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

A Study of Different Instantiations of the OpenMP Memory Model and Their Software Cache Implementations

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

CAPSL Technical Memo 086
January, 2009

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

Abstract

An important open problem for future many-core chip architectures is the development of shared-memory organizations and memory consistency models that are effective for small local memory sizes per core, scalable to a large number of cores, and still productive for software to use. Many multicore processors, such as the Cell Broadband Engine, Tilera, and Cyclops64, include the use of software-managed local memories that avoid the known power and scalability limitations of hardware-managed cache structures. OpenMP is a natural candidate as a programming model for multicore processors with software-managed local memories, thanks to its weak memory consistency model. The OpenMP memory model allows each thread to maintain a *temporary view* of the shared memory, and includes a *flush* operation that can be used to synchronize the temporary view with the shared memory.

In this paper, we address the problem of software cache implementations for the OpenMP memory model on multicore processors. We first formalize the idealized OpenMP memory model (*Model*IDEAL) that assumes unbounded space for temporary views, and then formalize three practical instantiations — *Model*GF (based on nondeterministic *global flushes*), *Model*LF (based on nondeterministic *local flushes*), and *Model*RLF (a further refinement of *Model*LF 's flush operations). We introduce corresponding cache protocols for the three instantiations. Performance evaluations of these protocols in software cache implementations for Cell show the following results: i) the cache protocol based on *Model*LF consistently outperforms the protocol based on *Model*GF , ii) this performance gap increases as the size of the local memory per core decreases. Our conclusion is that the OpenMP's relaxed memory model with temporary views is a good match for software cache implementations, and that the refinements in *Model*LF and *Model*RLF can lead to good opportunities for scalable OpenMP implementations on future multicore processors.

1 Introduction

An important open problem for future multicore chip architectures is the development of shared-memory organizations and memory consistency models that are effective for small local memory sizes per core, scalable to a large number of cores, and still productive for software to use. Despite the fact that strong memory models such as Sequential Consistency (SC) are supported on mainstream small-scale SMPs, it seems likely that weaker memory models will be explored in current and future multicore architectures such as the Cell Broadband Engine [1], Tilera [5], and Cyclops64 [15].

OpenMP is a natural candidate as a programming model for multicore processors with software-managed local memories, thanks to its weak memory consistency model. The OpenMP model allows each thread to maintain a *temporary view* of the shared memory which "allows the thread to cache variables and thereby avoid going to the memory for every reference of a variable". It includes a *flush* operation that can be used to synchronize the temporary view with the shared memory. It is a weak consistency model "because a thread's temporary view is not required to be consistent with memory at all times". This relaxation of the memory consistency constraints provides room for computer system designers to experiment with a wide range of caching schemes, each of which with different performance and cost tradeoffs.

In this paper, we address the problem of software cache implementations for the OpenMP memory model on multicore processors. OpenMP is an attractive alternative to lower level programming models for multicore processors with local memories, such as the Cell SDK in which the programmer explicitly manages memory-to-memory transfers. A faithful implementation of the OpenMP model requires each core to maintain a private copy of the entire shared memory, which is unrealistic especially for the small amount of memory available per core in current and future multicore processors. It is a requirement for any scalable software cache implementation to work with small cache sizes *e.g.,* the local memory size per SPE in the Cell processor is only 256KB. Therefore, we first formalize the idealized OpenMP memory model (*Model*IDEAL) that assumes unbounded space for temporary views, and then formalize three practical instantiations — *Model*GF (based on nondeterministic *global flushes*), *Model*LF (based on nondeterministic *local flushes*), and *Model*RLF (a further refinement of *Model*LF 's flush operations).

We conducted a performance evaluation of these protocols in a software cache implementation for the Cell processor based on the OPELL (OpenMP on CELL) framework [27]. Our experimental results are as follows: i) the cache protocol based on *Model*LF consistently outperforms the protocol based on *Model*GF , ii) this performance gap increases as the size of the local memory per core decreases. The impact of a small cache size on cache miss/hit ratios is well known. The new finding in this paper shows that a small cache size can also increase the rate of cache line eviction depending on the memory consistency model and cache protocol assumed, which is the motivation for studying these instantiations. As the size of available on chip memory space per core is getting smaller as the number of cores increasing - this finding demonstrates the increasing importance on the study of efficient memory models and cache protocols. Our conclusion is that the OpenMP's relaxed memory model with temporary views is a good match for software cache implementations, and that the refinements done under *Model*LF and *Model*RLF can lead to good opportunities for scalable implementations of OpenMP on future multicore processors.

The rest of the paper is organized as follows. Section 2 introduces the four instantiations of OpenMP memory models. Section 3 introduces the cache protocols which implement the models. Section 4 presents the experimental results. Section 5 discusses the related work. The conclusion is presented in Section 6.

2 Formalization of the OpenMP Memory Model Instantiations

A necessary prerequisite to build OpenMP's software cache implementations is the availability of formal memory models that establish the legality conditions for determining if an implementation is correct. As observed in [12], "it is impossible to verify OpenMP applications formally since the prose does not provide a formal consistency model that precisely describes how reads and writes on different threads interact". While there is general agreement that the OpenMP

memory model is based on *temporary views* and *flush* operations[1], discussions with OpenMP experts led us to conclude that the OpenMP specification provides a lot of leeway on when *flush* operations can be performed and on the inclusion of additional flush operations (not specified by the programmer) to deal with local memory size constraints. As we will see, this leeway can lead to a family of OpenMP memory models with different semantics and performance trade-offs.

In this section, we formalize four instantiations of the OpenMP Memory Model — *Model*IDEAL , *Model*GF , *Model*LF , and *Model*RLF . All four instantiations build on OpenMP's relaxed-consistency memory model in which each worker thread maintains a *temporary view* of shared data which may not always be consistent with the actual data stored in shared memory. The OpenMP *flush* operation is used to establish consistency between these temporary views and the shared memory at specific program points; furthermore, all flush operations for a given datum must be serialized.

*Model*IDEAL assumes that each thread has sufficient memory available to make a full copy of the address space, so that flush operations are only performed at the program points designated by the programmer. The other three instantiations assume that additional flush operations may be inserted nondeterministically between programmer-specified flush operations:

- *Model*GF has *global flush* semantics which force all temporary views to be synchronized with the shared memory.
- *Model*LF has *local flush* semantics which only force the local temporary view to be synchronized with the shared memory.
- *Model*RLF extends the flush operation of *Model*LF to support three types of refined flush operations. Each one has weaker semantics than *Model*LF , but may be implemented more efficiently.

Another important difference between *Model*IDEAL and the other three instantiations is that under *Model*IDEAL , a flush operation may be applied on a set of shared locations. However, in the other three, a flush operation is only applied on a single location. We assume that a memory access on a single location is always atomic. Therefore, under these three instantiations, the serializability requirement of flush operations is naturally satisfied. A flush operation on a set of shared locations is decomposed into unordered flush operations on each individual locations.

