Towards Exploiting the Architectural Features of Beehive *

Gautam Shah *Umakishore Ramachandran*

GIT-CC-91/51

College of Computing
Georgia Institute of Technology
Atlanta, GA 30332 USA
Ph: (404) 894-5136
e-mail: rama@cc.gatech.edu

Abstract

"Beehive" is a project that investigates the software and hardware issues in the design of scalable shared memory multiprocessors. The architecture is designed to support a form of weakly consistent memory model in a cache-based multiprocessor environment. The novel features of the architecture are decoupling the notion of the type of data (private or shared) from the domain (local or global) over which the access should be performed; eliminating false sharing; providing for software assisted consistency maintenance using reader-initiated coherence; and supporting queue-based shared and exclusive locks in hardware. We identify how the architectural features supporting a weak memory model can be exploited by the system software such as the compiler and the runtime. In particular, marking algorithms are developed for specifying the domain of loads and stores that occur between synchronization points in a parallel program.

Key Words:
Multiprocessor caches, shared memory consistency models, reader-initiated coherence, false sharing, compiler issues.

*This work is supported in part by the NSF PYI Award MIP-9058430.

1 Introduction

There are two main problems to be solved in realizing scalable shared memory multiprocessors: latency for memory accesses, and network contention generated by these memory accesses. Latency for memory accesses arises in two contexts: first due to accesses to normal read/write data; and second due to accesses to synchronization variables. It turns out that the latter has more potential for causing network contention than the former due to the possibility for simultaneous access to the same synchronization variable from several processors [PN85]. Network contention may be considered a second order effect induced by the shared memory accesses, and an efficient solution to combat the latency problem can go a long way in reducing the network contention.

The reality is that latency cannot be eliminated and the best that can be done is to reduce or hide it by architectural innovations. Techniques for latency hiding include paying attention to the model of memory presented to the programmer. Traditionally, the model assumed by the programmer is that the contents of the shared memory is identical at all times from all the processors. Further, the model assumes that the completion order of the memory references from a single processor is strictly in program order. Both these assumptions restrict the scalability of shared memory multiprocessors. For example, the second assumption prohibits out-of-order completion of memory accesses which may be important to enhance performance, especially when the memory latency is high. In parallel applications, it is not unusual to use synchronization operations to ensure the consistency of shared data. In such cases, a temporary inconsistency in the views of the shared data as seen from different processors may be tolerable in certain ranges of the program, e.g., inside a critical section and between barrier synchronization points.

A second approach to hiding latency is the time-tested technique of associating private caches with each processor and devising an efficient protocol (either in software or hardware) for maintaining coherence of the (potentially) multiple copies of data in the caches. An associated consideration in this approach is to increase the cache line size (i.e. the unit of transfer between the cache and main memory) with a view to exploiting spatial locality. While increasing the line size usually tends to be advantageous in uniprocessors, it may not be the case in multiprocessors unless care is taken either in hardware or software to eliminate false sharing across processors (cache lines appearing shared from coherence standpoint even though they are not from program standpoint). Efficient prefetching techniques can complement and/or supplement increased line size and thus may be used as another technique for latency hiding. Multithreading the processor and providing rapid context switching between the multiple contexts is yet another technique for latency hiding.

Synchronization latency and the associated network contention that could ensue are best tolerated via explicit hardware support for synchronization. Further, distinguishing between normal read/write accesses and synchronization accesses in hardware enables efficient implementation of weaker notions of memory consistency [LR91b].

"Beehive" is a project that addresses some of the hardware and software issues in the design of scalable shared memory multiprocessors. The rest of the paper is organized as follows: We trace the evolution of memory consistency models and their chronology in Section 2. With respect to a specific memory consistency model we outline the implementation issues in Section 3. The architectural features of Beehive are presented in Section 4, and the inter-relationship between the features and memory consistency models are discussed. Section 5 discusses the implications of these features from the point of view of the system software. In particular, with respect to a specific programming language developed for parallel computing, we present techniques for using the language constructs for exploiting the architectural features. Such techniques can then be incorporated into the compiler backend for Beehive. Concluding remarks and directions for future research are presented in Section 6.

2 Consistency Models

To allow reasoning about programs written for shared memory multiprocessors, it is important to clearly specify the programming model for such machines. The programming model is defined by the memory consistency model, which specifies the order of execution of the memory accesses from independent processors.

2.1 A Chronology

Sequential Consistency (SC) has been proposed by Lamport [Lam79] as the ordering constraint for the correct execution of multiprocess computations. This memory model requires the order of execution of the independent access streams to be any arbitrary interleaving of the streams which preserves the relative order in each access stream. All accesses are treated equally by SC regardless of the type of data being accessed (private or shared) and the nature of the access being performed (read/write or synchronization). SC makes it easy to reason about programs since the ordering of the multiprocess computation is expressed in terms of a sequential execution of the multiple streams. However, it is not uncommon to use explicit synchronization primitives

in parallel program design to ensure a specific ordering among the independent streams. In such a case, there need not be a single global order that has to be observed by all the streams since the relative order is constrained by the explicit synchronization. Therefore, SC over-specifies the ordering when explicit synchronization is used.

