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

Lamport Order Revisit: A Study on How to Efficiently Achieve Sequential Consistency on a Modern Multiprocessor-on-a-chip Architecture

Yuan Zhang†
Weirong Zhu
Fei Chen
Ziang Hu
Guang R. Gao

CAPSL Technical Memo 53
March 01, 2004

†Dept. of Electrical and Computer Engineering
University of Delaware
zhangy,weirong,fchen,hu,ggao@capsl.udel.edu

Abstract

In his seminal paper in 1979 [1] on memory consistency, Lamport proposed two requirements for a multiprocessor system to be sequentially consistent. The second condition stated that memory "requests from all processors issued to an individual memory module are serviced from a single FIFO queue. Issuing a memory request consists of entering the request on this queue". Recently, the authors have the opportunity to revisit Lamport's conditions in the context of a design study of the IBM Cyclops multiprocessor-on-a-chip architecture (known as BG/C) from the system software angle. We find that when a multiprocessor system employs a network to communicate with its shared memory modules – such as in the BG/C architecture - we need to carefully elaborate Lamport's requirements to cover the network. Thus we have extended the Lamport's second requirement along this line and demonstrated that the revised conditions are sufficient for ensuring the sequentially consistent behaviors for a class of BG/C-like architectures.

Contents

List of Figures

1 Introduction

A memory model defines how memory system behaves in a computer system. A sequential computer's memory model is straightforward: every read operation returns the value of the last write operation to the same location, and every write binds its value to the subsequent read operations until the next write to the same location. However, the memory model of a shared-memory multi-processor machine is complicated, since the definitions such as "subsequent read" and "last write" need to be redefined when there are multiple processors reading and writing the same memory location.

The most widely accepted memory model for the multiprocessor machine is Lamport's *sequential consistency* (SC) model - a simple extension of that of uniprocessor machine. It was described by Lamport in the following well-known statement [1]:

> [A system is *sequentially consistent* if] the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program order.

The above quote becomes the commonly used definition of sequential consistency in most text-books and research papers.

In his 1979 paper cited above, Lamport also illustrated how to satisfy SC using a simple hypothetical shared-memory multiprocessor system [1]. This hypothetical multiprocessor system will be described in more detail in Section 2, but briefly, it consists of a collection of processors and shared memory modules, and the processors communicate with each other only through shared memory read and write operations. Lamport posted the following two requirements as sufficient to ensure this system to be sequentially consistent:

1. R1: Each processor issues memory requests in the order specified by its program.
2. R2: Memory requests from all processors issued to an individual memory module are serviced from a single FIFO queue. Issuing a memory request consists of entering the request on this queue.

The work described below is mostly inspired by a suggestion from Henry Warren from IBM T.J. Watson Center [2] to consider a proof of sequential consistency for the BG/C multiprocessor machine designed by Monty Denneau [3]. Denneau's multiprocessor-on-a-chip architecture employs a switch network between the processors and memory modules in a so-called dance-hall configuration [4]. Compared with Lamport's hypothetical multiprocessor, the queue attached to each memory module is now on the "memory" side of the network. Denneau believes that the base BG/C chip architecture is sequentially consistent and there is no need to issue fence-like instructions after each memory operation to ensure SC.

Although Dennear's conjecture is, at first glance, quite obvious, it took us much longer to construct a proof to be presented in this paper. A major issue is: a memory operation needs to go through the network between "issued" by the processor and entering the corresponding memory module, instead of as in condition R2 "... Issuing a memory request consists of entering the request on this queue".

Therefore, the proof process will need to understand what features of the BG/C on-chip network satisfy the requirement of the original Lamport's R2 condition. We end up with a refined R2 condition that will be illustrated in Section 3. Another challenge in the proof is to show that the condition R2 (or the refined R2) is sufficient to lead the validity of the widely referred Lamport's SC definition (see the quotation in Section 1). In a real multiprocessor system, the lifetimes of memory operations may actually be overlapped - a fact also pointed out by Lamport. Therefore, the "total order" in Lamport's SC definition, in the mathematical sense, needs more deliberation. We believe that to construct a "total order" as described in Lamport's definition, it appears to be much easier for the programmers to use or think about the issuing order, i.e., the order of memory requests described by their issuing time from processors. To this end, our proof shows that in the context of the BG/C chip architecture, which satisfies our refined R2 condition, and begin with a total order based on the completion time, we can actually perform a series of "result-preserving" reordering transformations back to a total order based on the issuing order. Therefore, we can show that the base BG/C architecture is sequentially consistent compliant - a concept introduced later in Section 4.

Let us put this paper in a broader context. We have witnessed the emerging technology trend on multiprocessor-on-a-chip architecture with 10s-100s processing/thread units. In the general-purpose parallel computing arena, a representative multiprocessor-on-a-chip architecture is the class of cellular architectures (IBM BG/C architecture [5, 6, 7, 8] is one example of this class). At the chip level, the cellular architecture employs a decentralized microprocessor design – using a cellular organization interconnecting a large number of very light-weight processors called processing cells. A thread running on the processing cell carries very little state – the processor is very simple and employs simple in-order issue to reduce the hardware complexity. The on-chip communication network provides rich interconnection and sufficient bandwidth for interprocessor communication among the processing cells and their shared memory. Another class, in the domain of application-specific architectures, is called *extreme chips* [9]. As many as 300+ RISC processors are integrated within one chip to perform process communication tasks. Therefore, designing such a chip and establishing its memory consistency requirement have a practical significance. We are not aware of any de facto standard on how to prove a real architecture to be sequentially consistent. To this end, we wish to share our initial experience with BG/C to the readers and hope it may help to foster more discussions.

