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

A New Cache Protocol Based On The Order Free Consistency Memory Model

Chen Chen§
Joseph B Manzano†
Ge Gan†
Guang R. Gao†
Vivek Sarkar‡

CAPSL Technical Memo 082
May, 2008

§Tsinghua University
chchen00@mails.tsinghua.edu.cn
†University of Delaware
{jmanzano,gan,ggao}@capsl.udel.edu
‡Rice University
vsarkar@rice.edu

Abstract

Computer architects are now studying a new generation of chip architectures that may integrate hundreds of processing cores and memory banks on a single chip with novel interconnect technologies. A key challenge lies in the design and development of an efficient on-chip shared memory organization for these future many-core architectures. New approaches need to be developed to address this challenge by overcoming the scalability and power limitations imposed by past work on cache coherence for symmetric multiprocessors. In addition, the theoretical foundation for these new coherence protocols need to be based on memory consistency models with a well-defined and programmer-friendly operational semantics. We believe that many-core processors represent a qualitative shift from past shared-memory architectures, thereby motivating a fresh look at coherence protocols in conjunction with memory models.

In this paper, we revisit the notions of "uniprocessor dependences" and "multiprocessor memory coherence", both of which are fundamental assumptions in the Sequential Consistency (SC) model and other SC-derived memory consistency models [1]. We propose a new memory model called Order Free Consistency (OFC). OFC has a well-defined operational semantics that formalizes a weaker notion of uniprocessor dependences for memory operations than in past work and relaxes the memory coherence assumption by building on the Location Consistency (LC) memory model. We also introduce a highly parallel cache consistency protocol that satisfies the OFC model. Finally, as a proof-of-concept implementation for our proposed ideas, we include a preliminary evaluation of our OFC-based protocol in a software-controlled cache for the Cell Broadband Engine. These results demonstrate measurable improvements over software cache protocols for SC and OpenMP memory models, and also show good programmability of the OFC model for Data-Race-Free programs. We expect larger gains on future many-core processors with larger numbers of local memories and interconnects that are inherently out-of-order.

[1] The word "SC-derived" was earlier used in [14]. It implies that the memory model assumes memory coherence.

Contents

List of Figures

List of Tables

1 Introduction

Future microprocessor chip technologies are likely to head towards new generations of many-core chip architectures that contain 100 to 1,000 processing cores using a shared memory organization with a large number of on-chip memory banks. High end systems that contain many such chips (nodes), are likely to be accompanied by an increasing demand for lower latencies, higher bandwidth and support for a shared-address space among the nodes. Inter-chip and intra-chip interconnection technologies are fast progressing to meet these demands e.g., with optical inter-chip and photonics intra-chip technologies, thereby promising unprecedented bandwidth levels as well as multi-channel reordering capabilities [11, 19]. For the case of "wavelength division multiplexing" (WDM) over a single physical channel, it will be possible to establish multiple communication/data channels between two points by simply using different wavelengths for each channel. Given these trends, it is unlikely that past approaches to memory consistency models and cache coherence protocols for symmetric multiprocessors will lead to an efficient on-chip shared memory organization for future many-core architectures. New approaches need to be developed to address this challenge by overcoming the scalability limitations inherent in past approaches.

As a specific example, the large degrees of freedom in reordering memory accesses that are likely to be available in future interconnects are in stark contrast with the traditional notions of "uniprocessor dependences" and "multiprocessor memory coherence", which are fundamental assumptions in the Sequential Consistency (SC) model and other SC-derived memory consistency models. These 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 *release-consistency* (RC) model [15], 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 weaker models such as *location consistency* (LC) model [14]. Further, most SC-derived memory models have retained the memory coherence assumption, which requires that all processors observe writes to the same location in the same order [14]. As a result, compiler and architecture optimizations have traditionally stayed away from exploring reorderings of memory accesses that may violate the coherence assumption or reordering of memory accesses involved in classical data dependences. We believe that both the uniprocessor-dependence and memory-coherence assumptions need to be relaxed to enable efficient exploitation of future many-core architectures.

This paper focuses on the following fundamental questions: Can we define a memory model that is truly *asynchronous*, i.e., such that (1) memory based dependences can be more relaxed than in current uniprocessor dependence models, and (2) memory transmissions between the same source and/or destination can be performed out of order if so desired? Furthermore, can such a memory model be made *realizable* i.e., can we define an operational semantics (rather than a post-mortem semantics) for such a memory model?