2.1 *Model*IDEAL

In this section, we formalize the memory model for an *idealized* version of the "temporary view" introduced in the OpenMP Memory Model. The main idealization is that each thread is assumed to have sufficient memory available to make a full copy of the address space if needed, so that flush operations are only performed at the program points designated by the

[1]Flush operations may also be implicit in synchronization operations such as barriers.

programmer and no additional flush operations need to be performed due to limited buffer space.

A store, σ, is a mathematical representation of the machine's shared memory, which maps memory location addresses to values ($\sigma : addr \mapsto val$). We model temporary views by introducing a distinct store, σ_i, for each worker thread T_i in an OpenMP parallel region. Following OpenMP's convention, thread T_0 is assumed to be the master thread. $\sigma_i[l]$ represents the value stored in location l in thread T_i's temporary view. The flush operation, $flush(\sigma_i, \sigma)$ makes temporary view σ_i consistent with the shared view σ. As in OpenMP, we assume that all flush operations with a non-empty intersection of flush-sets are *serialized i.e.,* are observed by all threads to be completed in the same sequential order.

The operational semantics of memory operations in *Model*IDEAL is as follows:

- *Memory read* — If thread T_i needs to read the value of the location L, it performs a $read(T_i, L)$ operation on store σ_i. If T_i's temporary view does not contain any value of L, the value in the shared memory will be loaded to the temporary view and returned to the read operation.

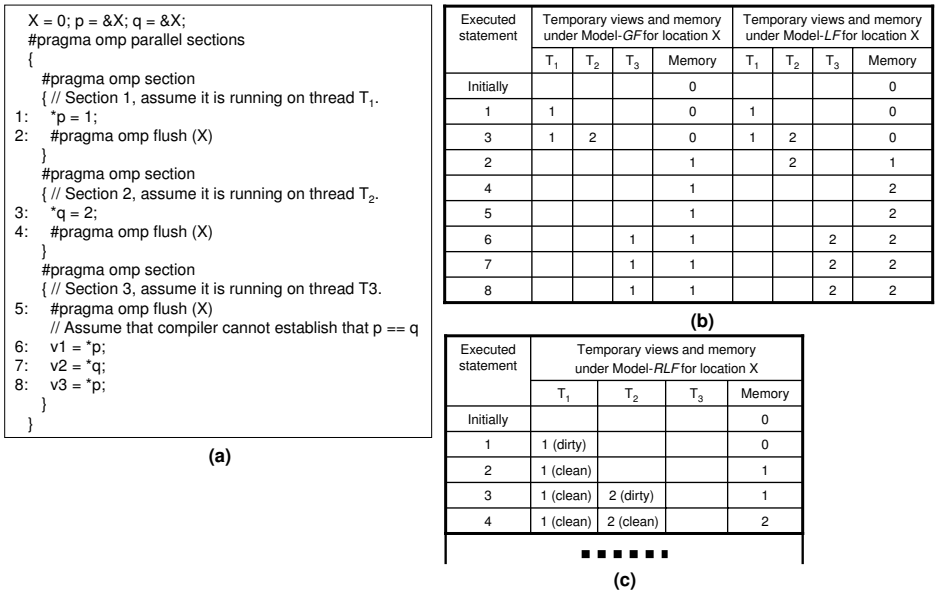
- *Memory write* — If thread T_i needs to write value v to the location L, it performs a $write(T_i, v, L)$ operation on store σ_i.

- *Program Flush* — If thread T_i needs to synchronize its own temporary view with the memory on a subset S of all the shared locations, it performs a $flush(T_i, S)$ operation. For any location L in S, if T_i's temporary view contains a "dirty value" of L, it will write back the value into memory. Here the term "dirty value" means that one of T_i's write operations modified the location in the temporary view, and the value has not been written back into memory as yet[2]. We also use the term "clean value" to represent a value that was read but not written. After the flush operation, T_i will discard all the values whose locations are in S.

We use the example OpenMP code fragment in Figure 1 (a) to illustrate the instantiations of the OpenMP Memory Model discussed in this paper. It contains a single `parallel sections` construct with three sections, all of which perform read/write accesses on a single shared location `X` through pointers `p` and `q`. We will assume that the compiler cannot establish statically that $p == q$ in this example. (If necessary, we can make the example more complicated with additional assignments to `p` and `q` to make this assumption more convincing, but we chose to avoid adding gratuitous clutter to the example.)

Let us focus our attention on the execution of Parallel Section 3 which performs three read operations on location X. In *Model*IDEAL , all three reads are guaranteed to return the same value (0, 1 or 2) since no write operations or flush operations occur between the reads.

[2]See page 15 of the OpenMP specification 3.0 [32].



```
X = 0; p = &X; q = &X;
#pragma omp parallel sections
{
  #pragma omp section
  { // Section 1, assume it is running on thread T1.
1:  *p = 1;
2:  #pragma omp flush (X)
  }
  #pragma omp section
  { // Section 2, assume it is running on thread T2.
3:  *q = 2;
4:  #pragma omp flush (X)
  }
  #pragma omp section
  { // Section 3, assume it is running on thread T3.
5:  #pragma omp flush (X)
    // Assume that compiler cannot establish that p == q
6:  v1 = *p;
7:  v2 = *q;
8:  v3 = *p;
  }
}
```

(a)

Executed statement	Temporary views and memory under Model-*GF* for location X				Temporary views and memory under Model-*LF* for location X			
	T_1	T_2	T_3	Memory	T_1	T_2	T_3	Memory
Initially				0				0
1	1			0	1			0
3	1	2		0	1	2		0
2				1		2		1
4				1				2
5				1				2
6			1	1			2	2
7			1	1			2	2
8			1	1			2	2

(b)

Executed statement	Temporary views and memory under Model-*RLF* for location X			
	T_1	T_2	T_3	Memory
Initially				0
1	1 (dirty)			0
2	1 (clean)			1
3	1 (clean)	2 (dirty)		1
4	1 (clean)	2 (clean)		2
▪▪▪▪▪▪				

(c)

Figure 1: Code example for the four instantiations. (a) Example OpenMP code fragment with three parallel sections. (b) Comparing *Model*GF with *Model*LF . The execution order is 1-3-2-4-5-6-7-8. (c) Status of temporary views and shared memory under *Model*RLF (assuming that statements 2 and 4 are release operations).

2.2 *Model*GF

In this section, we formalize *Model*GF . First, we define the program global flush operation as follows.

- *Program Global Flush* — If thread T_i needs to synchronize its own temporary view with the memory and all the other temporary views on a shared location L, it performs a $g\_flush(T_i, L)$ operation. If T_i's temporary view contains a "dirty value" of L, it will write back the value into memory. Moreover, if any other temporary view contains a clean or dirty value of L, that value will be discarded.