This paper is organized as follows. Section 2 first revisits Lamport's two requirements in the context of a multiprocessor with a dance-hall organization of its shared memory modules and network, then proposes our refined requirements. In section 3 we first introduce the base IBM BG/C Cyclops multiprocessor-on-a-chip architecture, which is referred to as base Cyclops architecture (or simply Cyclops architecture when no confusion may occur) when we discuss

the memory model in section 4, and then give the problem formulation. In section 4 we present an outline of our proof that the Cyclops architecture behaves like a sequentially consistent system, and leave some of the details in the Appendix I and Appendix II. In section 5, we give a brief discussion of some extensions to the base Cyclops architecture and discuss how they may influence the underline memory model. We then discuss our future work in the same section. The related work is listed is section 6. Section 7 summarizes the whole paper work and gives the conclusions.

2 Motivation

The architecture model we concern, as shown in figure 1, consists of a collection of processors and memory models, and the processors communicate with one another only through reading or writing shared data in memory modules. Processors and memory modules are connected by a single bus or general network. We called it a "dance-hall" configuration (much inspired by past literature).

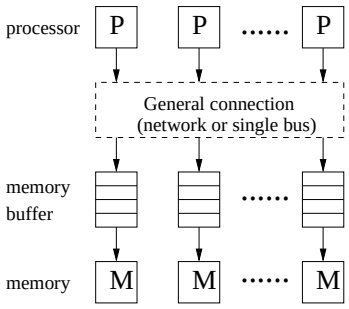


Figure 1: General Dance-hall Architecture Model

Let us first examine the sufficient condition for this system to be sequentially consistent. As mentioned in the previous section, Lamport has proposed two requirements in 1979, and the second requirement is "R2: Memory requests from all processors issued to an individual memory module are serviced from a single FIFO queue. Issuing a memory request consists of entering the request on this queue." R2 implies that an operation is *issued* only when it has reached the memory buffer. When the R2 condition was explained using his hypothetical multiprocessor system, Lamport did not elaborate what requirements the network should follow in order to ensure the sequential consistency.

In real architecture models, networks cause some delay in transmitting a memory request, i.e. a delay from the time it is issued by the processor to the time it arrives at the destination memory queue after travelling through the network. If no restriction on the distribution of such delay is imposed, a network design may violate the R2 condition. Let's consider the following example:

```
    Initially x = flag = 0

    P1:                 P2:

i1: x = 1;          i3: while(flag==0) {;}
i2: flag = 1;       i4: read x;
```

We expect that in most "reasonable" memory models $i4$ returns $x = 1$. Programmers may reason this example (say, under Lamport's hypothetical multiprocessor model) as follows. We know that $i1$ is issued before $i2$, and $i3$ is issued before $i4$ - from the condition R1. Since the *while* loop in $i3$ exits only when the check of $flag$ returns value 1, $i2$ must have been performed at that time. Consequently, when $i4$ is issued, $i1$ has already been issued. Thus $i4$ will be performed after $i1$ and return the value written by $i1$.

However, suppose in a real dance-hall system x and $flag$ are in different memory modules, and the delay occurs in the network such that although $i1$ is issued to the network before $i4$, it may arrive at the memory buffer for x later than $i4$. If this happens, then $i4$ returns the wrong value 0 - a contradiction to sequential consistency!

This inspires us to re-consider the sufficient requirements for a general dance-hall system to be sequentially consistent, and propose the following refined R2 requirement:

> **R2-refined:** Two operations designated to the same memory model M will be delivered to M's FIFO queue in the same order as they entered into the network.

We will prove later that, in the context of a class of multiprocessor-on-a-chip architecture such as the IBM BG/C architecture, R1 and R2-refined together are sufficient to ensure the sequentially consistent behavior of a general dance-hall system.

3 Problem Formulation

3.1 Target Architecture Model

Figure 2 illustrates the base Cyclops multiprocessor-on-a-chip architecture model. The base architecture model simplifies the actual Cyclops architecture and only contains the features relevant to our study of the memory model. In section 5 we will present a more "complete" Cyclops architecture model, and discuss how our results derived from the base model may also be applicable.

The base Cyclops architecture is composed of a collection of processors and memory banks connected by a crossbar. The memory banks could be either on-chip memory or off-chip memory. Each processor issues its memory operations in program order into a FIFO queue called the "issuing buffer". The design of the processor guarantees that memory requests are issued into the issuing buffer in program order. [1]

[1] In fact, a memory request cannot be issued unless its uniprocessor data dependence and control dependence are satisfied.

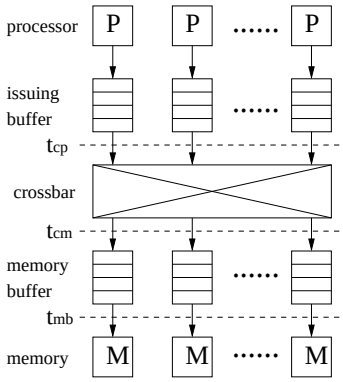


Figure 2: Base Cyclops Architecture Model

The crossbar provides the following nice property, which we name as the "equal latency" property. Once a memory operation, which accesses memory bank M, is "admitted" (from the issuing queue) to the network, the time it takes to travel the network and reach M's memory buffer is a constant, regardless its specific origination. This constant latency applies to all memory requests to M - hence the term "equal latency" property. If two memory requests have the same memory destination are ready to be issued at the same time, one of them, determined by arbitration, will be stalled at its issuing buffer until the conflict is resolved.