To address these questions, we first formalize the problem statement (Section 2) and propose a new memory model called Order Free Consistency (OFC) that relaxes the uniprocessor-

dependence and memory-coherence assumptions (Section 3). We show that OFC has a well-defined operational semantics (Section 3.1 and 3.2) and satisfies the properties expected from a sound memory model (Section 3.3), and discuss the benefits of the OFC model (Section 3.4). We also introduce a highly parallel cache consistency protocol that satisfies the OFC model (Section 4). Finally, as a proof-of-concept implementation for our proposed ideas, we include a preliminary evaluation of our OFC-based protocol in a software-controlled cache for the Cell Broadband Engine. These results demonstrate measurable improvements over software cache protocols for SC and OpenMP memory models. Moreover, these results shows that the OFC requires no changes in the source code of Data-Race-Free applications; thus, showing that the OFC model has very high programmability (Section 5). In Section 6 we listed the related work to this paper. And finally in Section 7 we summarized our work and discussed about the future plan.

2 Problem Formulation

Emerging multi-core and many-core chip architecture are rapidly dominating the landscape of future-generation hardware. A single chip may contain tens to hundreds of processing cores and memory banks interconnected with high-bandwidth on-chip networks. These multi-core chips predominately employ a shared-memory organization. As the number of cores increase, there is an increasing urgency in addressing the challenges present in the design and development of an efficient on-chip shared memory organization. Currently proposed multi-core shared memory organizations either use cache organizations with hardware support for coherence, or use local/scratch-pad memories with software support for coherence. Regardless of whether coherence support is in hardware or software, new approaches need to be developed that can overcome the scalability limitations imposed by past work on cache coherence for symmetric multiprocessors. We believe that the theoretical foundation for these new consistency protocols needs to be based on memory consistency models with a well-defined and programmer-friendly operational semantics. Consequently, many-core processors represent a qualitative shift from past shared-memory architectures, thereby motivating a fresh look at coherence protocols in conjunction with memory models.

To address the above challenges, here are what we believe to be the main problems to be studied by this paper and follow-on research:

- **Problem 1:** Can we conceive of and design a memory model that is not restricted by the notions of "uniprocessor dependences" and "multiprocessor memory coherence", both of which are fundamental assumptions in the Sequential Consistency (SC) model and other SC-derived memory consistency models ?

- **Problem 2:** Can we introduce a new kind of cache consistency protocol under the new memory model that has clear scalability advantages over past approaches?

- **Problem 3:** Can we demonstrate feasibility of the proposed cache consistency protocol and its advantages ?

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 Problem 1 introduced in Section 2. In Section 3.1, we introduce the OFC model using operational semantics defined by an abstract machine. Then, in Section 3.2, we extend the operational semantics to handle synchronization operations. Next, we discuss the properties of the OFC model in Section 3.3. Finally, in Section 3.4, we discuss the benefits of the OFC model.

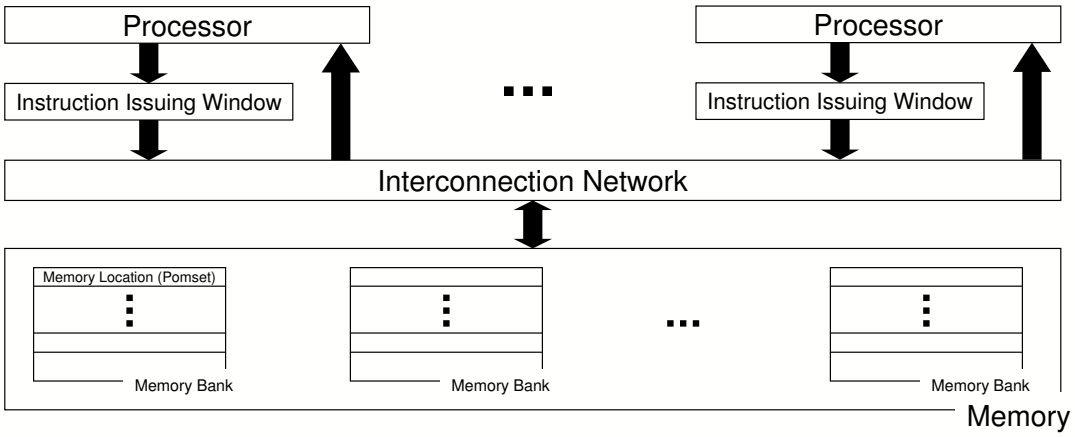

Figure 1: The OFC abstract machine architecture model

3.1 The OFC Abstract Machine and Operational Semantics

Figure 1 outlines the OFC abstract machine, which contains a number of processors and a shared memory connected by an interconnection network. Though the memory banks are shown as being separate from the processors in the figure, it is possible for memory banks and processors to be co-located in concrete realizations of this abstract machine.