In *Model*GF , program global flush operations are performed at the program points specified by the programmer. Moreover, additional global flush operations may be inserted nondeterministically by the implementation at any program point, which makes it possible to implement the memory model with bounded space for temporary views. The operational semantics of memory operations in *Model*GF include the *read*, *write*, *program global flush* operations and an additional *nondeterministic global flush* operation defined as follows:

- *Nondeterministic Global Flush* — Any thread T_i may choose to perform a $g\_flush(T_i, L)$ operation nondeterministically at any program point, for a shared location L. The nondeterministic global flushes must be serialized with the program global flushes *i.e.,* all

program global flushes and nondeterministic global flushes on the same shared location must be observed by all threads to be completed in the same sequential order.

To see the difference between *Model*IDEAL and *Model*GF , let us revisit the code example in Figure 1, but now under *Model*GF semantics. It may allow some outcomes which are not allowed under *Model*IDEAL *e.g.,* `v1 = 1`, `v2 = 2`, `v3 = 2` may be obtained if a nondeterministic global flush is inserted by the implementation between statements 6 and 7.

2.3 *Model*LF

The problem with *Model*GF is that every single global flush operation requires all threads to participate, which can become a scalability issue. To solve this problem, we formalize *Model*LF in this section. Like *Model*IDEAL , it only requires that the shared memory and the owner thread participate in a flush operation. Moreover, like *Model*GF , it also allows flush operations to be inserted nondeterministically by the implementation at any program point, which makes it possible to implement the memory model with bounded space for temporary views.

The operational semantics of memory operations in *Model*LF include the *read* and *write* operations introduced in Section 2.1 and the new *program local flush* and *nondeterministic local flush* operations defined as follows:

- *Program / Nondeterministic Local Flush* — If thread T_i needs to synchronize its own temporary view with the memory on a shared location L, it performs a $l\_flush(T_i, L)$ operation. If T_i's temporary view contains a "dirty value" of L, it will write back the value into memory. After the flush operation, T_i's temporary view will discard the value of L. A thread performs program local flush operations at program points specified by programmer, and can nondeterministically perform local flush operations at any program point. All the program and nondeterministic local flush operations on the same shared location must be observed by all threads to be completed in the same sequential order.

Let us revisit the code example in Figure 1 (a) to see the difference between *Model*GF and *Model*LF . Suppose that the execution order of the statements is 1-3-2-4-5-6-7-8 and no nondeterministic flush occurs. Under *Model*GF , the result will be `v1 = 1`, `v2 = 1`, `v3 = 1`. However, under *Model*LF , the result will be `v1 = 2`, `v2 = 2`, `v3 = 2`. Figure 1 (b) shows the status of temporary views and shared memory for location X under the two models at each step.

2.4 *Model*RLF

An extension to the temporary view models is to support finer-grain flush operations *e.g.,* one type of flush operation may write back the "dirty value" into memory, but not discard the value in the temporary view.

In this section, we introduce *Model*RLF which is an extension for *Model*LF to support finer-grain flush operations. *Model*RLF has three types of program flush operations — *acquire*, *release* and *barrier* — which can be used instead of programmer-inserted flush operations.

The operational semantics of memory operations in *Model*RLF include the *read*, *write*, and *nondeterministic local flush* operations, which are the same as those in *Model*LF , and additional operations, *i.e., program acquire*, *program release*, and *program barrier* operations, which are introduced below:

- *Program Acquire* — A thread T_i can perform an $acquire(T_i, L)$ operation on a shared location L. If T_i's temporary view contains a "clean value" of L, the value will be discarded. An acquire operation is performed when the thread owns a lock for accessing some data or enters a critical section.

- *Program Release* — A thread T_i can perform a $release(T_i, L)$ operation on a shared location L. If T_i's temporary view contains a "dirty value" of L, the value will be written back into memory. After that, it will be set as a "clean value". A release operation is performed when the thread releases a lock or exits a critical section.

- *Program Barrier* — A thread T_i can perform a $barrier(T_i, L)$ operation on a shared location L. The semantics of a barrier operation is equivalent to an acquire operation followed by a release operation on location L. A barrier operation is performed when the thread is performing a barrier semantic.

The situations of using different types of flush operations in OpenMP are as follows:

- Critical, Ordered, omp set lock and omp unset lock regions: On entry, a number of acquire operations on participating shared locations will be performed. On exit, a number of release operations on participating shared locations will be performed.

- OpenMP-Barrier: A number of barrier operations on participating shared locations will be performed.

Let us revisit the code example in Figure 1 (a) to see how *Model*RLF works. First, we have to replace each flush operations in the example by one of the three types of finer-grain flush operations. Suppose we replace statement 2 by a release operation on X, 4 by release, and 5 by acquire. Then assuming that the execution order of the first four statements is 1-2-3-4. As we can see in 1 (c). T_1 and T_2's temporary views keep "clean value" of X after each own flush operation, respectively. Therefore, these values can be used for future read operations of T_1 and T_2 on X. However, under *Model*LF , a flush operation will always discard the value in the temporary view.

2.5 Properties of the Four Instantiations

In this section, we claim that the four instantiations have the following properties.

1. ***Model*$_{\text{GF}}$ becomes equivalent to *Model*$_{\text{LF}}$ if no two temporary views contain values for the same location at the same time.**

   Proof hint: The difference between the definitions of *Model*$_{\text{GF}}$ and *Model*$_{\text{LF}}$ is that the former model's flush operation has global flush semantic but the latter one has local flush semantic. However, if no two temporary views contain values for the same location at the same time, the global flush semantic is equivalent to the local flush semantic. So the two definitions are equivalent, too. Therefore, *Model*$_{\text{GF}}$ becomes equivalent to *Model*$_{\text{LF}}$.

2. ***Model*$_{\text{IDEAL}}$, *Model*$_{\text{GF}}$, *Model*$_{\text{LF}}$ and *Model*$_{\text{RLF}}$ yield the same semantics for Data-Race-Free programs.**

   Proof hint: First of all, we prove that for Data-Race-Free programs, removing non-deterministic flush operations will not change the possible outcomes. The reason is that if one non-deterministic flush operation write back a "dirty value" from the temporary view of one thread (T_i) to the memory, no other thread can see the value before T_i performing a programmer-specified flush operation on the same location of the value. Otherwise, there is a data race. So a non-deterministic flush operation on a "dirty value" will never change the outcomes of a Data-Race-Free program. If one non-deterministic flush operation discards a "clean value" in the temporary view of one thread (T_i), the most close future read of T_i on the same location will get the same value from memory. Otherwise, there is either a data race or a programmer-specified flush operation on the same location between the read and the non-deterministic flush operation. So a non-deterministic flush operation on a "clea value" will never change the outcomes of a Data-Race-Free program. Therefore, in the following part of the proof we assume that no non-deterministic flush operation occrus.

