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

# A Study of a Software Cache Implementation of the OpenMP Memory Model for Multicore and Manycore Architectures

*Chen Chen*
*Joseph B Manzano*
*Ge Gan*
*Guang R. Gao*
*Vivek Sarkar*

**CAPSL Technical Memo 093**

February, 2010
Revised in May, 2010

## Abstract

This paper is motivated by the desire to provide an efficient and scalable software cache implementation of OpenMP on multicore and manycore architectures in general, and on the IBM CELL architecture in particular. In this paper, we propose an instantiation of the OpenMP memory model with the following advantages: (1) The proposed instantiation prohibits out-of-thin-air values that may cause problems of safety, security, programming and debugging. (2) The proposed instantiation is scalable with respect to the number of threads because it does not rely on communication among threads or a centralized directory that maintains consistency of multiple copies of each shared variable. (3) The proposed instantiation avoids the ambiguity of the original memory model definition proposed on the OpenMP Specification 3.0.

We also introduce a new cache protocol for this instantiation, which can be implemented as a software-controlled cache. Experimental results on the Cell Broadband Engine show that our instantiation results in nearly linear speedup with respect to the number of threads for a number of NAS Parallel Benchmarks. The results also show a clear advantage when comparing it to a software cache design derived from a stronger memory model that maintains a global total ordering among flush operations.

# 1 Introduction

An important open problem for future multicore and manycore chip architectures is the development of shared-memory organizations and memory consistency models (or memory models for short) that are effective for small local memory sizes in each 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) [21] are supported on mainstream small-scale SMPs, it seems likely that weaker memory models will be explored in current and future multicore and manycore architectures such as the Cell Broadband Engine [1], Tilera [4], and Cyclops64 [12].

OpenMP [24] is a natural candidate as a programming model for multicore and manycore processors with software-managed local memories, thanks to its weak memory model. In the OpenMP memory model, each thread may maintain a *temporary view* of the shared memory which “allows the thread to cache variables and thereby to avoid going to memory for every reference to a variable” [24]. It includes a *flush* operation on a specified *flush-set* that can be used to synchronize the temporary view with the shared memory for the variables in the flush-set. It is a weak consistency model “because a threads temporary view of memory is not required to be consistent with memory at all times” [24]. 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 has different performance and cost tradeoff. Therefore, the OpenMP memory model can exhibit very different instantiations, each of which may imply a different memory model. We say that a memory model *M* is an instantiation of the OpenMP memory model when every read operation under *M* returns less or the same possible values than the same read operation under the OpenMP memory model.

Among various instantiations of the OpenMP memory model, an important problem is to find an instantiation that can be efficiently implemented on multicore and manycore architectures and easily understood by programmers.

| Thread 1 | Thread 2 |
|---|---|
| 1: x = 1; | 4: x = 2; |
| 2: y = 1; | 5: y = 2; |
| 3: flush(x,y); | 6: flush(x,y); |

Is $x = 1, y = 2$ (or $x = 2, y = 1$) legal under the OpenMP memory model?

Figure 1: A motivating example for understanding the serialization requirement under the OpenMP memory model.

### 1.1 A Key Observation for Implementing the Flush Operation Efficiently

The flush operation synchronizes temporary views with the shared memory. So it is more expensive than read and write operations. In order to efficiently implement the OpenMP memory model, the instantiation should be able to implement the flush operation efficiently.

Unfortunately, the OpenMP memory model has the serialization requirement for flush operations, *i.e.,* "if the intersection of the flush-sets of two flushes performed by two different threads is non-empty, then the two flushes must be completed as if in some sequential order, seen by all threads" [24]. Therefore, it seems that it is very hard to efficiently implement the flush operation because of the serialization requirement. However, this requirement has a hidden meaning that is not clearly explained in [24]. The hidden meaning is the key for efficiently implement the flush operation.

