l Symp. on Computer Architecture (ISCA'92)*, pages 13–21, 1992.

[12] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[13] 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.

[14] Friedemann Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms: proceedings of the International Workshop on Parallel and Distributed Algorithms*. Chateau de Bonas, France, 1989.

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

[16] 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.

[17] Xiaowei Shen, Arvind, and Larry Rudolph. Commit-reconcile & fences (crf): a new memory model for architects and compiler writers. In *ISCA '99: Proceedings of the 26th annual international symposium on Computer architecture*, pages 150–161, Washington, DC, USA, 1999. IEEE Computer Society.

[18] K-S. Weng. *An Abstract Implementation for a Generalized Data Flow Language*. PhD thesis, MIT, Cambridge, MA, May 1979.

[19] Weirong Zhu, Vugranam C Sreedhar, Ziang Hu, and Guang R. Gao. Synchronization state buffer: supporting efficient fine-grain synchronization on many-core architectures. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, pages 35–45, New York, NY, USA, 2007. ACM.