Recognizing this fact, Dubois et al. [DSB86] have proposed the *Weak Consistency (WC)* model that relaxes the ordering constraint of SC by distinguishing between accesses to synchronization variables and ordinary data. WC requires (a) that synchronization accesses be sequentially consistent, (b) that all global data accesses preceding a synchronization access in a given stream be globally performed before issuing that synchronization access, and (c) that all global data accesses subsequent to a synchronization access in a given stream be delayed until the synchronization access is globally performed. In the rest of this paper we use the terms *performed* and *globally performed* in the same sense as defined by Dubois et al. [DSB86, SD87]. The *Release Consistency (RC)* model [GLL+90] goes one step further to distinguish between two kinds of synchronization accesses, namely, *acquire* and *release*. The main difference in the ordering constraint of RC with respect to WC, is that RC requires that constraint *b* above apply only for a release access, and constraint *c* apply only for an acquire access.

At about the same time that RC was developed, we proposed *weak coherence* [LR90a] and Adve and Hill proposed *DRF0* [AH90] both of which have very similar ordering constraint as RC. One significant difference between our model and RC is that the constraint *b* of WC above is further weakened in that we require only the global write accesses (as opposed to all global data accesses) before the release access to be globally performed. The name weak coherence has since been changed to *Buffered Consistency (BC)* [LR91b] to reflect the architectural features that we propose to support this model (see Section 4).

2.2 Buffered Consistency

The BC memory model recognizes two types of accesses: *data*, and *synchronization*. Data accesses (reads and writes) may be to *private* or to *shared* data. The synchronization accesses themselves are further subdivided into non-consistency preserving (*NP-Synch*) and consistency preserving (*CP-Synch*) accesses. NP-Synch and CP-Synch are always global accesses. BC requires synchronization accesses from a given stream to be globally performed in the order of issue. However the interleaving of synchronization accesses from arbitrary streams may be observed differently from other streams, i.e., the synchronization accesses need not be SC. The issue of an NP-Synch access does not require

the preceding data accesses from the same stream to be globally performed. Shared data accesses following an NP-Synch access from a given stream cannot be issued until the NP-Synch access is performed. A CP-Synch access is not issued until all preceding writes to shared data from the same stream are globally performed. But data accesses subsequent to a CP-Synch access in the same stream may be issued before the CP-Synch access is performed. Figure 1 shows these constraints and compares them to those of WC and RC.

The programming model assumed by WC, RC, and BC are the same. The weakening in the memory model of BC with respect to RC is exploited at the level of the system software (such as the compiler). The programming model and the compiler issues are further elaborated in Section 5.

3 Implementation Issues

The main purpose of weakening the memory model is with an expectation that such weakening would also have a correspondingly lesser constraint on the implementation thus resulting in enhanced performance. This section presents the separation of functions in hardware and software for the implementation of BC in the context of a cache-based system. Minimally, the processor-cache interface should provide for global accesses, and a *fence* that would ensure that the processor is blocked until all pending global writes are globally performed. Given this interface, the software can enforce the BC model by distinguishing between private and shared accesses, by identifying the nature of these accesses (read/write or synchronization), and by determining when to perform fence operations. In Beehive, in addition to the minimal interface the hardware is responsible for the following functions:

- distinguishing between local (to the cache) and global (eventually performed throughout the system) accesses,
- providing buffering for global writes thus smoothing the traffic on the interconnection network, and shielding the processor from the latency for global writes,
- maintaining coherence of the caches during global accesses under software control,
- providing synchronization operations in hardware, and
- eliminating false sharing.