We use an example to explain the real meaning of the serialization requirement. For the program in Fig. 1, it seems that the final status of the shared memory must be either $x = y = 1$ or $x = y = 2$ according to the serialization requirement. However, after discussion with the OpenMP community, $x = 1, y = 2$ and $x = 2, y = 1$ are also legal results under the OpenMP memory model. The reason is that the OpenMP memory model allows flush operations to be completed earlier (but cannot be later) than the flush points (statements 3 and 6 in this program). Therefore, one possible way to get the result $x = 1, y = 2$ is that firstly thread 2 assigns 2 to $x$ and immediately flushes $x$ into the shared memory, then thread 1 assigns 1 to $x$ and 1 to $y$ and then flushes $x$ and $y$, and finally thread 2 assigns 2 to $y$ and flushes $y$. Therefore, we get a key observation for implementing the flush operation efficiently as follows.

**The Key Observation:** A flush operation on a flush-set of shared locations can be decomposed into unordered flush operations on each individual location. Each flush operation after decomposition must be completed no later than the flush point of the original flush operation. Assuming that a memory location is the minimal unit for atomic memory accesses, the serialization requirement is naturally satisfied.

### 1.2 Main Contributions

In this paper, we propose an instantiation of the OpenMP memory model based on the key observation in Section 1.1. It has the following advantages.

- Our instantiation prohibits out-of-thin-air values that may cause problems of safety, security, programming and debugging. The OpenMP memory model may allow out-of-thin-air values for programs with data races. However, in our instantiation, a memory read operation always reads the initial value or a value that was written by some thread before. So it cannot generate any out-of-thin-air value. The out-of-thin-air values may cause various problems as pointed out in [23]. Since the OpenMP memory model supports programs with data races [1], our instantiation would be helpful when programming such programs.

- Our instantiation is scalable with respect to the number of threads because it does not rely on communication among threads or a centralized directory that maintains consistency of multiple copies of each shared variable.

- Our instantiation avoids the ambiguity of the original memory model definition proposed on the OpenMP Specification 3.0, such as the unclear serialization requirement, the problem of handling temporary overflow and the unclear semantics for programs with data races. Therefore, our instantiation is easy to understand from the angle of efficient implementations.

We also propose the cache protocol of the instantiation and implement the software-controlled cache on Cell Broadband Engine. The experimental results show that our instantiation has nearly linear speedup with respect to the number of threads for a number of NAS Parallel Benchmarks. The results also show a clear advantage when comparing it to a software cache design derived from a stronger memory model that maintains a global total ordering among flush operations.

The rest of the paper is organized as follows. Section 2 introduces our instantiation of the OpenMP memory model. Section 3 introduces the cache protocol of the instantiation. Section 4 presents the experimental results. Section 5 discusses the related work. The conclusion is presented in Section 6.

## 2 Formalization of Our OpenMP Memory Model Instantiation

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 [9], "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[2], discussion 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.

In this section, we formalize an instantiation of the OpenMP Memory Model — *Model*LF , based on the key observation in Section 1.1. *Model*LF builds on OpenMP's relaxed-consistency memory model in which each worker thread maintains a *temporary view* of shared data which may not always

[1] Section 2.8.6 of the OpenMP specification 3.0 [24] shows a program with data races that implements critical sections.

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

be consistent with the actual data stored in the shared memory. The OpenMP *flush* operation is used to establish consistency between these temporary views and the shared memory at specific program points. In *Model*LF , each flush operation only forces local temporary view to be consistent with the shared memory. That is why we call it *Model*LF where “LF” means local flush. A flush operation is only applied on a single location. We assume that a memory location is the minimal unit for atomic memory accesses. Therefore, the serialization 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 location, where those flush operations after decomposition must be completed no later than the flush point of the original flush operation. So it avoids the known problem of decomposition as explained in Section 2.8.6 of the OpenMP specification 3.0 [24], where the compiler may reorder the flush operations after decomposition to a later position than the flush point and cause incorrect semantics.

To compare with *Model*LF in our experiments, we also introduce another instantiation — *Model*GF which maintains a global total ordering among flush operations.

## 2.1 Operational Semantics of ***Model***LF

In this section, we define the operational semantics of *Model*LF . Firstly, we introduce a little background for the definition. A store, $\sigma$, 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, $\sigma_i$, 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*$(T_i, l)$ makes temporary view $\sigma_i$ consistent with the shared memory $\sigma$ on location $l$.

Under *Model*LF , program flush operations are performed at the program points specified by the programmer. Moreover, additional 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, such as caches. The operational semantics of memory operations of *Model*LF include the read, write, program flush operation and nondeterministic flush operation defined 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 $\sigma_i$. If $\sigma_i$ does not contain any value of $l$, the value in the shared memory will be loaded to $\sigma_i$ 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 $\sigma_i$.