   Now we prove that *Model*$_{\text{IDEAL}}$ is equivalent to *Model*$_{\text{LF}}$. For Data-Race-Free programs, the difference between *Model*$_{\text{IDEAL}}$ and *Model*$_{\text{LF}}$ is that under *Model*$_{\text{IDEAL}}$ all the flush operations must be serialized, but under *Model*$_{\text{LF}}$ only the flush operations on the same location should be serialized. However, the two requirements of the two models are equivalent for Data-Race-Free programs. The reason is that under *Model*$_{\text{IDEAL}}$, it is impossible that two flush operations have intersection of their flush-sets and at least one location in the intersection contains "dirty value". Otherwise, there is a data race. So the two flush operations can be performed simultaneously. Therefore, the two requirements of the two models are equivalent for Data-Race-Free programs. So *Model*$_{\text{IDEAL}}$ is equivalent to *Model*$_{\text{LF}}$.

   Then we prove that *Model*$_{\text{GF}}$ is equivalent to *Model*$_{\text{LF}}$. We have already that *Model*$_{\text{GF}}$ becomes equivalent to *Model*$_{\text{LF}}$ if no two temporary views contain values for the same location at the same time. For Data-Race-Free programs, the only case that two temporary

views contain values for the same location is that both of the two values are "clean values". In this case, replacing global flush operation by local flush operation will not change the possible outcomes of the program because flush a "clean value" will not change the value of the memory. So *Model*GF is equivalent to *Model*LF .

Finally, we prove that *Model*LF is equivalent to *Model*RLF . In Section , we have explained the situations of using different types of flush operations in OpenMP. From the explanation, we can see that the acquire and release operations are always well paired. We first consider the case that every memory accesse is in one of those acquire-release pairs. In this case, a release operation is followed by an acquire operation with no memory access between them. (The only exception is the last release operation, but we can add a following acquire operation because it will not change the outcomes of the program.) From the definition of acquire and release operations, we can see that if a release operation is followed by an acquire operation, they together is equivalent to a local flush operation. So in this case *Model*LF is equivalent to *Model*RLF . The other case is that there are some memory accesses outside the acquire-release pairs. In this case, because the program has no data race, one thread can not see the such memory accesses in another thread. In other words, we can move such memory accesses into some acquire-release pair without changing the possible outcomes of the program. So this case becomes the same as the previous case. Therefore, *Model*LF is equivalent to *Model*RLF . In the proof, we did not discuss barrier operation because it can be viewed as an acquire operation followed by a release operation.

The above properties indicate that it can be possible to implement a memory model which achieves good performance without losing programmability. For example, *Model*IDEAL seems to be the easiest memory model to understand, but the hardest to implement efficiently. However, the more relaxed instantiations, *Model*GF , *Model*LF or *Model*RLF , promise to be more efficient while still giving programmers the illusion that they are working with *Model*IDEAL for programs with no data races.

3 Cache Protocols of *Model*GF , *Model*LF and *Model*RLF

In this section, we introduce the cache protocols that implement *Model*GF , *Model*LF and *Model*RLF . We assume that each thread contains a cache which corresponds to its temporary view. Therefore, performing operations on the temporary view is equivalent to performing such operations on the caches. Without loss of generality, in this section, we assume that each operation is performed on one cache line. The reason is that an operation on one cache line can be decomposed into sub operations; each of which is performed on a single location. We use per-location dirty bits in a cache line to take care of the false sharing problem.

3.1 Cache Line States

We assume that each cache line contains multiple words. Each word can contain a "clean value", a "dirty value", or no value (i.e. invalid). In all three cache protocols, each cache line can be in one of five states which are addressed as follows.

Invalid: All the words of the cache line are invalid. For convenience, if a cache line does not exist in the cache, we also say that the state of the cache line is invalid.

Clean: All the words of the cache line contain "clean values".

Dirty: All the words of the cache line contain "dirty values".

Clean-Dirty: Some words of the cache line contain "clean values". The others contain "dirty values".

Invalid-Dirty: Some words of the cache line contain no value. The others contain "dirty values".

3.2 Cache Protocol of *Model*GF

In this section, we introduce the *Model*GF cache protocol. As we explained in Section 2.2, a global flush operation may require all threads to participate. Therefore, in this protocol, we assume that there is a centralized directory which contains the information for all the caches. When a global flush operation is performed, the centralized directory can be looked up to find which caches are involved in the global flush operation. We also assume that the centralized directory is "idealized", which means that the cost of cache information maintaining and lookup is trivial. However, for a global flush operation, the cost of the communication between the caches and the centralized directory cannot be ignored.

3.2.1 Cache Operations and State Transitions

The state transition diagram of *Model*GF cache protocol is shown in Figure 2. Because both *Model*GF and *Model*LF cache protocols use the same state transition diagram, in the figure the term "flush" has different meanings. Under *Model*GF , this flush is the global flush. On the other hand, under *Model*LF , this flush represents the local flush operation.

Next, we explain how each cache operations affects the state transition diagram.

- **Read:** If the original state of the cache line is invalid or invalid-dirty, the invalid words of the cache line will load the "clean values" from memory. Therefore, the state will change to clean or clean-dirty, respectively. In other cases, the state will not change. After that, the values in the cache line will be returned.
- **Write:** A write operation writes specified "dirty values" to the cache line. Therefore, if the original state is invalid or invalid-dirty, it becomes either invalid-dirty or dirty

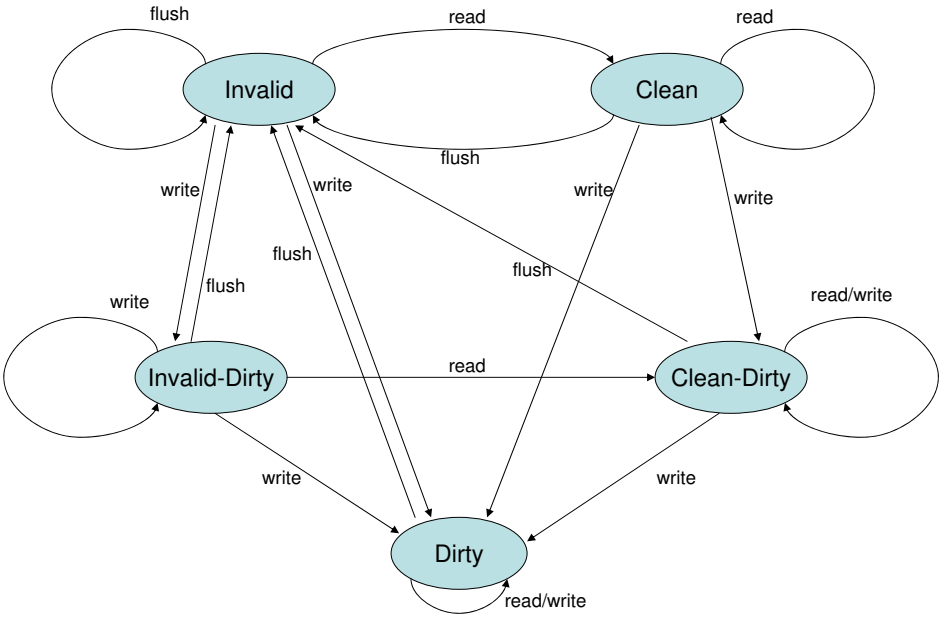


Figure 2: State transition diagram for the cache protocol of *Model*GF and *Model*LF .

after a write operation, which depends on whether all the words of cache line contain "dirty values". Similarly, when the line contains "clean values", the state becomes either clean-dirty or dirty.