Now, let us describe how a (parallel) program executes 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 sent to memory will be placed in the processor's instruction issuing window. In addition, each ready instruction carries a tag that denotes the order the instruction is fetched by the processor (i.e. the *fetch order*). Conceptually, all ready instructions from the same processor are totally ordered by their corresponding tags. However, as explained below, their issuing from the processor to the network, their arrival to the memory, and their final completion can all be out of order in general. The causes for out-of order execution in all these cases include:

- **Instruction issue:** Instructions may be issued in a different order from their fetch order.
- **Instruction transmission**: Instructions may be transmitted in a different order from their issue order, due to network congestion.
- **Instruction arrival:** Instructions may arrive out-of-order due to the presence of multiple channels between the same source-destination pairs.
- **Instruction completion:** Instructions may complete out-of-order due to (last minute) reordering opportunities in our abstract machine model.

Now 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 [14]. 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 [27]). 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.

For reader which are interested in further details of our model, including several motivating examples, please refer to [5].

3.2 Handling Synchronization Operations

Now, we explain how we handle synchronization operations introduced earlier in our program model. We give intuitions in this section. In [5] the details and examples on handling synchronization operations are given.

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 [21, 23].

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.

3.3 Properties of the OFC Model

In this section, we discuss the properties of the OFC model and their significance for memory models in general. A brief overview of each property is presented below. However, a further explanation for each property, complete with proofs, can be found in [5].

The Causal Ordering Property: This property states that if a load L dynamically gets the value written by a given store S (i.e. S *causes* L) then all other stores may perform in any order with respect to this load (i.e. L). This implies that Causal Ordering can greatly relax uniprocessor dependencies. An example is that a thread may execute a load before a dependent store [2] when there is another thread's store providing the value for the load.

The Equivalence Property: This property states that if a given program is Data-Race-Free (DRF) under Sequential Consistency, such program generates the same result (set) under both Sequential Consistency and the OFC model. This property is very important for the OFC model's programmability. This implies that any application that is DRF can be executed correctly under the OFC model without changing its source code.

The Monotonicity Property: This property states that given two versions of a parallel program, a program P and a more parallel version of P called program P', any result which is legal for P is also legal for P' under the OFC model. (A more parallel version means "in an execution that has fewer edges / constraints in the partial order of the program's execution." [14]) This property shows the robustness of our model since legal results must not be lost during parallelization transformations.

[2] Suppose there is a true-dependence between the store and the load.

The Non-intrusive Reads Property: This property states that inserting memory reads in a program will not change its result (set) under OFC. This property is very important for both cache and pre-fetching compiler optimizations. Since both of them insert memory reads in the program, either statically or dynamically, to decrease memory latencies and improve performance. Memory models without this property may produce incorrect results when using these optimizations. One example of the memory model which does not respect Non-intrusive Reads property can be found in page 37 of [12]

3.4 Benefits of the OFC Model

This section illustrates the benefits of the OFC model compared to well known memory models such as Release Consistency. The second and last benefits are unique contributions of our model, but the other benefits are (partially) achieved under a few other memory models (e.g. Location Consistency [13, 14] and the Java Memory Model [22]). This section may not include all the benefits provided by OFC. However, these benefits will help people in the field (such as computer architects, compiler writers and application developments) to understand and exploit the features of the OFC model.

Causal Ordering Increases Parallelism: Since causal ordering avoids unnecessary executing-order restrictions by either uniprocessor data dependences or inter-thread synchronizations, more instructions can be executed simultaneously. In other words, parallelism is increased.

Ability to Exploit Bandwidth Provided by Multi-Channel Networks: Recently, research is underway to investigate how to make the transition from electrons to photons in the interconnect network of a single chip. Although a production level solution is still several years away, it represents a very promising path to meet the demand for on-chip bandwidth which are projected to be up to a terabyte per second and beyond for future generation of many-core/multi-core chip architectures. [11, 19]

An important feature of optical based interconnection networks is the ability to provide multiple 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.

Portability of Some Uniprocessor Compiler Optimizations: One important challenge for compiler optimization is how to ensure simple portability of many existing uniprocessor compiler optimizations 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 [22].

This problem is partly solved by the OFC model. 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 some uniprocessor reordering constraints which are due to strict uniprocessor data dependences. In other words, 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 that the memory side may become a bottleneck. Optimizing compilers may perform static analysis which will classify and label certain instructions so their issue order from a processor should subject to the runtime load situation. If the memory side becomes congested, such annotated instruction can be delayed until the situation improves.