- **Program / Nondeterministic flush:** If thread $T_i$ needs to synchronize $\sigma_i$ with the shared memory on a shared location $l$, it performs a $flush(T_i, l)$ operation. If $\sigma_i$ contains a “dirty value” [3] of $l$, it will write back the value into the shared memory. After the flush operation, $\sigma_i$ will discard

[3] The term “dirty value” means that the value of location $l$ was modified by thread $T_i$.

the value of $l$. A thread performs program flush operations at program points specified by the programmer, and can nondeterministically perform flush operations at any program point. All the program and nondeterministic flush operations on the same shared location must be observed by all threads to be completed in the same sequential order.

## 2.2 Operational Semantics of *Model*GF

In this section, we define the operational semantics of *Model*GF . *Model*GF maintains a global total ordering among flush operations. The difference between *Model*GF and *Model*LF is that when *Model*GF performs a flush operation on a location $l$, it enforces the temporary views of all threads to see the same value of $l$ by discarding the values of $l$ in the temporary views. The operational semantics of memory operations of *Model*GF include the read, write, program flush operation and nondeterministic flush operation defined 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 $\sigma_i$. If $\sigma_i$ does not contain any value of $l$, the value in the shared memory will be loaded to $\sigma_i$ 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 $\sigma_i$.

- **Program / Nondeterministic flush:** If thread $T_i$ needs to synchronize $\sigma_i$ with the shared memory and all the other temporary views on a shared location $l$, it performs a $flush(T_i, l)$ operation. If $\sigma_i$ contains a “dirty value” of $l$, it will write back the value into the shared memory. Moreover, if any other temporary view contains a clean or dirty value of $l$, that value will be discarded. After the flush operation, $\sigma_i$ will also discard the value of $l$. A thread performs program flush operations at program points specified by the programmer, and can nondeterministically perform flush operations at any program point. All the program and nondeterministic flush operations on the same shared location must be observed by all threads to be completed in the same sequential order.

# 3 Cache Protocol of *Model*LF

In this section, we introduce the cache protocol that implements *Model*LF . We assume that each thread contains a cache which corresponds to its temporary view. Therefore, performing operations on temporary views 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 decomposition problem.

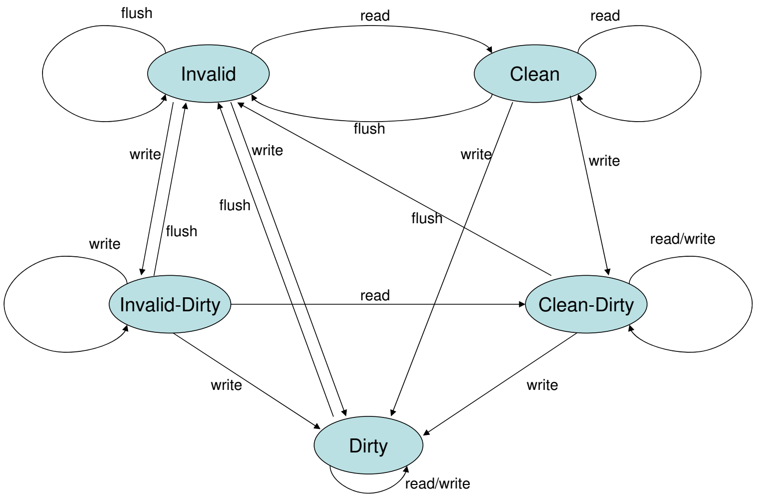


Figure 2: State transition diagram for the cache protocol of *Model*LF .

### 3.1 Cache Line States

We assume that each cache line contains multiple locations. Each location contains a value that can be a "clean value" [4], a "dirty value", or an "invalid value". Each cache line can be in one of the five states as follows.

**Invalid:** All the locations contain "invalid values".

**Clean:** All the locations contain "clean values".

**Dirty:** All the locations contain "dirty values".

**Clean-Dirty:** Some locations contain "clean values". The others contain "dirty values".

**Invalid-Dirty:** Some locations contain "invalid values". The others contain "dirty values".

For simplicity, the cache line cannot be in other states such as **Invalid-Clean**. Additional nondeterministic flush operations may be performed when necessary to force the cache line to be in one of the five states as above. We use a per-line flag bit together with the dirty bits to represent the state of the cache line. The flag bit indicates whether those non-dirty values in the cache line are clean or invalid.