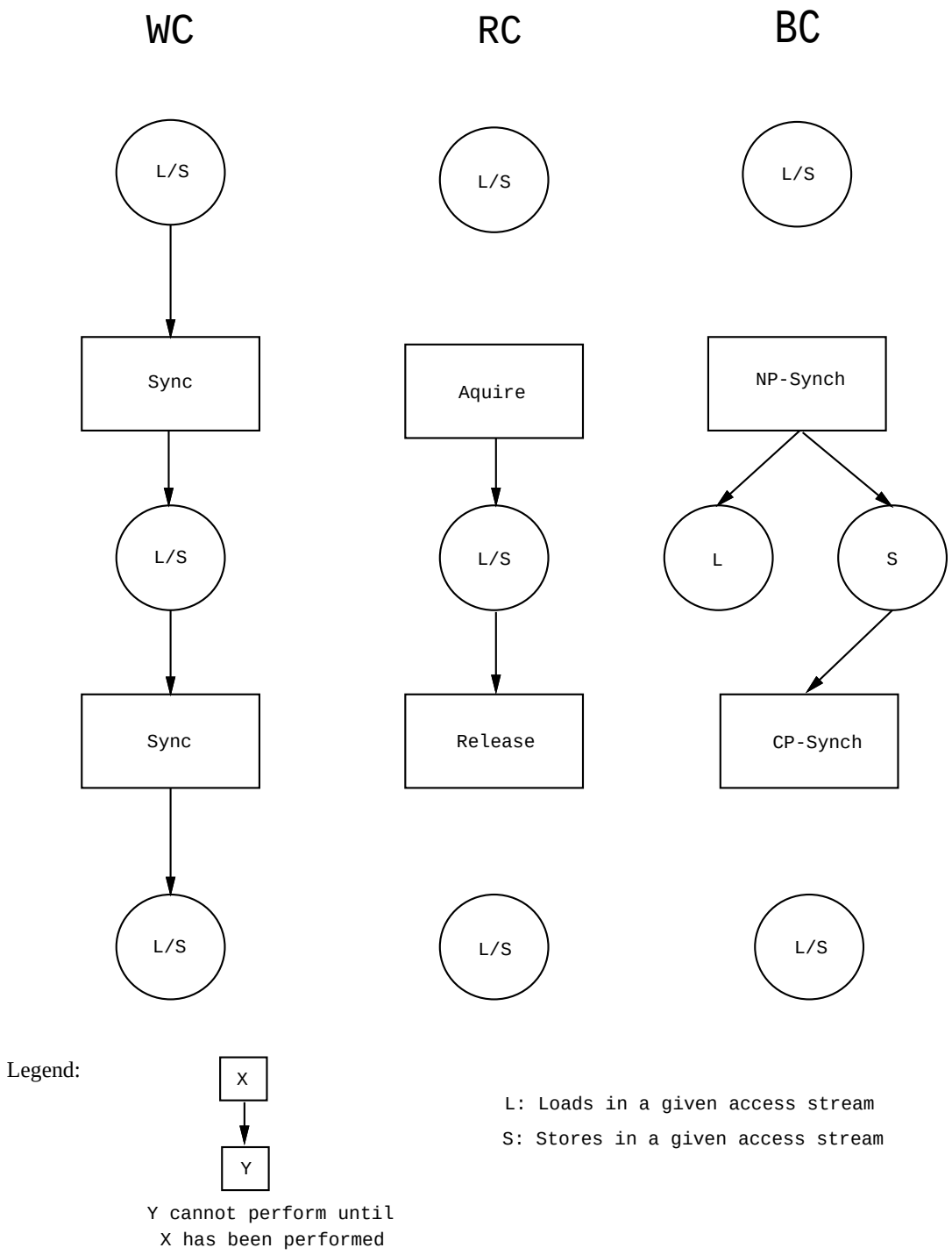


Figure 1: Model Constraints

3.1 Rationale

Since we expect most programs to be written in high-level language it is justifiable to make the software responsible for distinguishing between shared and private variables. In most cache-based shared memory architectures this distinction is made by the hardware dynamically based on the series of accesses from different processors to a given memory location. Similarly, differentiating between data accesses and synchronization accesses is achievable in software. Since the software can make these distinctions it is the best entity for deciding if and when to execute hardware fence operations, commensurate with program semantics and the memory consistency model. Beehive provides synchronization support mainly to hide the latency for synchronization operations. The memory consistency models define the ordering of accesses on shared variables (data and synchronization). However, cache-based shared memory architectures seldom provide features for decoupling the type of data from whether the access should be performed locally or globally. By providing local and global accesses in hardware, and recognizing shared and private data in software, we achieve this decoupling. For example, all operations on shared data need not be global. The software is the best judge of deciding when operations on shared data have to be global depending on the program semantics (see Section 5).

We are not aware of any cache-based architecture that solves the false sharing problem in hardware. It is upto the software to minimize the performance penalty that may result due to false sharing in such architectures. However, it has been recognized that false sharing is quite a difficult problem to solve in software in the general case [EJ91]. It turns out that with the consistency preserving mechanisms of Beehive it is not just a performance penalty but a correctness issue if false sharing is not eliminated. We show in the next section that with minimal overhead we can solve this problem in hardware.

4 Architecture of Beehive

A summary of the architectural features of Beehive were presented in an earlier paper [LR91b], which focussed on the performance implications of these features. We re-visit these features in this section to give the rationale behind them, present an exposition of the interaction between these architectural features, and set the stage for how they may be used to advantage by the system software.

Figure 2 shows the block diagram of a node in Beehive. Beehive is a distributed shared memory