The programming model is rather simple – only "read" and "write" operations are concerned. In section 5 we will discuss other memory operations like synchronization operations.

3.2 Problem Statement

As described in the introduction, it has been conjectured that the base Cyclops architecture *obeys* the sequential consistency, and this is true even without using an explicit "sync" or "fence" [10] instruction after each read or write instruction. The problem statement of this paper is: *can we establish the validity of this conjecture and how*?

Before we proceed to present our proof, we wish to discuss why the answer, although might look intuitive, is not trivial.

Intuitively, the equal-latency property of the Cyclops architecture should satisfy the R2-refined condition we have outlined earlier. And it is easy to see that Cyclops satisfies both R1 and R2-refined, hence the system appears to be sequentially consistent without elaboration - as Lamport did in his 1979 paper. However, as we have discussed earlier in the introduction, the relation between lifetimes of two memory operations in a real architecture cannot be characterized as "happen-before". In reality, in the construction of the "total order" mentioned in the Lamport definition of the sequential consistency, two operations may be ordered (in the sense of Lamport order) by their issuing time, or their completion time. Other possibilities also exist, but let us limit to these two which are the most intuitive ones used. It is our belief that it is often easier for a programmer to think about the order of memory operations

characterized by the time when they leave a processor, instead of worrying about the order characterized by their execution time at the memory end - as that may be "very far away" and the operations may travel through various stages, or queues, etc. To this end, our proof shows that in the context of the base Cyclops architecture (that satisfied our refined R2 condition) and begin with a total order based on the completion time, we can actually perform a series of "result-preserving" reordering transformation back to a total order based on the issuing order. Therefore, we can show that the base BG/C architecture is sequentially consistency compliant (a concept introduced later in Section 4).

4 Cyclops Memory Model

In this section, we will outline the proof that the base Cyclops architecture model, as presented in the previous section, obeys sequential consistency. This section is assisted by Appendix I and Appendix II, where some details of the proof is described.

The main body of the proof consists of **Theorem I** and **Theorem II** . **Theorem I** states that Cyclops is not sequentially consistent in the *classical* sense, and **Theorem II** states that, nonetheless, Cyclops architecture is sequentially consistent compliant, a notion we will introduce later.

The *classical* sequential consistency (SC) is defined by Lamport as:

> [A system is *sequentially consistent* if] the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program order [1].

There are two conditions indicated in this definition:

(1) Operations of all processors are executed in some sequential order (i.e., a total order).

(2) In that total order, operations from an individual processor are executed in the program order.

We call the total order which satisfies condition (2) the "Lamport order".

Theorem I: Cyclops is not sequentially consistent in the *classical* sense.

Proof:

Consider the following example:

```
Initially x=y=0

P1:              P2:

i1: write x=1    i3: read y
i2: write y=1    i4: read x
```

Suppose memory location x is in memory bank M_x, and memory location y is in memory bank M_y, $M_x \neq M_y$. Let $\prec$ denotes the program order, then $i1 \prec i2$, and $i3 \prec i4$. Because the processor in Cyclops issues its memory operations in program order, we have $t_{cp}(i1) < t_{cp}(i2)$ and $t_{cp}(i3) < t_{cp}(i4)$ (please refer to figure 2 for the definition of t_{cp}). Then $t_{cm}(i1) = t_{cp}(i1) + C(M_x)$, and $t_{cm}(i2) = t_{cp}(i2) + C(M_y)$. Without any loss of generality, we assume that before $i1$ there are already W_x operations waiting at M_x's buffer, and W_y operations waiting at M_y's buffer. Also assume that in average it takes S_x units of time and S_y units of time for M_x and M_y, respectively, to process a memory request, then $t_{mb}(i1) = t_{cm}(i1) + W_x * S_x = t_{cp}(i1) + C(M_x) + W_x * S_x$, and $t_{mb}(i2) = t_{cm}(i2) + W_y * S_y = t_{cp}(i2) + C(M_y) + W_y * S_y$. Suppose $W_x > W_y$, $S_x > S_y$ and $C(M_x) > C(M_y)$ (this probably happens in real machine), we have $t_{mb}(i1) > t_{mb}(i2)$. That means, $i1$ will be performed after $i2$, which breaks the program order specified by $P1$. The same thing may also happen between $i3$ and $i4$. Therefore the condition (2) in Lamport's definition may not be satisfied. Thus in the classical sense, Cyclops is not sequentially consistent, *i.e.*, not obeying the Lamport order.

Proof done.

Although Cyclops is not sequentially consistent in *classical* sense, it *behaves* like sequential consistency. If executing the above example in a sequentially consistent system, $i3$ and $i4$ may return the result for (x, y) as one of $(0, 0)$, $(1, 0)$ or $(1, 1)$, but not $(0, 1)$. In Cyclops, the readers may wish to be convinced that the same results will be obtained. (Hint: to show that $(0, 1)$ is not possible – note that $i3$ will happen after $i2$ to read $y = 1$. This implies that $i1$ is definitely issued before $i4$ to M_x, thus $i1$ is served by M_x before $i2$.)

In this sense, we call Cyclops "sequentially consistent compliant", a notion to be introduced later. We will try to prove this assertion in **Theorem II**. But before that, let us consider the intuitive reasoning behind the rigorous proof. Note that the "total order" in Lamport sense implicitly implies an ideal scenario that there is no *overlapping* between operations in the total order. In reality, such as in the case of Cyclops architecture model, however, this "lifetime" of an operation may be overlapped with others, *i.e.*, an operation may start earlier than another, but finish (perform) later. This is reflected, in fact, in the counter example shown in the proof of **Theorem I**. So we need to have a formalism to handle the above overlapping situation.