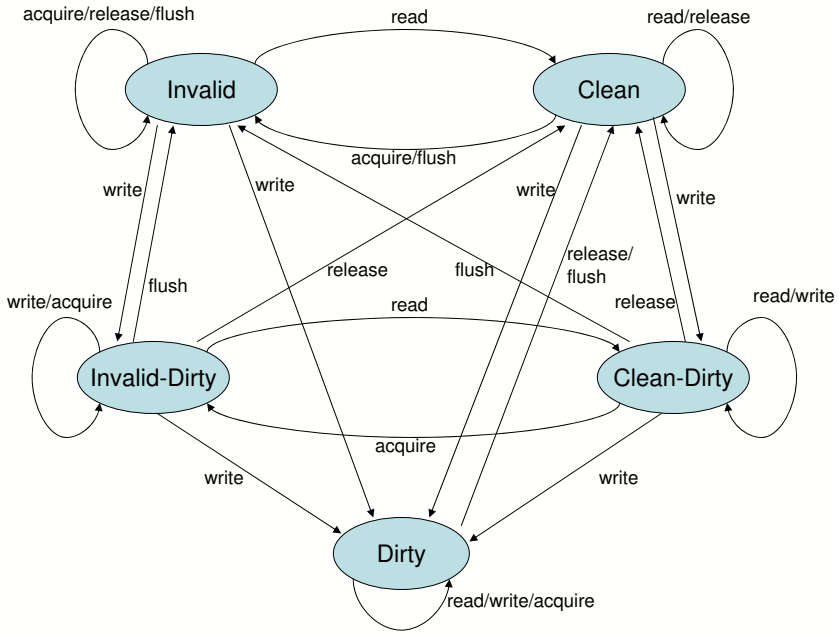

Figure 2: State transition diagram for OFC cache protocol.

4 The OFC Cache Consistency Protocol

In this section, we introduce the OFC cache consistency protocol, a new cache consistency protocol that supports the semantics of the OFC model, which presents our solution to problem 2 and 3 introduced in Section 2. As we will see, the OFC protocol does not use the traditional snooping or directory-based approaches [8]. Thus it avoids the scalability problem of snooping approach and the extra cost of directory-based approach. Moreover, by breaking cache flush

operations into finer-grain operations, our protocol gains more opportunities to work efficiently and parallelly. The details are explained in the following sections.

The OFC protocol consists of five states and five different operations which can be applied to each cache line. The state transition diagram is shown in Figure 2. Section 4.1 introduces the five cache line states. Later, Section 4.2 illustrates the cache operations and states transitions. Finally, as a case study of using the OFC cache protocol in realworld programming languages, Section 4.3 explains how to further relax the OpenMP memory model [3] by using our OFC cache protocol.

4.1 Cache Line States

The OFC cache protocol assumes that each cache line may be in one of the following five states:

Invalid: The cache line does not contain valid information.

Clean: The cache line contains valid information. Moreover, the words in the cache line have never been locally modified [3] since it was loaded from memory.

Dirty: The cache line contains valid information and every word in the cache line has been locally modified.

Clean-Dirty: The cache line contains valid information and some words in the cache line have been locally modified while others have not.

Invalid-Dirty: The cache line contains both valid and invalid information and every valid word in the cache line has been locally modified.

To maintain the cache line states, each word requires an extra bit which is called the dirty-bit. This dirty-bit indicates whether the corresponding word has been locally modified. Moreover, each cache line requires an extra bit which is called the invalid-line-bit. This invalid-line-bit indicates whether the words with unset dirty-bits are invalid or not. Therefore, the state of a cache line is determined by the dirty-bits and the invalid-line-bit as follows.

Invalid: Invalid-line-bit is set. All the dirty-bits are unset.

Clean: Invalid-line-bit is unset. All the dirty-bits are unset.

Dirty: Invalid-line-bit is either set or unset. All the dirty-bits are set.

Clean-Dirty: Invalid-line-bit is unset. Some of the dirty-bits are set.

Invalid-Dirty: Invalid-line-bit is set. Some of the dirty-bits are set.

4.2 Cache Operations

The state transitions are trigger by a group of five operations in the OFC protocol. These are read, write, release, acquire and flush. To simplify the explanation, we are assuming that

[3] A word is locally modified when a write operation changes the value of the word in the cache.

each operation affects only a single cache line at a time. In Section 4.3 we explain how these operations can be extended to deal with multiple lines. Specifically, the actions triggered by the OFC cache protocol given an operation are as follows:

Read operation We assume that a read operation always returns the values on a whole cache line. The operation triggers the following actions which depends on the cache line original state.

- **Invalid:** It is considered as a cache miss. The line will be fetched from memory and stored in the cache line. Then, the cache line state is set to Clean. After that, the values on the whole cache line will be returned.
- **Invalid-Dirty:** It is considered as a cache miss. The line will be fetched from memory to a buffer and then merged into [4] the cache line. Then, the cache line state is set to Clean-Dirty by unsetting the invalid-line-bit. Finally, the values on the whole cache line will be returned.
- **Other states:** It is considered as a cache hit. The values on the cache line are returned. The cache line state is not changed.

Write operation We assume that a write operation always writes value(s) to one or more words on a cache line. The write operation triggers the following actions which depends on the cache line original state.