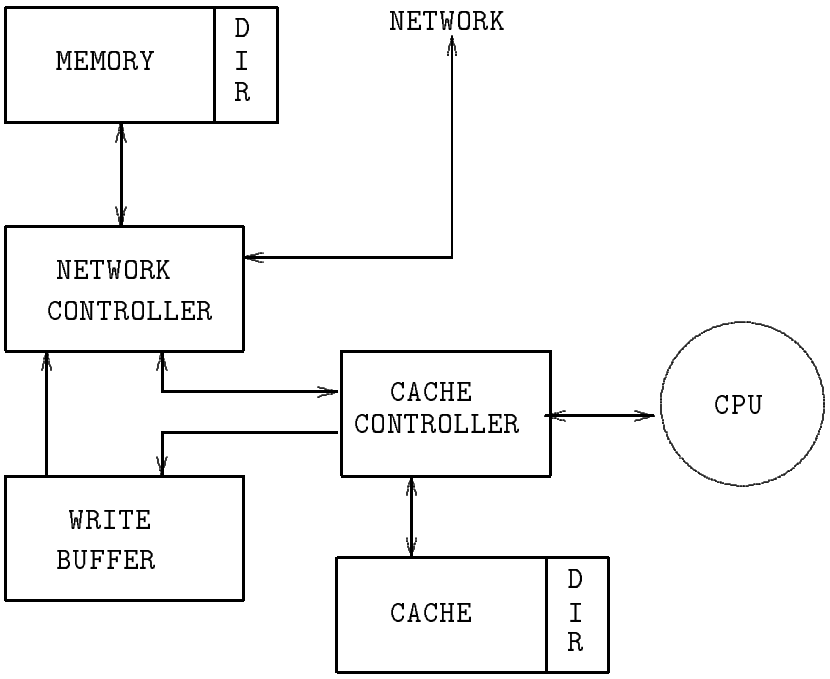


Figure 2: Node Architecture of Beehive

machine. Each node has a private cache and an associated directory. A piece of the global shared memory resides on each node with its associated directory. Beehive uses a directory-based protocol [CFKA90] for maintaining cache coherence. The write-buffer is provided for buffering global writes. Each node communicates with the rest of the system through the network controller. The interconnection network is intentionally left unspecified since the architectural features of Beehive do not rely on any specific characteristic of the network.

4.1 Read/Write Primitives

In multiprocessor caches, coherence actions are usually enforced blindly on writes, either via invalidation or update. Both approaches may incur unnecessary overhead since a location referred to in the past may not be referred to again in the future. This overhead surfaces in the form increased latency for global writes and increased network traffic. In large-scale multiprocessors with long latency for inter-node communication, and large cache sizes it is expected that this overhead would be more acutely felt. Ideally it is exactly those processors using a particular location in the future who should get the updates when the writes for that location are performed globally. We attempt to get toward this ideal by providing the notion of global and local writes, and by

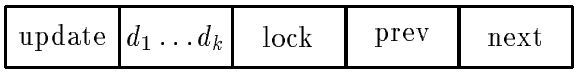
Instruction	Operations
READ	retrieve data without coherence maintenance
WRITE	write data without coherence maintenance
READ-GLOBAL	read data from the shared memory, bypassing local cache
WRITE-GLOBAL	globally perform the write
READ-UPDATE	retrieve data from shared memory and request future updates
RESET-UPDATE	cancel the request for updates from shared memory
FLUSH-BUFFER	stall the processor until all the requests in the write-buffer are globally performed
READ-LOCK	request a shared lock for a memory block
WRITE-LOCK	request an exclusive lock for a memory block
UNLOCK	release the lock

Table 1: Hardware primitives

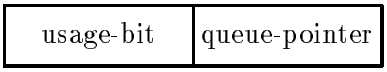
requiring the readers to explicitly request updates to cached locations. Note that these updates will be propagated only on global writes. We refer to this strategy as *reader initiated* coherence.

The primitives provided by the cache are summarized in Table 1. *Read* and *write* are treated as local operations if the data is in the cache, i.e., the semantics of these operations are exactly the same as in a uniprocessor write-back cache. *Read global* bypasses the cache and retrieves the data from the shared memory. *Write global* specifies that this operation has to be performed globally. The primitive that allows readers to hear updates to shared locations is *read update*. This primitive specifies that the data is retrieved from shared memory the 'first' time this operation is executed, and instructs the shared memory to propagate future updates to this location. *Reset update* informs the shared memory that updates to the specified location need not be propagated in the future to this processor.

Figure 3 shows a cache directory entry (CE) and a shared memory directory entry (DE). The *update bit* in CE is set (if it is not already set) when a read-update for a word in that cache line is issued. Note that the update bit is for the entire cache line, and there may be multiple words per cache line depending on the line size. Read-update is a local operation if the bit is already turned on; otherwise, the data is fetched from the shared memory and the *queue pointer* in DE is set to point to the requesting processor. The set of read-update requestors for the same memory block are linked together in a doubly-linked list structure through the *prev*, and the *next* pointers of the



a. An entry in the cache directory (CE)



b. An entry in the Memory directory (DE)

Figure 3: Structure of directory entries

participating CEs and the queue-pointer of the DE. Note that there is a distinct list associated with each memory block and its corresponding cache lines. When a write-global updates the shared memory this linked list is used by the associated DE to propagate updates to the participating cache lines. A reset-update to a specific location, or a cache line replacement deletes that processor from the appropriate list.

If each cache line has k words then $d_1 ... d_k$ are the dirty bits associated with a corresponding word in the line. Both write and write-global operations set the corresponding dirty bit in the cache line. The dirty bit is cleared when an acknowledgement for the corresponding memory write (either because of write-global or cache line replacement) is received. When a cache line is replaced only the dirty words are written back to the memory. Similarly, when a block is fetched from memory (due to a read-update request for a line that is already present in the cache) only the words that are not dirty in the line are filled. It can be easily illustrated that with our reader-initiated coherence, which has no concept of ownership of a cache line, there is a correctness problem without having multiple dirty bits, one for each word of the cache line. Consider two private variables v_1 and v_2 belonging to processors P1 and P2 respectively, colocated in the same cache line. If both v_1 and v_2 are updated and eventually this line is replaced by P1 and P2, only one of the two private variables will have the correct value if there is a single dirty bit per cache line. As a nice side benefit of solving this potential correctness problem in hardware, we also eliminate the ill-effects of false sharing.

We have chosen a pointer-based directory structure since it is more scalable than either a full-map or a limited map directory structure from the point of memory requirement [Ste90]. A criticism against such a structure is the latency for serial propagation of invalidations or updates, and the inability to use multicast. However, since the update propagation happens in the background without stalling the processor that issued a write-global, we do not expect this to be a severe

performance penalty. Further, it is not clear whether the dynamic level of sharing is high enough to lead to long serial latency. Many of these issues need further investigation through simulation studies and are currently underway.