Our proof of **Theorem II** proceeds as follows: we first represent all operations executed for a parallel program *Prog* as a multi-set Ψ. On Ψ we build up two total orders, one is χ, based on the *performing time* (*i.e.*,t_{mb}) of operations, and the other is λ, based on the *issuing time* (*i.e.*,t_{cp}) of operations. Then the "execution" is conceptually represented as the (Ψ, χ) pair. We also introduce another "pseudo" execution (Ψ, λ). We prove in **Lemma 1** that λ is a Lamport order, thus (Ψ, λ) is a sequentially consistent execution. Then in **Lemma 3**, we show that the order χ can be transformed – through a series of reordering steps that preserve "equivalence" (based on **Lemma 2**) under the guidance of order λ – into a Lamport order, hence (Ψ, χ) is equivalent to a sequentially consistent execution.

The two total orders on Ψ are mathematically defined as:

- $\chi = \{< i, j > | i \in \Psi \wedge j \in \Psi \wedge (t_{mb}(i) < t_{mb}(j) \vee (t_{mb}(i) = t_{mb}(j) \wedge Mem(i) \leq Mem(j)))\}$
- $\lambda = \{< i, j > | i \in \Psi \wedge j \in \Psi \wedge (t_{cp}(i) < t_{cp}(j) \vee (t_{cp}(i) = t_{cp}(j) \wedge Mem(i) \leq Mem(j)))\}$

where $Mem(i)$ shows which memory bank the operation i accesses, assume every memory bank has a unique ID. Although in a real system two operations with the same t_{mb} may be executed simultaneously (by different memory banks), forcing an order here neither loses any generality nor influences the validity of the proof and conclusions. From **Theorem I** we know $\chi \neq \lambda$.

Our proof is based on some important definitions:

Definition 1: An **execution** of a parallel program *Prog* at system S is a (M, O) pair, where M is the multiset of operations executed, and O is an order on M.

Definition 2: An execution (M, O) is a **sequentially consistent execution** if O is a Lamport order.

Definition 3: Two executions (M_1, O_1) and (M_2, O_2) of a parallel program $Prog$ are **equivalent** if:

(1) They have the same set of memory operations, *i.e.*,$M_1 = M_2$ [1];

(2) Any memory read operation in (M_1, O_1) returns the same value as (M_2, O_2), and vice versa;

It is obvious that an execution is equivalent to itself.

Definition 4: A shared-memory system S is **sequentially consistent compliant** if any execution of a program $Prog$ on S is equivalent to a sequentially consistent execution of $Prog$.

Lemma 1: Execution (Ψ, λ) is a sequentially consistent execution.

Proof:

[1] The need for this condition may not be obvious. It is needed when there are some reads whose return values determine whether an operation is executed or not, or what memory address will be accessed. Let us look at the following example:

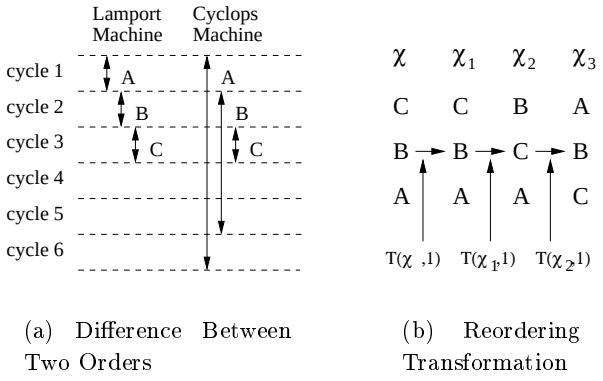
```
r1 = read x;
if(r1 == 0)
        write y = 1;
else
        write z = 1;
```

Here the return value of "read x" determines whether then-branch or else-branch is executed. If two executions have different return values of "read x",they are not equivalent because they have different *execution paths*.

Let i and j be two memory operations issued from processor P, and $i \prec j$, then $t_{cp}(i) < t_{cp}(j)$. According to the definition of λ, we have $< i, j > \in \lambda$. Program order is preserved in λ. According to **Definition 2**, we know (Ψ, λ) is a sequentially consistent execution.

Proof Done.

Readers can think of this pseudo execution (Ψ, λ) as executing Ψ on a "ideal Lamport machine", in which operations are executed one by one in a sequential order. For each operation its issuing time is its performing time, therefore there is no lifetime overlapping between any pair of operations. This case is illuestrated in figure 3(a). However, in Cyclops, memory operations may be out-of-order executed, because of network delay, for instance; their lifetimes are overlapped with each other. The order based on their performing time is χ, which is different from λ.



(a) Difference Between Two Orders

(b) Reordering Transformation

Figure 3: Illustration of Different Orders and Reordering Transformations

In order to prove (Ψ, λ) and (Ψ, χ) are equivalent, we introduce a reordering transformation T. Given a total order, say θ, $T(\theta, i)$ transforms θ into θ' by "percolating" the ith operation $O(i)$ in θ upward and insert it in a position that do not violate the corresponding ordering specified by λ. We have developed a pseudo code (see Appendix I) to describe this process. Set $\chi_0 = \chi$ and begin from $i = 1$, we apply a sequence of Ts on χ with increasing values of i, *i.e.*,$\chi_1 = T(\chi_0, 1), \chi_2 = T(\chi_1, 2), \ldots, \chi_n = T(\chi_{n-1}, n)$. Finally we reach the order $\chi_n = \lambda$. We will proof that $\chi = \chi_1 = \ldots = \chi_n = \lambda$ (where "=" denotes equivalence).