- **Program / Nondeterministic Global Flush:** A flush operation forces all the "dirty values" of the cache line to be written back into memory. Then, the cache line will become invalid. After that, the centralized directory will be looked up to find all the other cache lines involved in the global flush operation. For each word of such cache lines, if the corresponding word in the original cache line (on which the flush operation performs) contained a "dirty value", the word will be set to invalid. Moreover, because none of the five cache line states allows a cache line to contain some "clean values" and some invalid words at the same time, all words which contain "clean values" will be set invalid too.

3.3 Cache Protocol of *Model*LF

The significant difference between *Model*GF cache protocol and *Model*LF cache protocol is that under *Model*LF cache protocol a flush operation does not require to inform other threads. Therefore, a centralized directory is not needed.

The state transition diagram of *Model*LF cache protocol is also shown in Figure 2. The state transition rules for read and write operations are the same as those under *Model*GF cache protocol. The state transition rule for program / nondeterministic local flush operation is defined as follows.

- **Program / Nondeterministic Local Flush:** A flush operation forces all the "dirty values" of the cache line to be written back into memory. Then, the cache line will become invalid.

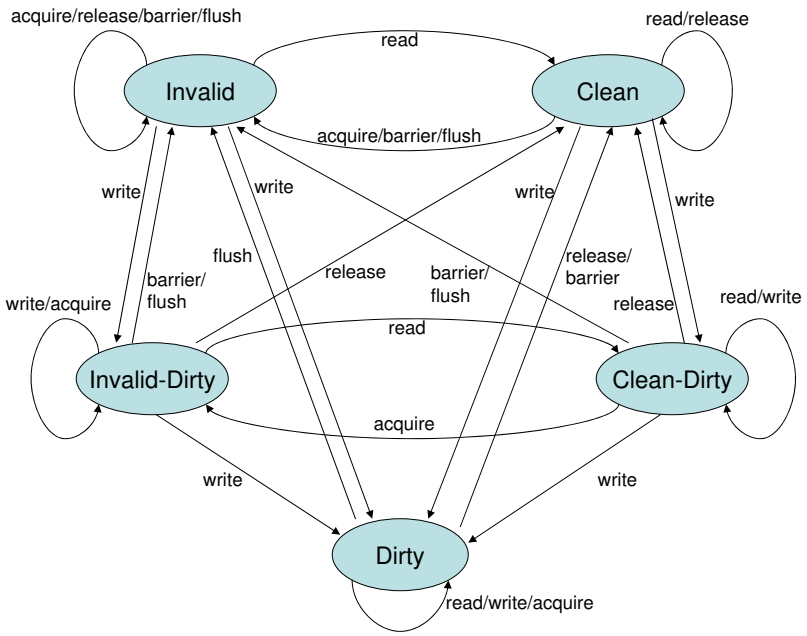


Figure 3: State transition diagram for the cache protocol of *Model*RLF .

3.4 Cache Protocol of *Model*RLF

In this section, we introduce *Model*RLF cache protocol. This cache protocol supports six cache operations, *i.e., read*, *write*, *nondeterministic local flush*, *program acquire*, *program release* and *program barrier*. The read, write and nondeterministic local flush operations have the same state transition rules as those in the other two cache protocols. The barrier operation can be performed as an acquire operation followed by a release operation. Thus, we only introduce the state transition rules for acquire and release operations. The complete state transition diagram is shown in Figure 3.

- **Program Acquire:** An acquire operation sets the words with "clean values" to be invalid. Therefore, if the cache state is clean or clean-dirty, the state will change to invalid or invalid-dirty, respectively. In other cases, the state will not change.

- **Program Release:** An release operation write back the "dirty values" of the cache line into memory. However, those values will still stay in the cache as "clean values". Therefore, if the cache state is dirty or clean-dirty, the state will change to clean. But if the cache state is invalid-dirty, the state will change to invalid because no cache state in

the protocol can represent a cache line which has both "clean values" and some invalid words. In other cases, the state will not change.

4 Experimental Results and Analyses

In this section, we introduce our experimental results under *Model*GF , *Model*LF and *Model*RLF cache protocols. In section 4.1 we introduce the experimental testbeds. Then in section 4.2 we introduce the major observations of our experiments. Finally, we introduce the details and analyses of the observations in the last three sections.

4.1 Experimental Testbeds

The experimental results presented in this paper were obtained for the Cell Broadband Engine (CBE). The framework used to test these software cache protocols is the OPELL (OPenmp for cELL) [27] which is an open source toolchain / runtime effort to implement OpenMP for the CBE. This framework is composed on four components, each of which can be modified to test several research ideas, like our software cache protocols.

We implemented *Model*GF , *Model*LF and *Model*RLF cache protocols. All the three cache protocols use 4-way set associative caches. The size of each cache line is 128 bytes. We ran the experiments on various cache sizes which range from 4KB to 64KB. We executed the programs on a PlayStation 3 [4] which has one 3.2 GHz Cell Broadband Engine CPU (with 6 accessible SPEs) and 256MB global shared memory. Our experiments used all 6 SPEs with the exception of the evaluation of speedup which used various numbers of SPEs from 1 to 6.

We used five benchmark programs in our experiments — RandomAccess and Stream from the HPC Challenge benchmark suite [2], Integer Sort (IS), Embarrassingly Parallel (EP) and Multigrid (MG) from the NAS Parallel Benchmarks [3]. In our experiments, the OpenMP code was used with little change from the original benchmark version. Hence, the performance advantages obtained reported in this paper were achieved without any adverse impact on OpenMP programmability.

4.2 Summary of Main Results

The main results of our experiments are as follows:

- *Result I: Performance and Scalability (Section 4.3).*

  *Model*LF cache protocol consistently outperformed *Model*GF cache protocol on all five benchmark programs. Our results also demonstrate good overall performance scalability as a function of the number of SPEs under *Model*LF cache protocol for the collection of programs tested.