- **Invalid or Invalid-Dirty:** It is considered as a cache miss. The value(s) will be written to the cache line and the corresponding dirty-bit(s) will be set. Note that the cache line state will be set to either Invalid-Dirty or Dirty, which depends on whether the values are written to all the words on the cache line.
- **Other states:** It is considered as a cache hit. The value(s) will be written to the cache line and the corresponding dirty-bit(s) will be set. Note that the state of the cache line may be either unchanged or changed to Clean-Dirty or Dirty.

Acquire operation Acquire operation implies that the thread just owns a lock for accessing some data or enter a critical section. We assume that the data to be accessed is in one cache line. It is very important to clarify that the cache acquire operation (as well as the cache release operation) will not perform any lock operation or semantic. It is compiler / programmer's responsibility to ensure the lock semantic. Moreover, if the compiler / programmer wants to implement an acquire semantic for multiple lines, only one lock semantic should be ensured, and after that multiple cache acquire operations on all the involved cache lines are performed in any order.

[4] The merge action is done by replacing the values of words with unset dirty-bits in the cache line by the values of corresponding words in the buffer.

The action of the acquire operation is done by setting the invalid-line-bit of the cache line. As a result, a Clean state will be changed to Invalid state. Moreover, a Clean-Dirty state will be changed to Invalid-Dirty state. All the other states are unchanged.

Release operation Release operation implies that the thread just releases a lock or exit a critical section. We assume that the data which was accessed is in one cache line.

- **Invalid or Clean:** Nothing to be done.
- **Dirty:** The cache line is written back to the memory. After that, the cache line state is set to Clean.
- **Clean-Dirty or Invalid-Dirty:** The line will be fetched from the memory to a buffer and then merged into the cache line. After that, the cache line is written back to the memory. Note that the fetching and writing back actions should be performed atomically, i.e. no other memory load or store on the same memory line is allowed to happen between them. Finally, the cache line state is set to Clean.

Flush operation The Flush operation implies that the thread is performing a barrier semantic. We assume that the data which was accessed is in one cache line.

- **Invalid:** Nothing to be done.
- **Clean:** The cache line state is set to Invalid.
- **Dirty:** The cache line is written back to the memory. After that the cache line state is set to Clean.
- **Clean-Dirty or Invalid-Dirty:** The line will be fetched from the memory to a buffer and then merged into the cache line. After that, the cache line is written back to the memory. Note that the fetching and writing back actions should be performed atomically. Finally, the cache line state is set to Invalid.

Cache line replacement

Cache line replacement is not considered as an independent operation in the OFC cache protocol. However it can be simply achieved by performing a Flush operation on the cache line to force the old data out, setting the cache line state to Invalid, and then doing the next operation.

4.3 Using OFC Cache Protocol to Further Relax OpenMP Memory Model

The OpenMP specification [3] defines a relaxed memory model. However such memory model can be further relaxed by using the OFC cache protocol. This section explains this process.

Note that after such further relaxation, the semantics of a program with data races may be changed. However the equivalence property of the OFC model guarantees that the semantics of Data-Race-Free programs are preserved.

Intuitively, one OpenMP basic operation (such as read, write or flush [5]) can be decomposed and replaced by one or more OFC cache operations which can be performed in any order. Therefore, after such process, two (or more) OpenMP basic operations get more opportunities to be performed simultaneously. For example, the OpenMP specification defines that if two openmp-flush operations have a non-empty intersection of their flush-sets, they must be performed as if in some sequential order. However, after they are decomposed and replaced by one or more OFC flush operations, they can be performed simultaneously. The reason is that each OFC flush operation only requires atomicity for itself. Thus the two openmp-flush operations can be performed in parallel, as long as those atomicities are not broken.

Now, it is time to explain the details of decompositions and replacements for the OpenMP basic operations. To save space, we just list some common basic operations. However, from this list it is not hard to derive the decompositions and replacements for the rest OpenMP operations. Moreover, it is always safe to replace an openmp-flush by flush operations on every cache line.

- **Openmp-read:** Read operations are performed on all the cache lines which are involved in the openmp-read, and they can be performed in any order. After that, the required values are returned.
- **Openmp-write:** Write operations are performed on all the cache lines which are involved in the openmp-write, and they can be performed in any order.
- **Barrier:** The openmp-flush is replaced by the flush operations on every cache line. The flush operations can be performed in any order.
- **Critical, Ordered, omp_set_lock and omp_unset_lock regions:** On entry, the openmp-flush is replaced by the acquire operations on every cache line. The acquire operations can be performed in any order. On exit, the openmp-flush is replaced by the release operations on every cache line. The release operations can be performed in any order.