### 3.2 Cache Operations and State Transitions

The state transition diagram of *Model*LF cache protocol is shown in Fig. 2. Now we explain how each cache operation affects the state transition diagram.

**Memory read:** If the original state of the cache line is invalid or invalid-dirty, the invalid locations will load "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.

[4] The term "clean value" means that the value of the location was not modified by the thread.

**Memory 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 after the write operation, which depends on whether all the locations contain “dirty values”. In other cases, the state will become either clean-dirty or dirty, which depends on whether all the locations contain “dirty values”.

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

There may be various ways to implement the flush operation. For example, many architectures support a block of data to be written back at a time. So a possible way of implementing the flush operation is to write back the entire cache line that is being flushed together with the dirty bits and then merge the “dirty values” into the corresponding memory line in the shared memory. If the mergence in memory is not supported, a thread has to load the memory line, and then merge it with the cache line, and finally write back the merged line, where the process must be atomic to handle the false sharing problem.

# 4 Experimental Results and Analyses

In this section, we introduce our experimental results under *Model*LF cache protocol. In section 4.1, we introduce the experimental testbed. 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 two sections.

## 4.1 Experimental Testbed

The experimental results presented in this paper were obtained on the Cell Broadband Engine Architecture (CBEA) [1] under the OPELL (OPenmp for cELL) framework [20].

### 4.1.1 CBEA.

CBEA has a main processor called the Power Processing Element (PPE) and a number of co-processors called the Synergistic Processing Elements (SPEs). The PPE handles most of the computational workload and has control over the SPEs, *i.e.*, start, stop, interrupt, and schedule processes onto the SPEs. Each SPE has a 256KB local storage which is used to store both instructions and data. An SPE can only access its own local storage directly. Both PPE and SPEs share main memory. SPEs access main memory via DMA (direct memory access) transfers which are much slower than the access on each SPE’s own local storage.

We executed the programs on a PlayStation 3 [3] 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.

### 4.1.2 OPELL Framework.

OPELL is an open source toolchain / runtime effort to implement OpenMP for the CBEA. OPELL has a single source compiler which compiles an OpenMP program to a single source file that is executable on CBEA.

During runtime, the executable file starts to run sequential codes of the program on PPE. Once the program enters a parallel region, PPE will assign tasks of computing parallel codes to SPEs. After SPEs finish the tasks, the parallel region ends and PPE will go ahead to execute the following sequential codes.

Since each SPE only has 256KB local storage to store both instructions and data, OPELL has a partition /overlay manager runtime library that partitions the parallel codes into small pieces to fit for the local storage size, and loads and replaces those pieces on demand.

Since a DMA transfer is much slower than an access on the local storage, OPELL has a software cache runtime library to take advantage of locality. The runtime library manages a part of local storages as caches and has a user interface for accessing. We implement our cache protocol in OPELL's software cache runtime library. The cache protocol uses 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 did not try bigger cache size because the size of local storage is very limited (256KB) and a part of it is used to store instructions and maintain stack.

### 4.1.3 Benchmarks

We used three benchmark programs in our experiments — Integer Sort (IS), Embarrassingly Parallel (EP) and Multigrid (MG) from the NAS Parallel Benchmarks [2]. In our experiments, the OpenMP code was used with little change from the original benchmark version. Hence, our implementation does not have adverse impact on OpenMP programmability.

## 4.2 Summary of Main Results

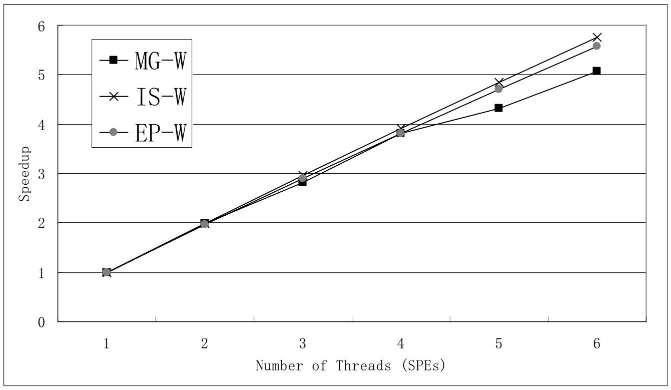
The main results of our experiments are as follows:

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

  *Model*LF cache protocol has nearly linear speedup with respect to the number of threads for the tested benchmarks.