Readers will notice that after $\chi_i = T(\chi_{i-1}, i)$, the order of the first i operations in χ_i are consistent with λ, and the rest operations are ordered in the same way as χ. Figure 3(b) gives a simple example of this series of transformations. We also assume that the multi-set Ψ is changed to Ψ' due to T. We will proof later that $\Psi = \Psi'$.

Lemma 2: Execution (Ψ', χ_i) is equivalent to (Ψ, χ_{i-1}), after the transformation $\chi_i = T(\chi_{i-1}, i)$, which reorders $O(i)$ in χ_{i-1}.

The details of this proof is shown in Appendix II. But briefly, this proof is proceeded by

contradiction. We first assume that reordering operation $O(i)$ in χ_{i-1}, which is either read or write, will cause at least one read operation to return different value. Then a conflict with an obvious fact will occur.

Lemma 3 (Ψ, χ) is equivalent to a sequentially consistent execution.

Proof:

Because $\chi = \chi_1 = \chi_2 = \ldots = \chi_{n-1} = \chi_n = \lambda$, then $\chi = \lambda$. From **Lemma 1**, we know (Ψ, λ) is a sequentially consistent execution. From **Definition 3**, we know that (Ψ, χ) is equivalent to a sequentially consistent execution.

Proof Done.

Theorem II: Cyclops is sequentially consistent compliant.

Proof:

Lemma 3 can be applied to every execution of a parallel program *Prog* at Cyclops. According to **Definition 4**, Cyclops is sequentially consistent compliant.

Proof Done.

5 Discussion and Future Work

5.1 More About Cyclops Architecture

In section 3 we introduce a base Cyclops architecture and then discuss its memory model in section 4. In this section we present the actual Cyclops architecture models with extended features and show they are still sequentially consistent compliant.

The extended architecture model is shown in figure 4. There are the same number of processors and memory banks in the system. Compared with the base model shown in figure 2, we take into account the acknowledgments for read operations, which are issued to processors from memory banks, and are buffered before entering the crossbar and corresponding processors. Another extension is that each pair of processor and memory bank share the same crossbar port in order to utilize the ports efficiently.

Under these extensions, a memory request, either read or write, is first issued by the processor into buffer α. From there it enters buffer β of its destination memory bank after travelling through the crossbar. On the other hand, a read acknowledgment first enters buffer α, then arrives in buffer β of its destination processor through the crossbar. A write operation needs no acknowledgment. Therefore, both buffer α and buffer β are shared by read/write requests and read acknowledgments.

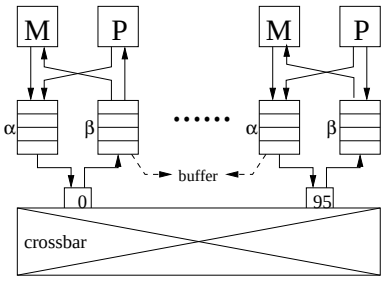


Figure 4: Actual Cyclops Architecture Model

This architecture still behaves like sequential consistency. Although read acknowledgments share same buffers with memory requests, they just cause more delay but do not change the relative *performing order* of memory requests. Moreover, in a system without cache, an operation is *globally performed* [11, 12] when it has finished taking action on the memory. In this sense how the processor is acknowledged does not change the memory model, suppose uniprocessor data dependence and control dependence are preserved.

The architecture model shown in figure 5 tries to improve the system performance further by grouping the memory operations into two classes: class 0 and class 1, and providing each class with a separate set of buffers. The crossbar serves both classes in the manner that whenever possible, it chooses one operation from the Least Recent Used (LRU) class to send. Thus we call it dual *virtual* channel crossbar. This model provides more efficient network flow control mechanism, and avoids system deadlocks caused by overwhelming network traffic [2].

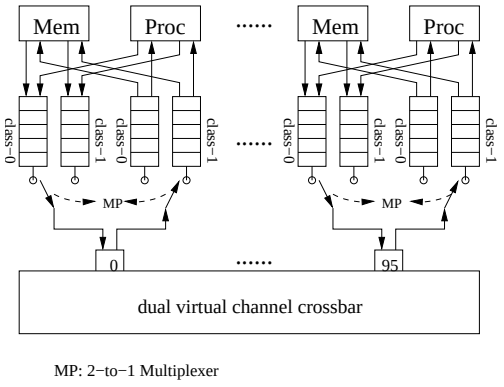


Figure 5: Cyclops Architecture Model with Dual Virtual Channel Crossbar

How to group the memory operations here determines the type of memory model. We have three options:

(a). class 0: read request and write request;
class 1: read acknowledgment;

[2]For example, processor P1 stops and waits at shared variable $flag$. By polling repeatedly, it may congest the network. At the same time, process P2 attempts to change the value of $flag$ thereby processor P1 can catch and continue to work, but is stalled by the network congestion.