4.2 Buffer Primitives

To hide the latency for write-global from the issuing processor we provide the write buffer. The write-global operation simply puts the request in this buffer and completes immediately allowing the processor to continue without having to wait for this write to be globally performed. However, before a CP-Synch access is issued there may be a need to know that all the preceding write-global operations have been globally performed. *Flush buffer* is a primitive that is provided for this purpose. This primitive stalls the issuing processor until the writes pending in the write buffer are globally performed.

4.3 Synchronization Primitives

Synchronization latency has two components: The first is the potential wait times at synchronization point due to simultaneous access from parallel processors. The second is the intrinsic overhead for implementing the synchronization. An atomic *read-modify-write* type of operation is sufficient to implement higher level synchronization primitives but this could lead to bursty traffic on the release of a mutual exclusion lock. Recent work [MCS91] has shown that it is possible to implement locks and barriers with minimal network traffic. In spite of reducing the network traffic, these software algorithms still have to pay the latency penalty for synchronization operations. Therefore, we have provided synchronization support in hardware. Further, providing synchronization operations in hardware aids an efficient implementation of the BC memory model.

We provide *read lock* (a shared lock), and *write lock* (an exclusive lock) both of which are NP-Synch type of operations. *Unlock* relinquishes the lock and is a CP-Synch type of operation. Locks are implicitly associated with a cache line, and granting of a lock request is combined with the transfer of the data to the requestor. This combination of data transfer with locking is another attractive reason for implementing these locks in hardware. The same hardware queue structure used for propagating updates for read-update requests is used for maintaining a FIFO queue of lock requestors for the same cache line. The *usage bit* in DE denotes whether the associated list structure is for lock requests or read-update requests. The *lock* field in CE denotes the state of the associated lock request. Since the same queue structure is used for both, care has to be taken in

data allocation (see Section 5). To overcome the problems associated with replacement of cache lines participating in a lock operation, we implement a small separate fully-associative lock cache. Thus even though an implicit lock is associated with every memory block, the size of this cache places limitations on the number of hardware locks that can be active in a processor at a given time, and needs careful resource management by the software (see Section 5).

Performance implications (through analytical and simulation studies) of reader-initiated coherence, the BC memory model, and queue-based locks may be found in [LR91b]. The protocols and performance implications of incorporating synchronization in multiprocessor caches is discussed in [LR90b].

5 Exploiting the Features of Beehive

The programming model that corresponds to the BC memory model assumes that the data is divided into three classes: private variables, shared variables, and synchronization variables. In the programming model, there are two types of accesses to synchronization variables corresponding to the NP-Synch and the CP-Synch accesses of BC. Correspondingly, the programming model guarantees that coherent values of shared data become visible to concurrent threads that comprise a parallel program only after the completion of synchronization operations that fall into the CP-Synch category. Programs that fall into this class have been referred to as *properly labeled (PL)* programs [GLL+90].

The language used for writing programs for Beehive should support the above programming model. Further, to exploit the multiple processors in the architecture the language should have constructs to explicitly specify concurrency. Using such constructs it is possible to construct a task graph that represents the program. Given the programming model, the tasks themselves are comprised of *synchronization epochs (SE)*. An SE is a region of code that is terminated by a CP-Synch operation. The task graph generated from the explicit concurrency constructs in the language may restrict the true concurrency that may exist in the program. The true concurrency is really dictated by the inter-relationship between the SEs that comprise the tasks.

The language Jade [LR91a] has constructs that match well with our intuition of synchronization epochs. Jade separates the notions of synchronization and concurrency thus allowing the exploitation of SE-level concurrency. The language requires that the programmer specify *side effects* (which could be arbitrary code) on the shared data it accesses, while using synchronization and concurrency constructs. The three Jade constructs that are relevant to the discussion in this section are

with, *withonly*, and *withth*. The "with" construct is used to signify execution of the associated scope under the protection of the synchronization governing this scope specified in the side-effect; the "withonly" construct is used to signify concurrency in addition to specifying the set of intention locks on shared objects over which the associated scope applies; and the "withth" construct is similar to "with" but allows concurrent execution of the associated scope. The construct "without" allows explicit release of intention locks acquired by an enclosing "withonly" construct. The usage of "withth" primitive is illustrated in Figure 4. The other constructs have similar syntax.

The fact that BC model allows loads inside an SE to complete after the corresponding CP-Synch point, does not affect the programming model as defined above. This property is really something the compiler can exploit to reorder loads that occur after a CP-Synch operation in the program to further help in latency hiding. This situation is similar to the one in pipelined uniprocessors such as MIPS [GHPR88], wherein there are no hardware interlocks, and the compiler is entrusted with the responsibility of ensuring that the dependencies in the sequential program are respected. In a similar vein, we expect the compiler for Beehive to ensure that loads to shared data occurring in the original program before a CP-Synch are completed before issuing the CP-Synch operation. However, the loads that have been hoisted above the CP-Synch point during the optimization phase of the compilation do not have to be completed before issuing the CP-Synch operation.