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

  We use *Model*GF to compare with *Model*LF . To implement *Model*GF , we simulate a centralized directory that maintains the information for all the caches. So a flush operation under *Model*GF can look up the directory and inform the threads that contain the value of the same location to discard the value. We also assume that the centralized directory is "idealized", which



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

MG-W performs worse because of unbalanced workloads.

Figure 3: Speedup as a function of the number of SPEs under *Model*LF cache protocol.

means that the cost of cache information maintaining and lookup is trivial. However, the cost of informing another thread is as expensive as a DMA transfer in CEBA. To our knowledge, currently there is no other formalized instantiation of the OpenMP memory model that can be implemented for comparison.

*Model*LF outperforms *Model*GF due to its cheaper flush operations. 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 multicore and many-core processors is that the local memory size per core decreases as the number of cores increases.

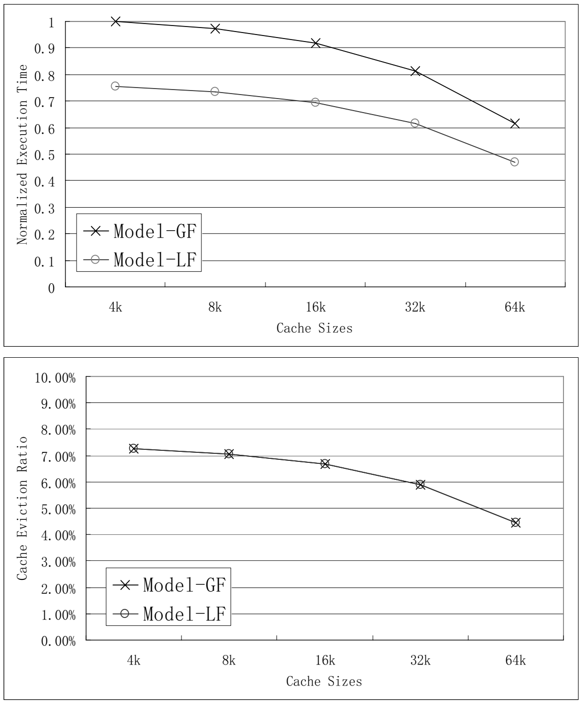
## 4.3 Scalability

Fig. 3 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 with a 32KB cache size, and IS and EP with a 64KB cache size. All the three applications have input size *W*. 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

Fig. 4 and 5 show execution time and cache eviction ratio curves for IS and MG with input size *W* on various cache sizes (4KB, 8KB, 16KB, 32KB and 64KB [5]) per thread. The two figures show that the cache eviction ratio curves under the two cache protocols are equal, but the 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. To show the change of performance gap clearly, we normalize the execution time into the interval $[0, 1]$ by applying division on every execution

[5] 64KB is only for IS



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

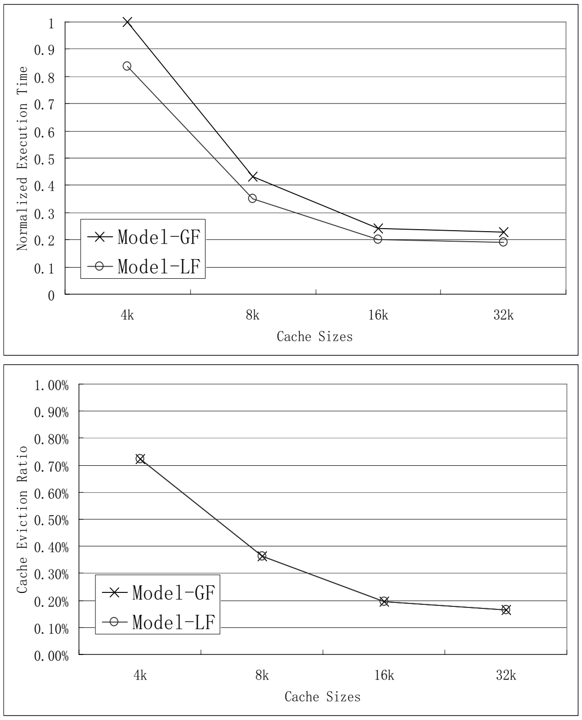
time where the divisor is the maximal execution time in all tested configurations. The corresponding configurations to the maximal execution time are 4KB cache sizes under *Model*GF for both MG and IS.