(b). class 0: read request and read acknowledgment;
class 1: write request;

(c). class 0: read request;
class 1: write request and read acknowledgment;

Option (a) is in effect similar as the model shown in figure 4, thus it is sequentially consistent. Option (b) will break the sequential consistency in two ways: break the uniprocessor data dependence, and break the inter-processor synchronization. For the former case, consider a processor issues n read operations followed by a write operation, which depends on the ith read operation, $i \leq n$. For some reason channel 0 is congested, so the write operation is executed before the ith read operation, uniprocessor data dependence is broken. For the latter case, consider the following code piece:

```
Initially x = flag = 0

  P1:            P2:

read x;       while(flag==0){;}
flag = 1;     write x=1;
```

Suppose P1's class 0 buffer is stalled by previous operations, then its "read x" operation may get the value written by P2's $writex = 1$, which breaks the sequential consistency.

Option (c) has the same problems as option (b), so is not sequential consistency either.

5.2 Future Work

In present Cyclops architecture, there is no explicit synchronization operation like *acquire* and *release* etc. The exclusive access to a memory location is achieved by applying spin-locks in user code. As one direction of future work, we are considering to add explicit synchronization operations into Cyclops' instruction set, and see how we can take advantage of Cyclops' sequential consistency to efficiently implement those synchronization operations.

6 Related Work

Memory consistency model is an intensively studied field in parallel computer architecture, with a large amount of research work published in the literature. Sequential consistency was first defined by Lamport [1, 13] for shared-memory multiprocessor system with network but no cache. Afterwards, a number of cache coherence protocols, which ensure sequential consistency, have been proposed for single bus cache-based systems [14, 15, 16, 17]. Scheurich and Dubois proposed a sufficient condition for sequential consistency at a cache-based system [11, 12]. Shasha and Snir also proposed a software algorithm to ensure sequential consistency [18].

Besides, a lot of research work were conducted to relax the strong condition of sequential consistency to allow more performance optimization. They include processor consistency [19], weak consistency [20], release consistency [10], etc. All of the above discuss the memory model from the system point of view. Adve proposed two memory models from the programmer's point of view [21, 22], and stated that if software obeys the synchronization model defined by the memory model, then the hardware appears sequentially consistent. Some researchers try to find the *weakest* memory model [23, 24, 25]. Gao and Sarkar proposed the location consistency model (LC) and a cache coherence protocol in 2000 [26], which does not rely on the memory coherence assumption.

At the contex of memory consistency model proof, Lamport proposed a method based on logical clock and time [27]. Based on Lamport's work, Plakal proposed a reasoning technique to verify a directory cache coherence protocol [28, 29].

7 Conclusions

In this paper we specify the memory model of Cyclops multiprocessor-on-a-chip architecture (known as BG/C). We first check Lamport's two requirements for a dance-hall architecture to be sequentially consistent. We find that Lamport's two requirements R1 and R2 need to be carefully elaborated when we take into account the network delay. Then we proposed the revised requirement – R2-refined. We informally prove that the base Cyclops architecture, which satisfies both R1 and R2-refined conditions, obeys sequential consistency. We also discuss the real Cyclops architecture and show it is also sequentially consistent compliant.

8 Acknowledgments

We would like to acknowledge Monty Denneau and Henry S. Warren from IBM T.J. Watson Research Center for their suggestions on this topic and the many communications without which the progress of this research would not be possible. We also thank useful discussions from members of the CAPSL group at the University of Delaware, in particular Hongbo Rong, Juan del Cuvillo and Andres Marquez. Finally, the last author wish to acknowledge the support in part by NSF, under the NGS grant 0103723, DOE, under grant number DE-FC02-01ER25503, DARPA, under the HPCS program, and other funding agencies.

References

[1] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):690–691, September 1979.

[2] Henry S. Warren. Personal communication, February 2004.

[3] Monty Denneau. Personal communication, February 2004.

[4] Jaswinder Pal Singh David E. Culler and Anoop Gupta. *Parallel Computer Architecture, a Hardware/Software Approach.* Morgan Kaufmann, 1998.

[5] George S. Almasi, Călin Caşcaval, José G. Castaños, Monty Denneau, Wilm Donath, Maria Eleftheriou, Mark Giampapa, Howard Ho, Derek Lieber, José E. Moreira, Dennis Newns, Marc Snir, and Henry S. Warren, Jr. Demonstrating the scalability of a molecular dynamics application on a petaflops computer. *International Journal of Parallel Programming*, 30(4):317–351, August 2002.

[6] George S. Almasi, Daniel K. Beece, Ralph Bellofatto, Gyan Bhanot, and EtAl. Blue gene/l, a system-on-a-chip. In *2002 IEEE International Conference on Cluster Computing (CLUSTER 2002)*, Chicago, IL, U.S.A, 2002. IEEE Computer Society.

[7] Calin Cascaval, Jos G. Castaos, Luis Ceze, Monty Denneau, Manish Gupta, Derek Lieber, Jos E. Moreira, Karin Strauss, and Henry S. Warren Jr. Evaluation of a multithreaded architecture for cellular computing. In *Proceedings of the Eighth International Symposium on High-Performance Computer Architecture (HPCA'02)*, Boston, Massachusettes, USA, 2002.

[8] Hirofumi Sakane, Levent Yakay, Vishal Karna, Clement Leung, and Guang R. Gao. Dimes: An iterative emulation platform for multiprocessor-system-on-chip designs. In *IEEE International Conference on Field-Programmable Technology (FPT'03)*, Tokyo, Japan, 2003.

[9] Steven J. Vaughan-Nichols. Vendors go to extreme lengths for new chips. *Computer*, pages 18–20, Jan 2004.

[10] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture* [30], pages 15–26.

[11] Christoph Scheurich and Michel Dubois. Correct memory operation of cache-based multiprocessors. In *Proceedings of the 14th Annual International Symposium on Computer Architecture*, pages 234–243, Pittsburgh, Pennsylvania, June 1987.