5 Experimental Results

In this section we summarize a preliminary evaluation of our OFC-based protocol in a software-controlled cache for the Cell Broadband Engine. This implementation is designed as a proof-of-concept for our proposed ideas. Section 5.1 describes the experimental setup, Section 5.2

[5] They are denoted as openmp-read, openmp-write and openmp-flush in the rest of the section in order to distinguish from OFC cache operations.

presents the results using the number of DMA accesses as our main metric. Finally, Section 5.3 summarizes our major observations. These results demonstrate non-trivial improvements of the OFC cache protocol over software emulated cache protocols for SC and the OpenMP memory models. Moreover, these results show that the OFC requires no changes in the source code of Data-Race-Free applications; thus, showing that the OFC model has very high programmability. We expect large performance gains in future many-core processors that feature large number of local memories and interconnects that are inherently out-of-order.

5.1 Experimental Setup

We use OPELL (OPenmp for cELL) toolchain [18] as the testbed of our experiments. OPELL implements a shared memory model (i.e. OpenMP) for the Cell Broadband Engine. The shared memory model is implemented by a fully software controlled cache. Therefore, we can implement various memory models by changing the cache protocol.

We have implemented three different protocols outlined below for the software cache in our testbed: SC-MESI, OpenMP and OFC. All three software cache protocols were executed directly on the Cell hardware and produced correct results for all test and benchmark programs that we tried. Moreover all of the three cache protocols use the same benchmark source code. (i.e. We did not change any source code during when we evaluate different protocols.) The preliminary results in Section 5.2 are limited to counts of DMA operations because we do not feel that the current execution time overheads of our software cache are representative of production-strength implementations such as [9]. However, we expect the DMA operation counts to remain unchanged even when our software cache implementation becomes more efficient in the future.

5.1.1 SC-MESI Protocol

This protocol is a direct software implementation of the MESI protocol (in page 299 of [8]) that has historically been implemented in hardware to support sequential consistency on shared-memory multiprocessors. For simplicity, we implement a "virtual bus" on the Cell by maintaining a shared table in the global memory. The table records the states and tags of all cache lines of all threads, and locks are used to ensure exclusive access. When a thread intends to perform a read/write operation, it has to (a) Acquire the lock of the table, (b) Load the table from global memory, (c) Update the table, (d) Write the table back, and (e) Release the lock of the table. In this way, it is possible for one thread to modify the states of "cached" data on other threads in accordance with a snooping-based MESI protocol.

For the purpose of our experimental evaluation, we maintain a software counter for the number of non-local DMA operations that would be necessary to support the MESI protocol. The counter is incremented whenever any inter-processor communication or cache miss occurs in the MESI protocol.

5.1.2 OpenMP Protocol

This protocol is a software implementation of the OpenMP memory model [3]. In this protocol, a cache line can be in one of five states — *invalid*, *clean*, *dirty*, *invalid-dirty*, or *clean-dirty* — the same as states of the OFC cache protocol. Dirty-bits are also maintained at a word level granularity to indicate which words in the line are clean vs. dirty. A flush operation on one or more dirty words in the same line is accomplished by performing the following steps atomically — load the line from memory, merge in the dirty words, and write the line back to memory. A flush-all operation is accomplished by performing individual flush operations on all dirty lines. According to the OpenMP memory model, each flush-all operation must be completed atomically with respect to the others. A global lock is used to guarantee that the flush-all operations are performed serially, which is implemented by atomic DMA operations.

5.1.3 OFC Protocol

The OFC protocol has already been described in Section 4. The difference between the OpenMP protocol and the OFC protocol is the granularity of the flush-all operation. In OpenMP, the flush-all operations will be serialized to guarantee the strict OpenMP memory model. On the other hand, under OFC, the OpenMP flush-all operation is translated into one or more OFC acquire, release or flush operations(which depends on the type of synchronization). Thus, as pointed out in Section 4, each operation has more opportunity to execute concurrently with other operations.

The advantages of the OFC protocol are as follows:

Compared with SC-MESI the OFC protocol avoids the overhead of inter-processor communication.

Compared with OpenMP the OFC protocol avoids the overhead of global locks on each OpenMP flush-all operation.

5.2 Preliminary Results

Table 1 summarizes the DMA operation counts obtained for the SC-MESI, OpenMP and OFC protocols on four benchmark programs — RandomAccess and Stream from the HPC Challenge benchmark suite [1], Integer Sort (IS) and Embarrassingly Parallel (EP) from the NAS Parallel Benchmarks [2]. Results for IS and EP were obtained for three different input sizes — Class S, Class W, and Class A. The OFC protocol consistently led to reductions relative to SC-MESI, with reduction factors ranging from 1.53× to 13.36×. The OpenMP protocol also led to reductions relative to SC-MESI in most cases. However, in all cases the reduction for OpenMP was less than that for OFC and in two cases — NPB-IS (Class W) and NPB-IS (Class A) — the reductions were < 1, i.e., the OpenMP protocol performed a larger number of DMA operations than the SC-MESI protocol for those two cases. The main reason for this apparent anomaly is