- *Result II: Impact of Cache Size (Section 4.4).*

  Our results show that the performance gap between *Model*LF and *Model*GF cache protocols increases as the cache size becomes smaller. This observation is significant because the current trend in manycore processors is the local memory size per core decreases as the number of cores increases.

In Section 4.5, we introduce the preliminary experimental study of *Model*RLF comparing with *Model*LF . However, we are not able to provide a complete experimental analysis and comparison between *Model*RLF and *Model*LF . It will be left as a topic for future work. We realize that a complete *Model*RLF performance evaluation should be conducted in conjunction with an efficient implementation of OpenMP critical sections that can exploit the acquire and release operations of *Model*RLF . In our current implementation we follow the convention that only a single global lock is used for all the OpenMP (unnamed) critical sections. We believe that to fully exploit the advantage of acquire and release operations, a finer-grain implementation of global locks must be pursued as in [36].

4.3 Performance and Scalability

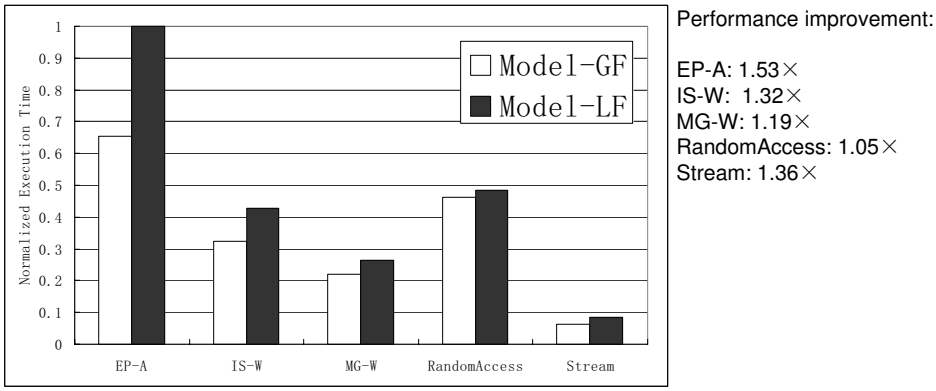


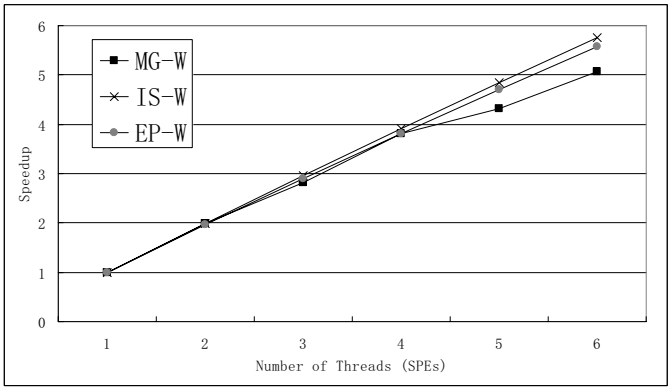
Figure 4: Execution time comparison between *Model*GF and *Model*LF cache protocols. Cache size is 32KB.

Figure 4 shows the execution time comparison between *Model*GF and *Model*LF cache protocols using 32KB cache size on all five benchmark programs.[3] *Model*LF cache protocol consistently outperformed *Model*GF cache protocol. The reason is that the cost of global flush operations (mainly caused by cache evictions) is much higher than the cost of local flush operations. For example, Figure 5 shows the numbers of accessing centralized directory under *Model*GF and *Model*LF for MG-W and IS-W on various cache sizes. Because *Model*LF dose not have centralized directory, the number of accessing is always zero. However, under *Model*GF the number is quite large.

[3]For convenience, we use EP-A to represent EP benchmark with input size A. The similar way is used for all NAS Parallel Benchmarks.

Cache size	# of accessing centralized directory for MG-W		# of accessing centralized directory for IS-W	
	Model-*GF*	Model-*LF*	Model-*GF*	Model-*LF*
4K	16487653	0	12034247	0
8K	8271146	0	11706403	0
16K	4436299	0	11055545	0
32K	3747588	0	9769592	0

Figure 5: The numbers of accessing centralized directory under *Model*GF and *Model*LF for MG-W and IS-W on various cache sizes.



IS-W and EP-W achieve almost linear speedup.

MG-W performs worse because of unbalanced workloads.

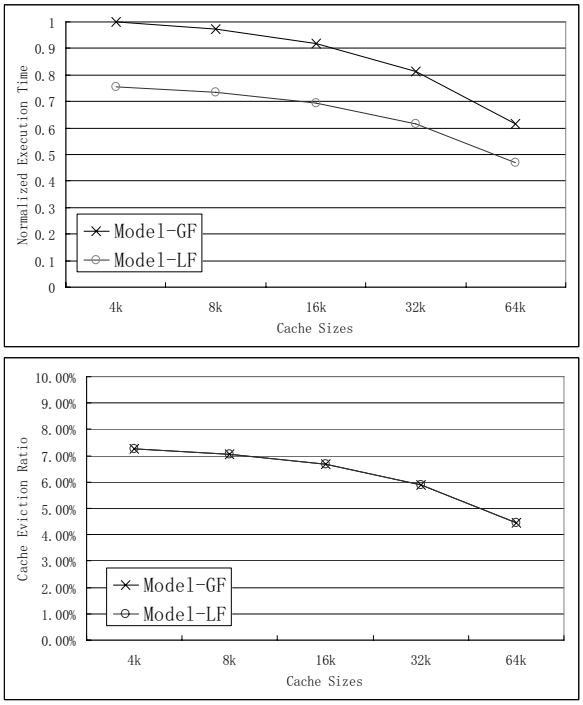
Figure 6: Speedup as a function of the number of SPEs under *Model*LF cache protocol.

Figure 6 shows the speedup as a function of the number of SPEs (We assume that each SPE runs a thread.) under *Model*LF cache protocol. The tested applications are MG-W with a 32KB cache size, and IS-W and EP-W with a 64KB cache size. We can see that for IS and EP benchmarks, *Model*LF cache protocol nearly achieves linear speedup. For MG benchmark, the speedup is not as good as the other two when the number of threads is 3, 5 and 6. The reason is that the workloads among threads are not balanced when the number of threads is not a power of 2.

4.4 Impact of Cache Size

Figure 7 and 8 show execution time and cache eviction ratio curves for IS-W and MG-W on various cache sizes (4KB, 8KB, 16KB, 32KB and 64KB [4]) per thread. The two figures show that the cache eviction ratio curves under the two cache protocols are equal, but the normalized execution time curves are not. Moreover, the difference in execution time becomes larger as the cache size becomes smaller. This is because the cost of cache eviction in *Model*GF cache protocol is much higher. Moreover, the smaller the cache size is, the higher the cache eviction ratio is.

[4] 64KB is only for IS-W



The difference of normalized execution time increased from 0.15 to 0.25 as the cache size per SPE was decreased from 64KB to 4KB.