[12] C. E. Scheurich. *Access Ordering and Coherence in Shared Memory Multiprocessors.* PhD thesis, University of Southern California, 1989.

[13] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, 1977.

[14] James Archibald and Jean-Loup Baer. An economical solution to the cache coherence problem. In *Proceedings of the 11th Annual International Symposium on Computer Architecture* [31], pages 355–362.

[15] Larry Rudolph and Zary Segall. Dynamic decentralized cache schemes for MIMD parallel processors. In *Proceedings of the 11th Annual International Symposium on Computer Architecture* [31], pages 340–347.

[16] W. C. Brantley, K. P. McAuliffe, and J. Weiss. RP3 processor-memory element. In *Proceedings of the 1985 International Conference on Parallel Processing* [32], pages 782–789.

[17] G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton, and J. Weiss. The IBM Research Parallel Processor Prototype (RP3): Introduction and architecture. In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 764–771.

[18] Dennis Shasha and Marc Snir. Efficient and correct execution of parallel programs that share memory. *ACM TOPLAS*, 10(2):282–312, April 1988.

[19] J. R. Goodman. Cache consistency and sequential consistency. Technical Report 1006, Department of Computer Science, University of Wisconsin, Madison, February 1991.

[20] Michel Dubois, Christoph Scheurich, and Faye Briggs. Memory access buffering in multiprocessors. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 434–442, Tokyo, Japan, June 1986.

[21] Sarita V. Adve and Mark D. Hill. Weak ordering—a new definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture* [30], pages 2–14.

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

[23] R. D. Blumofe, M. Frigo, C. F. Joerg, C. E. Leiserson, and K. H. Randall. Dag-consistent distributed shared memory. In *Proc. 10th Int. Parallel Processing Symp. (IPPS'96) CD-ROM*, Honolulu, HA, April 1996. IEEE.

[24] Robert D. Blumofe, Matteo Frigo, Christopher F. Joerg, Charles E. Leiserson, and Keith H. Randall. An analysis of dag-consistent distributed shared-memory algorithms. In *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 297–308, Padua, Italy, June 1996.

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

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

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

[28] Anne Condon, Mark D. Hill, Manoj Plakal, and Daniel J. Sorin. Using lamport clocks to reason about relaxed memory models. In *Proceedings of the Fifth International Symposium on High-Performance Computer Architecture (HPCA1999)*, pages 270–278, Orlando, FL, USA, January 1999.

[29] Manoj Plakal, Daniel J. Sorin, Anne Condon, and Mark D. Hill. Lamport clocks: Verifying a directory cache-coherence protocol. In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA98)*, pages 67–76, Puerto Vallarta, Mexico, June 1998.

[30] *Proceedings of the 17th Annual International Symposium on Computer Architecture*, Seattle, Washington, May 1990.

[31] *Proceedings of the 11th Annual International Symposium on Computer Architecture*, Ann Arbor, Michigan, June 1984.

[32] *Proceedings of the 1985 International Conference on Parallel Processing*, St. Charles, Illinois, August 1985.

Appendix I

Algorithm 1. Reordering Transformation $\theta' = T(\theta, i)$
input: θ – total order; i – position of the operation being reordered
output: θ' – new total order

BEGIN
$newpos \leftarrow 1$
/* find the position ($newpos$) where the $O_\theta(i)$ should go in θ' */
while($newpos < i$) **do**
if($t_{cp}(O_\theta(i)) < t_{cp}(O_\theta(newpos)) \vee (t_{cp}(O_\theta(i)) = t_{cp}(O_\theta(newpos)) \wedge Mem(O_\theta(i)) \leq Mem(O_\theta(newpos))$)
break;
else $newpos \leftarrow newpos + 1$
end while
/* the first newpos-1 operations are kept unchanged, copy them to θ'*/
for $j \leftarrow 1$ to $newpos - 1$ do
$O_{\theta'}(j) = O_\theta(j)$
end for
$O_{\theta'}(newpos) = O_\theta(i)$ /*move the ith operation in θ to be the ($newpos$)th operation in θ' */
$j \leftarrow j + 1$
/* for all operations between $O_\theta(i)$ and $O_\theta(newpos)$, move them one operation behind, because we have inserted $O_\theta(i)$ at $newpos$ */
for j to i do
$O_{\theta'}(j) = O_\theta(j - 1)$
end for
/* copy all operations after $O_\theta(i)$ in θ into θ' */
for j to n do
$O_{\theta'}(j) = O_\theta(j)$
end for
END

Appendix II

Lemma 2: Execution (Ψ', χ_i) is equivalent to (Ψ, χ_{i-1}), after the transformation $\chi_i = T(\chi_{i-1}, i)$, which reorders $O(i)$ in χ_{i-1}.