The rest of the section deals with using the language features of a 'Jade-like' language to exploit the architectural primitives of Beehive. The issues that need to be resolved in this context are:

1. mapping the loads and stores in the program to the read/write primitives of Beehive commensurate with the program semantics while enhancing performance,
2. exploiting hardware locks of Beehive to implement the user level synchronization,
3. identifying limitations of Beehive with respect to thread migration and processor reassignment.

5.1 Marking Algorithms

While the algorithms presented are in the context of Jade, the discussions are general and are applicable to any language that allows the explicit specification of synchronization and concurrency. All of the discussion is predicated on the fact that the language (and hence the compiler) explicitly recognizes the synchronization points in the program from the source code. The start of the scope of "with" and "withth" constructs signify the opening of an SE, and therefore correspond to the

```
withth {x.wr(); /* side-effect specification*/}
        (/* parameters for task */) {
    x=f1();
}
```

Figure 4: Use of *withth* in Jade

NP-Synch point with respect to the BC model. Similarly, the end of the scope of "with" and "withth" corresponds to CP-Synch points.

In order to map the loads and stores to the read/write primitives of Beehive, we need to construct a program dependence graph (PDG). The only information that we need to glean from the side-effect specification is a list of locks on shared data (if any) and their types (exclusive, shared, intention) that are needed in the scope of the associated construct. This specification list is needed by the marking algorithm to be described shortly. Every shared data that is accessed within an SE needs to be explicitly specified in this list for the marking phase to work correctly. For example, if the program uses a *mutex* variable to govern access to a set of shared data, then that set needs to be enumerated in the specification list. On encountering either a *with* or a *withth* special nodes $SEstart$ and $SEend$ are created in the PDG to mark the beginning and end of a new SE, respectively. The list of locks on shared data mentioned in the side-effect specification of the construct is maintained with these nodes. The rest of the dependence graph is constructed in the normal way [FOW87]. The main difference between the PDG constructed in [FOW87] and the one we construct is that in our graph there are additional nodes that explicitly signify all the loads and stores in the program.

Once the PDG has been constructed the mapping of loads and stores in the PDG to the read/write primitives of Beehive proceeds as follows. In Beehive the type of operation (local or global) is decoupled from the type of data (private or shared). Thus it is possible for the compiler to exploit this architectural feature to track the BC memory model more closely than other architectures [LLG+90]. For example, loads and stores outside of the SEs are marked as reads and writes, respectively, which are local operations if the data is already in the cache (see Section 4). Note that this rule applies regardless of whether the data being accessed is shared or private, since in the BC memory model coherence of shared data is enforced only at the termination of an SE; and the program should not make any assumptions regarding the coherence of shared

data outside of the SEs. In other words, we do not provide "true shared semantics" for loads or stores to shared data not enclosed by SEs.

Inside an SE accesses to private data are marked as either reads or writes. In the BC memory model, only the *last* store in a particular SE to a given shared data need be globally performed; all the preceding writes to the same data within this SE need not be global. The marking algorithm, *write-global-mark* (see Figure 5), identifies the stores that need to be write-global. The algorithm does a reverse traversal of the PDG every time an $SEend$ node is encountered in the graph. Starting from this node, we recursively follow every parental link one at a time until a store of a given shared data is reached in that ancestral subgraph, or the corresponding $SEstart$ is reached. The stores thus encountered are marked as write-globals. The other stores for the same shared data in this SE are marked as writes (local operations). The marking algorithm is performed for every item in the specification list of the SE.

Since we conservatively estimate that the programmer expects loads on shared data within SEs to follow true shared semantics, we mark these loads as read-updates (see Section 4). However, upon exiting an SE the task (and therefore the processor it is executing on) may not need to get updates to these shared data. The reset-update primitive is used for this purpose. A marking algorithm very similar to the one described above is used for determining the last load of a given shared variable and performing a reset-update immediately following that load. Upon reaching an $SEend$, the compiler needs to generate a flush-buffer since an $SEend$ signifies a CP-Synch point of the BC model. This flush-buffer should be executed before any locks associated with this SE are released.

5.2 Storage Allocation

In Beehive the domain of an access is decoupled from the type of data. This decoupling may be exploited by the compiler (as detailed above) to reduce the number of global accesses. Note that consistency is maintained only when accesses are performed globally (i.e. write-global, read-update). This is in contrast to other shared memory architectures such as [LLG+90] where there is a concept of *ownership* of a cache line at all times regardless of whether the line contains shared or private data. The absence of ownership in Beehive is the key aspect that eliminates false sharing. Since in Beehive there is no false sharing, private variables of distinct tasks and shared variables can co-exist in the same cache line. If this colocation were not possible, the compiler would have to perform extensive analysis and possibly use heuristics to minimize the ill-effects of false sharing

```
write-global-mark() {
  for all identifiers x with writes in the specification list of corresponding SEstart d
       mark(SEend, x);
}
mark(node y, identifier x) {
  for all nodes i in the dependence graph do {
       check[i] = false;
  }
  for each node i which is a parent of y do {
       if i is the matching SEstart then {
            check[i] = true;
       }
       if i has a store of x {
            mark the store as write-global;
            check[i] = true;
       } else {
            mark(i, x);
            check[i] = true;
       }
  }
}
```

Figure 5: Write-global marking

by careful data placement [EJ91].