Benchmark (Input Size)	SC-MESI DMA counts	OpenMP DMA counts	OFC DMA counts	SC/OpenMP Reduction	SC/OFC Reduction
RandomAccess (16MB)	42,173,630	31,251,332	27,643,819	1.35×	1.53×
Stream (16MB)	105,962,373	43,035,798	34,325,854	2.46×	3.09×
NPB-IS (Class S)	2,211,386	215,847	165,481	10.25×	13.36×
NPB-IS (Class W)	45,343,682	104,550,675	31,458,977	0.43×	1.44×
NPB-IS (Class A)	335,341,054	1,029,633,888	304,803,447	0.33×	1.10×
NPB-EP (Class S)	25,052,104	5,412,745	5,275,910	4.63×	4.75×
NPB-EP (Class W)	46,310,886	10,511,766	10,368,892	4.41×	4.47×
NPB-EP (Class A)	363,633,983	81,894,499	81,671,303	4.44×	4.45×

Table 1: DMA counts for SC-MESI, OpenMP, and OFC protocols

that even though the OpenMP memory model is more relaxed than the SC model, the flush-all operations in OpenMP can result in a significant amount of communication.

5.3 Major Observations

Though the results presented in Section 5.2 are still preliminary, we can draw the following initial observations from them:

- Memory consistency models can have a significant impact on the amount of inter-core communication needed for cache coherence protocols.
- In some cases, a relaxed memory model like OpenMP can result in larger communication overhead than a strong memory model like SC because of the communication overhead inherent in flush-all operations.
- For all data points presented in Section 5.2, the OFC protocol always resulted in the lowest number of communication operations.
- The results in Section 5.2 do not reflect the full benefits that the OFC model can provide when the limitations due to uniprocessor dependencies and in-order interconnect network requirements are removed.
- All of the benchmarks in our testbed have a self-verification step. During our experiments, all benchmarks passed this step under the three software-emulated cache protocols. As stated before, no changes were needed in the application source code; thus, showing the high programmability of our cache protocol / memory model.

6 Related Work

This section introduces both previous research that involves memory models, as well as, work that deals with the implementation of software cache in the CELL B.E. architecture.

6.1 Memory Model Related Work

This section discusses research that involves memory models. Several representative memory models are listed which are relevant to the OFC model. Finally, we provide a comparison between these models and ours. These models include: Gharachorloo et al. the Release Consistency (RC) [15], Keleher et al. the Lazy Release Consistency (LRC) [20], Bershad et al. the Entry Consistency (EC) [4], Gao et al. the Local Consistency (LC) [13], Shen et al. the Commit-Reconcile & Fences (CRF) model [25], Manson et al. the Java memory model [22] and Saraswat et al. the Relaxed Atomic + Ordering (RAO) model [24].

The SC-derived Models: [14] defines SC-derived models as the memory models which assume memory coherence. Memory coherence can be stated as follows [15]: all writes to the same location are serialized in some order and are performed in that order with respect to any processor. If we follow this definition, RC, LRC and EC are memory coherent models. The fundamental difference between the SC-derived models and the OFC model is that the former assumes memory coherence and the latter does 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. [24] Like LC, the OFC model does not rely on the coherence assumption. However LC model assumes that "all uniprocessor control and data dependences are satisfied." [13, 14] Therefore, 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-grained operations [25], 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 presented in the conclusion of [25], "instruction reordering is constrained only by data dependences and memory fences." Thus, the CRF model introduces more reordering restrictions than needed.

The Java memory model: The Java memory model 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." [22] However, as pointed out in [5], the Java memory model violates the monotonicity property. Moreover, as presented in [24]: "In contrast, the methodological stance of [22] 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 to the Java memory model since "RAO is generative, given a source program it generates all possible sequences of executions." [24] However, as pointed out in [5], the RAO model also violates the monotonicity property. Moreover, the term "generative" does not fully explain how a real parallel machine works under the specification of the RAO model.

6.2 OpenMP-CELL-Software-Cache Related Work

There are a number of publications discussing the implementation of software cache on IBM CELL processor [9, 10, 17], or introducing the usage of software cache in the parallel programming runtime system for the CELL processor [7, 6, 26, 16]. Until now, all the implementations of software controlled cache for CELL processor are used in the context of an OpenMP compiler toolchain, which is either developed by IBM or by a third party.