Proof:

let's first check the second condition in **Definition 3** – all read operations in χ_i return the same results as in χ_{i-1}. Suppose the ith operation $O(i)$ in χ_{i-1} is moved backward to position

j in χ_i, $0 < j \leq i$. We consider two cases:

Case #1: $O(i)$ is a read operation;

Case #2: $O(i)$ is a write operation;

Case #1 is proved by contradiction. We assume that $O(i)$ reads a different value after it is reordered. Suppose in χ_{i-1}, $O(i)$ returns the value written by a write operation $O(k)$, as shown in figure 6, then $k < i$. It means:

$$t_{mb}(O(k)) < t_{mb}(O(i)) \vee (t_{mb}(O(k)) = t_{mb}(O(i)) \wedge Mem(O(k)) \leq Mem(O(i)))$$

Since $O(k)$ and $O(i)$ access the same memory bank, $O(k)$ must arrive earlier than $O(i)$ at the memory bank, we have:

$$t_{mb}(O(k)) < t_{mb}(O(i)) \tag{1}$$

And only when $O(i)$ is moved before $O(k)$ in χ_i can $O(i)$'s value be changed. "$O(i)$ is moved before $O(k)$" means:

$$t_{cp}(O(i)) < t_{cp}(O(k)) \vee (t_{cp}(O(i)) = t_{cp}(O(k)) \wedge Mem(O(i)) \leq Mem(O(k)))$$

that is,

$$\begin{aligned} t_{cp}(O(i)) &< t_{cp}(O(k)) \\ t_{cp}(O(i)) + C = t_{cm}(O(i)) &< t_{cp}(O(k)) + C = t_{cm}(O(k)) \end{aligned}$$

Because each memory bank has a single point $t_{cm}(O(i)) < t_{cm}(O(k))$ is equivalent to

$$t_{mb}(O(i)) < t_{mb}(O(k)) \tag{2}$$

But, (1) and (2) are in conflict, a contradiction. Thus $O(i)$ still returns the same result. Note other read operations' values cannot be changed by reordering $O(i)$, all read operations return the same values before and after the reordering.

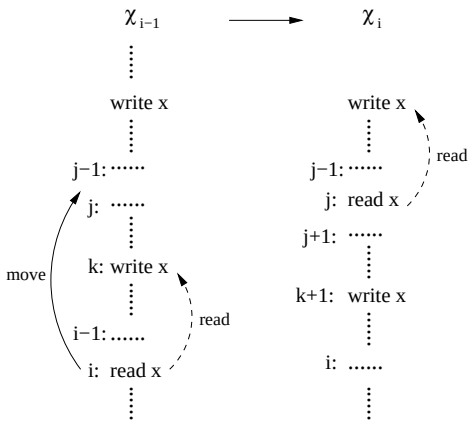
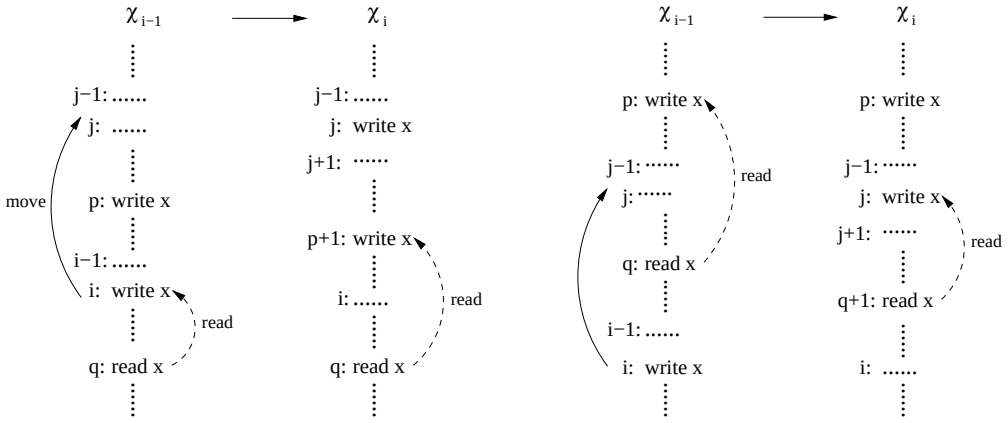


Figure 6: Case #1

For Case #2, $O(i)$ is a write. Again, proof proceeds by contradiction. We consider two sub-cases below.

Sub-Case #1: Assuming that there is a read $O(q)$ later than $O(i)$ ($q > i$) in χ_{i-1} that will read a different value after the reordering. It implies the existence of a write $O(p)$, $i > p \geq j$, as shown in figure 7(a), and that $O(q)$ reads the value written by $O(p)$ (other than $O(i)$) after the reordering. By reordering $O(i)$ to position j in χ_i, $O(i)$'s value will be killed by $O(p)$ and then $O(q)$ returns $O(p)$'s value, instead of $O(i)$. The proof of a contradiction can follow a similar argument as in Case #1 by comparing t_{mb} and t_{cp} of $O(p)$ and $O(i)$.

Sub-Case #2: Assuming that there is a read $O(q)$ earlier than $O(i)$ ($q < i$) in χ_{i-1} that will read a different value after the reordering. It implies that on χ_{i-1}, there is a write $O(p)$, $p < j$ and $j \leq q < i$, as shown in figure 7(b), and $O(q)$ returns the value written by $O(p)$. By reordering $O(i)$ to position j in χ_i, $O(i)$ will kill $O(p)$ so that $O(q)$ will return $O(i)$'s value. Again and similarly, this can generate the contradiction.



(a) Case #2 - Subcase #1

(b) Case #2 - Subcase #2

Figure 7: Case #2

Now we can draw the conclusion that in χ_i all reads return the same values as in χ_{i-1}.

Now let us look at the condition 1 in *Definition 3*. Because all reads on the first i operations in χ_i return the same values as in χ_{i-1}, any if the branch after i operations which depends on the read values in the first i operations will have the same execution path in χ_{i-1}. This guarantees that χ_i and χ_{i-1} execute the same operations, *i.e.*,$\Psi' = \Psi$.

Therefore both conditions in *Definition 3* are satisfied, thus (Ψ', χ_i) is equivalent to (Ψ, χ_{i-1}).

Proof Done.