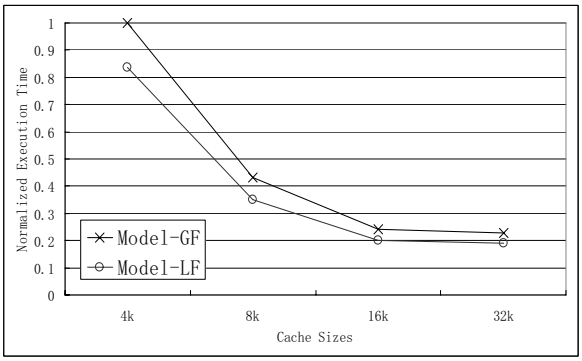
The two curves of cache eviction ratio are overlapped because of completely identical cache settings.

Figure 7: Trends of execution time and cache eviction ratio for IS-W on various cache sizes.

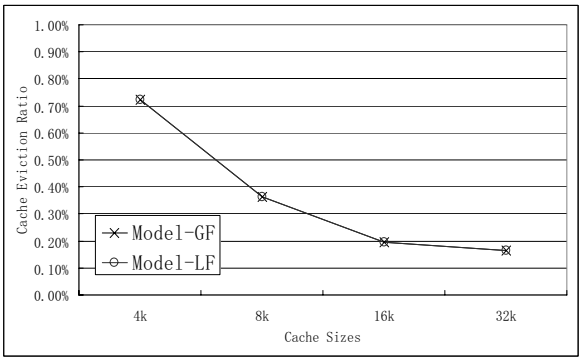
4.5 Impact of Refinement of Flush Operations

In this section, we introduce the preliminary experimental results for refining the flush operations, which was implemented by *Model*RLF cache protocol. Under *Model*RLF cache protocol, since its finer-grain flush operations may not remove data from cache, future cache access may achieve more cache hits. Therefore, the performance of the application can be improved.

In the following table, we show the performance improvement of *Model*RLF cache protocol comparing with *Model*LF cache protocol on MG benchmark. From the table, we can see that when the input size is fixed, the bigger the cache size is, the better the performance improvement is. Moreover, when the cache size is fixed, the smaller the input size is, the better the performance improvement is. It satisfies our analysis because when the input size is smaller and the cache size is bigger, the cache hit ratio is higher. Therefore, more cache hits gain benefits from *Model*RLF cache protocol since its finer-grain flush operations may not remove data from cache.



The difference of normalized execution time increased from 0.04 to 0.16 as the cache size per SPE was decreased from 32KB to 4KB.



The two curves of cache eviction ratio are overlapped because of completely identical cache settings.

Figure 8: Trends of execution time and cache eviction ratio for MG-W on various cache sizes.

Input Size	Cache Size	Performance Improvement of *Model*RLF
W	4K	0.0002
W	8K	0.0008
W	16K	0.0019
W	32K	0.0031
S	4K	0.0019
S	8K	0.0030
S	16K	0.0100
S	32K	0.0112

5 Related Work

Despite over two decades of research on memory consistency models [29, 24, 17, 22, 28, 26, 9, 10, 21], there does not appear to be a consensus on how memory models should be formalized [7, 35, 34, 8]. The efforts to formalize memory models for mainstream parallel languages such as the Java memory model [31], the C++ memory model [11], and the OpenMP memory model [12] all take different approaches.

The authoritative source for the OpenMP memory model can be found in the specifications for OpenMP 2.5 [6] and OpenMP 3.0 [32], but the memory model definitions therein are provided in terms of informal prose. To address this limitation, a formalization of the OpenMP

memory model was presented in [12]. In this paper, the authors developed a formal, mathematical language to model the relevant features of OpenMP. They developed an operational model to verify its conformance to the OpenMP standard. Through these tools, the authors found that the OpenMP memory model is weaker than the weak consistency model [17]. The authors also claimed that they found some ambiguities in the informal definition of the OpenMP memory model presented in the OpenMP specification version 2.5 [6]. Their work demonstrates the need for the OpenMP community to work towards a formal and complete definition of the OpenMP memory model.

Some early research on software controlled caches can be found in the NYU Ultracomputer [25], Cedar [20], and IBM RP3 [33] projects. All three machines have local memories that can be used as programmable caches, with software taking responsibility for maintaining consistency by inserting explicit synchronization and cache consistency operations. By default, this responsibility falls on the programmer but compiler techniques have also been developed in which these operations are inserted by the compiler instead *e.g.,* [16]. Interest in software caching has been renewed with the advent of multicore processors with local memories such as the Cell Broadband Engine. There have been a number of reports on more recent software cache optimization from compiler angle as described in [19, 18, 14].

Examples of recent work on software cache protocol implementation on CELL processors can be found in [30, 13, 23]. The cache protocol used in [30] uses a centralized directory to keep tract cache line state information in the implementation - reminds us the *Model*GF cache protocol in this paper. The cache protocols reported in [13, 23] do not appear to use a centralized directory - hence appear to be more close to the *Model*LF cache protocol. However, we do not have access to the detailed information on the implementations of these models, and cannot make a more definitive comparisons at the time when this paper is written.

OPELL [27] is an open source toolchain / runtime effort to implement OpenMP for the Cell Broadband Engine. Our cache protocol framework reported here has been developed much earlier in 2006-2007 frame and embedded in OPELL (see [27])- but the protocols themselves are not published externally.

6 Conclusion and Future Work

In this paper, we investigate the problem of software cache implementations for the OpenMP memory model on multicore processors. We first formalize the idealized OpenMP memory model (*Model*IDEAL) that assumes unbounded space for temporary views, and then formalize three practical instantiations — *Model*GF (based on nondeterministic *global flushes*), *Model*LF (based on nondeterministic *local flushes*), and *Model*RLF (a further refinement of *Model*LF 's flush operations). We observe that, for data-race-free programs, the four models are equivalent.

We present cache protocols for the three instantiations of the OpenMP memory models we discussed: *Model*GF , *Model*LF and *Model*RLF , and describe their implementation for soft-

ware cache of the CELL processor. show the following results: i) the cache protocols based on *Model*LF consistently outperforms the protocol based on *Model*GF , ii) furthermore this performance gap increases as the size of the local memory per core decreases. As the size of available on chip memory space per core is getting smaller as the number of cores increasing - this is an important observation favor more decentralized memory models/protocols that does not rely on centralized directory as the *Model*GF .

This provides a useful way that how to formalize (architecture unspecified) OpenMP memory model in different ways and evaluate the instantiations to produce different performance profiles. Our conclusion is that OpenMP's relaxed memory model with temporary views is a good match for software cache implementations, and that the refinements in *Model*LF and *Model*RLF can lead to good opportunities for scalable implementations of OpenMP on future multicore processors.