In [9, 10], a software controlled cache runtime module is introduced. This implementation uses a 4-way set-associative cache, in which each set is searched simultaneously, effectively exploiting the SIMD parallelism of the SPE instruction set. This implementation can be used for both sequential and parallel program. In [17], an asynchronous software cache implementation for CELL has been introduced. In this variant, all sub-operations in a cache access transaction (e.g. data lookup, data placement and replacement, data write back, memory synchronization and address translation) are implemented in a decoupled fashion. This provides an opportunity to organize the generated code to overlap communication and computation. In [26], a mechanism providing coherent between software cache and static buffer (which co-exist on the SPU) has been introduced, together with the associated compiler optimizing techniques. However, such coherent mechanism is only local to each individual SPE, not across all SPEs.

The OPELL (OPenmp for cELL) toolchain [18] is an effort to implement a shared memory model for the CBE architecture. It introduces a software cache module, which can be used as a basic framework for implementing various cache coherence protocols.

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 by 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, to make the memory model realizable, we defined a set of operational semantics (rather than a post-mortem semantics) for such memory model. We argued that our memory model satisfies the new generation of multi-core chip architectures and takes advantage of the optical inter-chip and photonics intra-chip interconnection technologies. Besides this, it also allows the exploitation of many reordering opportunities that were hidden before due to unnecessary constrains imposed by uniprocessor data dependences. Furthermore, we proposed a highly parallel cache consistency protocol that satisfies the OFC model. As a case study, we also explained how to use the OFC cache consistency protocol to further relax the OpenMP memory model which one of the most well known shared memory programming models nowadays.

Finally, as a proof-of-concept implementation for our proposed ideas, we include a preliminary evaluation of our OFC-based protocol in a software-controlled cache for the Cell Broadband Engine. these results shows that the OFC requires no changes in the source code of Data-Race-Free applications; thus, showing that the OFC model has very high programmability.

We expect large performance gains in future many-core processors that feature large number of local memories and interconnects that are inherently out-of-order, which is the next step of our study.

Acknowledgement

This work was conducted at Computer Architecture and Parallel System Lab (Director: 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] *HPC Challenge Benchmark*. http://icl.cs.utk.edu/hpcc/.

[2] *NAS Parallel Benchmark*. http://www.nas.nasa.gov/Resources/Software/npb.html.

[3] *OpenMP Application Program Interface*, 2005. http://www.openmp.org/mp-documents/spec25.pdf.

[4] 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.

[5] C. Chen, J. Manzano, W. Chen, and G. Gao. Order free consistency: Towards a fully asynchronous memory model. Technical report, November 2007. http://www.capsl.udel.edu/publications.shtml#3.

[6] Tong Chen, Zehra Sura, Kathryn M. O'Brien, and John K. O'Brien. Optimizing the use of static buffers for DMA on a CELL chip. In George Almási, Calin Cascaval, and Peng

Wu, editors, *LCPC*, volume 4382 of *Lecture Notes in Computer Science*, pages 314–329. Springer, 2006.

[7] Tong Chen, Tao Zhang, Zehra Sura, and Mar Gonzales Tallada. Prefetching irregular references for software cache on cell. In *CGO '08: Proceedings of the sixth annual IEEE/ACM international symposium on Code generation and optimization*, pages 155–164, New York, NY, USA, 2008. ACM.

[8] Jaswinder Pal singh David E. Culler and Anoop Gupta. *Parallel computer Architecture A HARDWARE / SOFTWARE APPROACH*. Morgan Kaufmann Publishers, Inc., 1999.

[9] A. E. Eichenberger et al. Using advanced compiler technology to exploit the performance of the cell broadband engine tm architecture. *IBM Systems Journal*, 45(1):59–84, 2006.

[10] Alexandre E. Eichenberger et al. Optimizing compiler for the cell processor. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 161–172, Washington, DC, USA, 2005. IEEE Computer Society.

[11] 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.

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

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

[14] 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.

[15] 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.

[16] J. K. O'Brien, K. M.O'Brien, Zehra Sura, Tong Chen and Tao Zhang. Support openmp on cell. In *International Workshop on OpenMP (2007)*, Beijing, China, 2007.

[17] Jairo Balart, Marc Gonzalez, Xavier Martorell, Eduard Ayguade, Zehra Sura, Tong Chen, Tao Zhang, Kevin O'brien, Kathryn O'brien. A novel asynchronous software cache implementation for the cell-be processor. In *The 20th International Workshop on Languages and Compilers for Parallel Computing*, Urbana, Illinois, 2007.

[18] Yi Jiang Joseph Manzano, Ziang Hu and Ge Gan. Towards an automatic code layout framework. In *IWOMP '07: Proceedings of the International Workshop on OpenMP (2007)*, Beijing, China, 2007.

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

[20] 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.

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

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] Tong Chen, Haibo Lin, Tao Zhang, Kathryn O'Brien, Kevin O'Brien. Orchestrating data transfer for the cell/b.e. processor. In *22nd ACM International Conference on Supercomputing*, Island of Kos, Greece, 2008.

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