The performance gap between *Model*GF and *Model*LF keeps constantly for EP when we change the cache sizes. The reason is that EP has very bad temporal locality. So it is insensitive to the change of cache sizes.

## 5 Related Work

Despite over two decades of research on memory consistency models, there does not appear to be a consensus on how memory models should be formalized [6, 27, 26, 7]. The efforts to formalize memory models for mainstream parallel languages such as the Java memory model [23], the C++ memory model [8], and the OpenMP memory model [9] all take different approaches.

The authoritative source for the OpenMP memory model can be found in the specifications for OpenMP 3.0 [24], but the memory model definition therein is provided in terms of informal prose. To address this limitation, a formalization of the OpenMP memory model was presented in [9]. 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 [14]. 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 [5]. Since there



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

is no significant change of the OpenMP memory model from version 2.5 to version 3.0, 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 [19], Cedar [17], and IBM RP3 [25] 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.*, [13]. 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 [16, 15, 11].

Examples of recent work on software cache protocol implementation on CELL processors can be found in [22, 10, 18]. The cache protocol used in [22] 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 [10, 18] 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 [20] 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 [20])- but the protocols themselves are not published externally.

## 6 Conclusion

In this paper, we investigate the problem of software cache implementations for the OpenMP memory model on multicore and manycore processors. We propose an instantiation of the OpenMP memory model — *Model*LF which prohibits out-of-thin-air values and avoids the ambiguity of the original memory model definition on OpenMP Specification 3.0. *Model*LF is scalable with respect to the number of threads because it does not rely on communications among threads or a centralized directory that maintains consistency of multiple copies of each shared variable.

We propose the corresponding cache protocol and implement the cache protocol by software cache on the Cell processor. The experimental results show that *Model*LF cache protocol has nearly linear speedup with respect to the number of threads for a number of NAS Parallel Benchmarks. The results also show a clear advantage when comparing it to *Model*GF cache protocol derived from a stronger memory model that maintains a global total ordering among flush operations.

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 can lead to good opportunities for scalable implementations of OpenMP on future multicore and manycore processors.

## References

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

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

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

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

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

[6] Sarita Adve and Mark D. Hill. A unified formalization of four shared-memory models. *IEEE Transactions on Parallel and Distributed Systems*, 4:613–624, 1993.

[7] Arvind Arvind and Jan-Willem Maessen. Memory model = instruction reordering + store atomicity. *SIGARCH Comput. Archit. News*, 34(2):29–40, 2006.

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

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

[10] 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.

[11] 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.

[12] Juan Cuvillo, Weirong Zhu, Ziang Hu, and Guang R. Gao. Fast: A functionally accurate simulation toolset for the Cyclops-64 cellular architecture. In *In Proceedings of the Workshop on Modeling, Benchmarking and Simulation, pages 11–20, Madison, Wisconsin, June 4, 2005. Held in conjunction with the 32nd Annual International Symposium on Computer Architecture.*, pages 11–20, 2005.

[13] 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.

[14] Michel Dubois, Christoph Scheurich, and Faye Briggs. Memory access buffering in multiprocessors. In *ISCA '98: 25 years of the international symposia on Computer architecture (selected papers)*, pages 320–328, New York, NY, USA, 1998. ACM.

[15] 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.

[16] 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.

[17] D Gajski, D Kuck, D Lawrie, and A Sameh. CEDAR—a large scale multiprocessor. pages 69–74, Los Alamitos, CA, USA, 1986. IEEE Computer Society Press.

[18] 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.

[19] 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. In *ISCA '98: 25 years of the international symposia on Computer architecture (selected papers)*, pages 239–254, New York, NY, USA, 1998. ACM.

[20] 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.

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

[22] 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.

[23] 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.

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

[25] 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.

[26] 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.

[27] 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.