Beehive provides implicit queue-based locking primitives that are associated with each cache line. To implement the language recognized synchronization constructs, the compiler or runtime may choose to use these primitives. However, care has to be taken in allocating shared variables when these implicit locks are in use. First there is only one implicit lock per cache line; second the same hardware queue structure is used for both update propagation as well as lock maintenance. Therefore, only one shared data object can be allocated per cache line when used in conjunction with an implicit lock. Note that any number of private variables may be colocated in the same cache line. However, if the synchronization constructs are implemented using software mutual exclusion algorithms [MCS91, PS85], then colocation of shared data objects in the same cache line is not a problem.

As we mentioned in the earlier section, (implicit) hardware locks that can be active in a processor at a given time are limited by the size of the fully-associative lock cache. The system (i.e. compiler and/or runtime) has to ensure that there will never be a need to break a hardware queue that is currently being used for maintaining a set of lock requestors due to cache line replacement. This requirement means that the system has to know at all times that the set of active locks will never exceed the size of the lock cache. This requirement can be fulfilled in one of two ways: by conservatively allocating at compile time a fixed number of hardware locks and simulating the rest in software; or by the runtime keeping a count of the dynamic usage of locks and blocking a lock requestor when the count exceeds some threshold.

5.3 Concurrency Management

We noted earlier that there is no concept of ownership of a cache line in Beehive. The absence of ownership has implications on task migration in Beehive. Consider the following scenario: A task that has executed for a while on a processor P1 is migrated to another processor P2. The private variables of this task may be in a dirty state in the cache on P1. Due to the lack of ownership, when the task starts executing on P2 accesses to these variables will not result in fetching the dirty values that are currently cached in P1. This scenario illustrates correctness problems that could ensue if task migration is allowed in Beehive. This problem could be handled if the architecture provides mechanisms for selective "purging" of dirty cache lines. However, Beehive does not provide such a feature currently since it is not clear if such task migration is necessary or even justified in large-scale multiprocessors. Therefore, we currently disallow task migration.

A similar problem occurs with respect to shared variables when a task T1 spawns another task T2, and the system decides to execute T2 on a different processor. This situation can be handled as illustrated in Figure 6 where we use *fork* to signify any general concurrency construct. In T1's PDG the fork results in the creation of an $SEend$ followed by an $SEstart$. In T2's PDG the concurrency construct that created T2 itself results in an SE. The marking algorithm described above would then ensure that the writes to shared data are globally performed before T2 starts executing on a different processor. However, note that executing multiple tasks that share data on the same processor does not cause any correctness problem. In all of the discussion above relating to concurrency we assume that the language runtime is responsible for maintaining the task dependence relationship that is inherent in the parallel program.

The work presented in this section is inspired by the kind of work done by Cytron et al. [CMM88]. While their work deals with analyzing and exploiting loop-level parallelism for programs using the SC memory model, our work is applicable to arbitrary synchronization and concurrency scenarios in the context of Beehive for the BC memory model. It is interesting note that while Beehive is primarily intended to support the BC memory model, SC memory model may also be simulated quite easily. This simulation is achieved by restricting the program to use only read-global for all program loads, and write-global immediately followed by a flush-buffer for all program stores.

6 Concluding Remarks

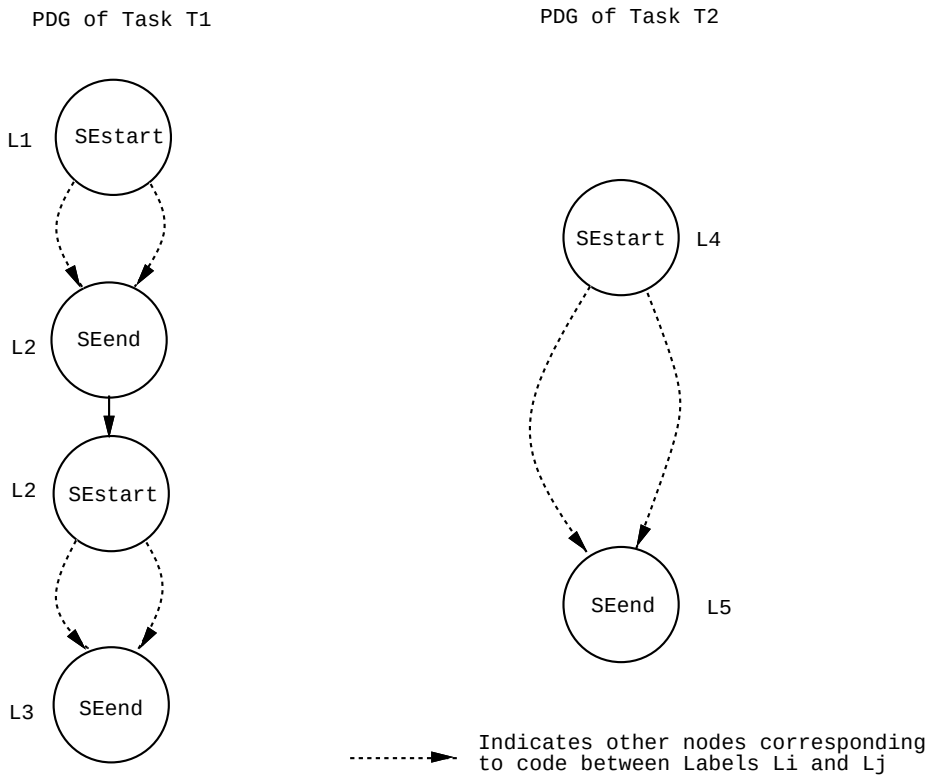
Programming using the shared memory paradigm is well understood. Conceptually, a global shared memory appears as a central resource and hence a point of potential bottleneck for the scalability of this machine model compared to the message-passing style of architecture. However, recent architectural trends such as weaker memory models, physical distribution of the shared memory, efficient prefetching techniques, and the aggressive use of caches make the underlying architecture look more like message-passing style while preserving the semantics of shared memory programming paradigm. Beehive provides architectural primitives for supporting a weak memory model called Buffered Consistency. The novel features of the architecture are decoupling the notion of the type of data (private or shared) from the domain (local or global) over which the access should be performed; eliminating false sharing; providing for software assisted consistency maintenance using reader-initiated coherence; and supporting shared and exclusive queue-based locks in hardware. Techniques for exploiting some of these feature from the point of view of both latency hiding and

```
L1: Task T1 {
    .  .  .
    L2: fork(T2);
    .  .  .
L3: }


L4:  Task T2 {
     .  .  .
L5:  }

/* Note that the same labels Li are used to correlate the program
   fragment with the corresponding nodes in the PDG */
```