We intend to do the future work as follows:

- **Studies of *Model*RLF .** As we pointed out in Section 4.2, a complete *Model*RLF performance evaluation should be conducted in conjunction with an efficient implementation of OpenMP critical sections that can exploit the acquire and release operations of *Model*RLF . We think it is an interesting topic and expect to fully exploit the advantage of *Model*RLF .

- **Tests on more benchmarks.** By testing more benchmarks, we expect to have a complete study of the advantage and condition of applying *Model*LF rather than *Model*GF .

- **Evaluations on more many-core architectures.** By doing this, we expect to show that the idea of software-managed cache and our memory models can be widely used on various many-core architectures to achieve performance and programmability goals.

References

[1] *Cell Broadband Engine*. http://www-01.ibm.com/chips/techlib/techlib.nsf/products/Cell_Broadband_Engin

[2] *HPC Challenge Benchmark*. http://icl.cs.utk.edu/hpcc/.

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

[4] *PlayStation3*. http://www.us.playstation.com/ps3/features.

[5] *Tilera*. http://www.tilera.com/.

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

[7] S. V. Adve and J. K. Aggarwal. A unified formalization of four shared-memory models. *IEEE Trans. Parallel Distrib. Syst.*, 4(6):613–624, 1993.

[8] Arvind Arvind and Jan-Willem Maessen. Memory model = instruction reordering + store atomicity. In *ISCA '06: Proceedings of the 33rd annual international symposium on Computer Architecture*, pages 29–40, Washington, DC, USA, 2006. IEEE Computer Society.

[9] 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.

[10] 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 *SPAA '96: Proceedings of the eighth annual ACM symposium on Parallel algorithms and architectures*, pages 297–308, New York, NY, USA, 1996. ACM.

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

[12] Greg Bronevetsky and Bronis R. de Supinski. Complete formal specification of the openmp memory model. *International Journal of Parallel Programming*, 35(4):335–392, 2007.

[13] Tong Chen, Haibo Lin, and Tao Zhang. Orchestrating data transfer for the cell/b.e. processor. In *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*, pages 289–298, New York, NY, USA, 2008. ACM.

[14] 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.

[15] Juan Cuvillo, Weirong Zhu, Ziang Hu, and Guang R. Gao. Fast: A functionally accurate simulation toolset for the cyclops-64 cellular architecture. In *In: Workshop on Modeling, Benchmarking, and Simulation (MoBS2005), in conjuction with the 32nd Annual International Symposium on Computer Architecture (ISCA2005*, pages 11–20, 2005.

[16] Ron Cytron, Steve Karlovsky, and Kevin P. McAuliffe. Automatic management of programmable caches. In *ICPP'88: Proceedings of the 1988 International Conference on Parallel Processing*, pages 229–238, Augest 1988.

[17] M. Dubois, C. Scheurich, and F. Briggs. Memory access buffering in multiprocessors. In *ISCA '86: Proceedings of the 13th annual international symposium on Computer architecture*, pages 434–442, Los Alamitos, CA, USA, 1986. IEEE Computer Society Press.

[18] A. E. Eichenberger, J. K. O'Brien, K. M. O'Brien, P. Wu, T. Chen, P. H. Oden, D. A. Prener, J. C. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, M. K. Gschwind, R. Archambault, Y. Gao, and R. Koo. Using advanced compiler technology to exploit the performance of the cell broadband enginetm architecture. *IBM Syst. J.*, 45(1):59–84, 2006.

[19] Alexandre E. Eichenberger, Kathryn O'Brien, Kevin O'Brien, Peng Wu, Tong Chen, Peter H. Oden, Daniel A. Prener, Janice C. Shepherd, Byoungro So, Zehra Sura, Amy Wang, Tao Zhang, Peng Zhao, and Michael Gschwind. 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.

[20] Daniel Gajski, David Kuck, Duncan Lawrie, and Ahmed Sameh. Cedar: a large scale multiprocessor. *SIGARCH Comput. Archit. News*, 11(1):7–11, 1983.

[21] Guang R. Gao and Vivek Sarkar. Location consistency: Stepping beyond the barriers of memory coherence and serializability. Technical Report 78, December 1994.

[22] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th ACM International Symposium on Computer Architecture*, pages 15–27, May 1990.

[23] Marc Gonzàlez, Nikola Vujic, Xavier Martorell, Eduard Ayguadé, Alexandre E. Eichenberger, Tong Chen, Zehra Sura, Tao Zhang, Kevin O'Brien, and Kathryn O'Brien. Hybrid access-specific software cache techniques for the cell be architecture. In *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 292–302, New York, NY, USA, 2008. ACM.

[24] J. R. Goodman. Cache consistency and sequential consistency. Technical Report 1006, Wisconsin, Madison, February 1991.

[25] Allan Gottlieb, Ralph Grishman, Clyde P. Kruskal, Kevin P. McAuliffe, Larry Rudolph, and Marc Snir. The nyu ultracomputer—designing a mimd, shared-memory parallel machine (extended abstract). In *ISCA '82: Proceedings of the 9th annual symposium on Computer Architecture*, pages 27–42, Los Alamitos, CA, USA, 1982. IEEE Computer Society Press.

[26] Liviu Iftode, Jaswinder Pal Singh, and Kai Li. Scope consistency: a bridge between release consistency and entry consistency. In *SPAA '96: Proceedings of the eighth annual ACM symposium on Parallel algorithms and architectures*, pages 277–287, New York, NY, USA, 1996. ACM.

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

[28] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed memory. In *Proceedings of the 19th ACM International Symposium on Computer Architecture*, pages 13–21, May 1992.

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

[30] Jaejin Lee, Sangmin Seo, Chihun Kim, Junghyun Kim, Posung Chun, Zehra Sura, Jungwon Kim, and SangYong Han. Comic: a coherent shared memory interface for cell be. In *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 303–314, New York, NY, USA, 2008. ACM.

[31] 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.

[32] OpenMP Architecture Review Board. OpenMP Application Program Interface Version 3.0, May 2008. http://www.openmp.org/mp-documents/spec30.pdf.

[33] 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 research parallel processor prototype (rp3): Introduction and architecture. In *ICPP'85: Proceedings of the 1985 International Conference on Parallel Processing*, pages 764–771, 1985.

[34] 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.

[35] Xiaowei Shen, Arvind, and Larry Rudolph. Commit-reconcile & fences (crf): A new memory model for architects and compiler writers. In *In Proceedings of the 26th International Symposium on Computer Architecture*, pages 150–161, 1999.

[36] Yuan Zhang, Vugranam C. Sreedhar, Weirong Zhu, Vivek Sarkar, and Guang R. Gao. Minimum lock assignment: A method for exploiting concurrency among critical sections. In José Nelson Amaral, editor, *LCPC*, volume 5335 of *Lecture Notes in Computer Science*, pages 141–155. Springer, 2008.