a: Program Fragment showing a Fork



b: PDG for tasks T1 and T2

Figure 6: Handling Concurrency through SE's

reducing the network traffic were presented. These techniques are amenable for use by a compiler and runtime designed for a parallel programming language.

There are numerous issues that need careful investigation: automatic management of hardware locks, reordering of loads and stores to further help in latency hiding, evaluation of the need for and the design of mechanisms to handle task migration, evaluation of prefetching techniques and their incorporation in Beehive, and detailed performance evaluation of the mechanisms provided in Beehive. In general, evaluation of architectural features is an arduous task. Especially, in our architecture we have identified a division of responsibility between the hardware and the software. This division coupled with the choice of primitives leads to a complex interplay making performance evaluation a challenging task.

References

[AH90] S. Adve and M. Hill. Weak Ordering - A New Definition. In *17th Annual International Symposium on Computer Architecture*, pages 2–14, May 1990.

[CFKA90] David Chaiken, Craig Fields, Kiyoshi Kurihara, and Anant Agarwal. Directory-based cache coherence in large-scale multiprocessors. *IEEE Computer*, 23(6):49–58, June 1990.

[CMM88] R. Cytron, S. Marlovsky, and K. P. McAuliffe. Automatic management of programmable caches. In *17th International Conference on Parallel Processing*, pages II–229–238, August 1988.

[DSB86] M. Dubois, C. Scheurich, and F. Briggs. Memory access buffering in multiprocessors. In *13th Annual International Symposium on Computer Architecture*, pages 434–442, June 1986.

[EJ91] Susan J. Eggers and Tor E. Jeremiassen. Eliminating false sharing. In *20th International Conference on Parallel Processing*, pages I–377–381, August 1991.

[FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[GHPR88] Thomas R. Gross, John L. Hennessy, Steven A. Przybylski, and Christopher Rowen. Measurement and evaluation of the MIPS architecture and processor. *ACM Transactions on Computer Systems*, 6(3):229–257, August 1988.

[GLL+90] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. L. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *17th Annual International Symposium on Computer Architecture*, pages 15–26, May 1990.

[Lam79] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[LLG+90] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In *17th Annual International Symposium on Computer Architecture*, pages 148–159, May 1990.

[LR90a] Joonwon Lee and Umakishore Ramachandran. Locks, directories, and weak coherence - A recipe for scalable shared memory multiprocessors. In *1990 ISCA Workshop on Scalable Shared-Memory Multiprocessors*, May 1990. To appear in *Scalable Shared Memory Multiprocessors*, Kluwer Academic Publishers, 1991.

[LR90b] Joonwon Lee and Umakishore Ramachandran. Synchronization with multiprocessor caches. In *17th Annual International Symposium on Computer Architecture*, pages 27–37, May 1990.

[LR91a] Monica S. Lam and Martin C. Rinard. Coarse-grain parallel programming in Jade. In *Third ACM SIGPLAN Symposium on the Principles and Practices of Parallel Programming*, pages 94–105, April 1991.

[LR91b] Joonwon Lee and Umakishore Ramachandran. Architectural primitives for a scalable shared memory multiprocessor. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 103–114, July 1991.

[MCS91] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.

[PN85] G. F. Pfister and V. A. Norton. Hotspot contention and combining in multistage interconnection network. *IEEE Transactions on Computers*, C-34(10):943–8, October 1985.

[PS85] James L. Peterson and Abraham Silberschatz. *Operating System Concepts*. Addison-Wesley Publishing Coompany, 1985.

[SD87] C. Scheurich and M. Dubois. Correct memory operations of cache-based multiprocessors. In *14th Annual International Symposium on Computer Architecture*, pages 234–243, June 1987.

[Ste90] Per Stenstrom. A survey of cache coherence schemes for multiprocessors. *IEEE Computer*, 23(6):12–24, June 1990.