2014 IEEE 28th International Parallel & Distributed Processing Symposium Workshops

# *ABC*$^2$: Adaptively Balancing Computation & Communication in a DSM cluster of Multicores for Irregular Applications

Sai Charan Koduru, Keval Vora, Rajiv Gupta
Department of Computer Science and Engineering
University of California, Riverside
{scharan,kvora001,gupta}@cs.ucr.edu

***Abstract*—Graph-based applications have become increasingly important in many application domains. The large graph sizes offer data level parallelism at a scale that makes it attractive to run such applications on distributed shared memory (DSM) based modern clusters composed of multicore machines. Our analysis of several graph applications that rely on speculative parallelism or asynchronous parallelism shows that the balance between computation and communication differs between applications. In this paper, we study this balance in the context of DSMs and exploit the multiple cores present in modern multicore machines by creating three kinds of threads which allows us to dynamically balance computation and communication: compute threads to exploit data level parallelism in the computation; fetch threads that replicate data into object-stores before it is accessed by compute threads; and update threads that make results computed by compute threads visible to all compute threads by writing them to DSM. We observe that the best configuration for above mechanisms varies across different inputs in addition to the variation across different applications. To this end, we design *ABC*$^2$: a runtime algorithm that automatically configures the DSM using simple runtime information such as: observed object prefetch and update queue lengths. This runtime algorithm achieves speedups close to that of the best hand-optimized configurations.**

***Keywords*-Distributed Shared Memory; Clusters; Runtime Monitoring; Dynamic Adaptive Model; Speculative Parallelism; Asynchronous Parallelism;**

## I. Introduction

Clusters offer an attractive computing platform for achieving scalable performance on data and compute intensive applications. To simplify the task of programming clusters, distributed shared memory (DSM) has been widely used. The memory resources available across the machines in a cluster are harnessed as one and made available to the application in form of shared-memory. The large amount of application data stored in DSM is actually scatterred across the machines and must be transferred across them as needed by the computation. A variety of DSMs have been built in the past [1], [2], [3]. These DSMs were developed before the advent of multicore machines. To deliver high performance, latency hiding mechanisms were deployed to mitigate the impact of communication delays associated with transferring data across machines: for example, creating multiple application threads was the common approach. The primary limitation of this approach is that computation and communication are still coupled: the communication is still in the critical path of computation. Moreover, with the advent of multi-cores, a much larger number of application threads becomes feasible – but that also increases communication which can quickly become the bottleneck on DSMs. Decoupling communication and computation will allow for dynamically balancing computation and communication – this will automatically achieve optimal performance even in the face of changes to the DSM/cluster HW/SW configuration. In this work, we study this balance in the context of DSM for applications that benefit from speculation [4], [5], [6] and asynchronous [7], [8], [9] parallelism.

### A. Communication Bottleneck on Multicore Machines

Since each machine in a modern cluster supports multiple cores, simultaneous requests for communication originating from multiple computation threads can rapidly cause the communication to become a performance bottleneck – if programs designed for prior DSMs and developed for a cluster of uniprocessor machines are naively executed on modern multicore machines by simply running more threads on each machine, the network becomes the bottleneck. For example, on a cluster of six 8-core machines, we observed that a 5x increase in the number of threads generating communication requests resulted in a 7x increase in fetch time from the DSM. In this work we exploit multicore machines to tolerate communication latency by introducing dedicated communication threads and move the communication latency off the critical path.

Computation being performed generates two types of network requests: *fetch* and *update*. In addition, to tolerate fetch latency, *prefetch* requests may also be issued. Fetch requests are of the highest priority since a computation must stall on a fetch. On the other hand, *prefetch* requests can be given a lower priority because they are issued to reduce latency of future fetch requests and *update* requests can be given lower priority because there may or may not be other computations waiting on the updates. Therefore, depending upon the spare network capacity, *prefetch* requests and *update* requests should be accomodated at a rate that does not overwhelm the network. Moreover, depending upon the needs of the application, a balance between handling of prefetch and update requests should be maintained. In this work, for each machine, the number of outstanding *prefetch* requests and outstanding *update* requests is controlled to: limit the amount of network capacity they use; and to maintain a balance between the two types of requests. We develop a system to achieve this goal that, for a given number of *compute* threads, dynamically varies the number of *prefetch* threads and *update* threads to meet the needs of the application.

As an example, our experiments show that merely increasing the number of compute threads can actually hurt the performance of distributed Graph Coloring – on a cluster of 6 machines, the speedup with 4 computation threads and 4 prefetch threads was 2.4x that of the speedup with 8 computation threads and 4 prefetch threads. We further observed that blindly increasing either prefetch or update threads in DSM based programs also can hurt performance. An optimal configuration that uses just 4 compute threads per machine and dynamically varies the number of update threads was faster than the case of 4 compute threads with 4 prefetchers by a factor of 1.2x. To summarize, this data illustrates that fewer computation threads when balanced with communication gives better speedup compared to more computation threads. Clearly, this data is an indicator for the need to carefully balance computation and communication in DSM based applications.

### B. Dynamically Adaptive Communication

This latency tolerance mechanism hides communication latency by creating communication threads that run concurrently with computation on the multicore machine. Two types of communication threads are used: *prefetch threads* and *update threads*. Prefetch threads fetch objects from the DSM into the prefetch buffer before they are accessed by the threads on a machine so that computation threads can avoid potential long access latency. Update threads are responsible for writing modified objects to the DSM so that computation threads can proceed with execution without waiting for object updates in DSM to complete. Given a fixed number of computation threads, the number of communication threads required will depend upon the rate at which the computation threads initiate fetch and update operations from and to the DSM. Therefore we must decide:

- *Degree of prefetching* - determined by the number of prefetch threads that are created to run concurrently with the given number of computation threads; and
- *Aggressiveness of updates* - determined by the number of update threads that are created to run concurrently with computation threads.

An appropriate balance between computation and communication threads must be dynamically maintained at runtime to optimize performance.

## II. Motivating Study

In this section, we present results of a study that motivates the conjecture that different applications require different balance between communication and computation. Table I lists the applications considered in this work and their characteristics. We consider two classes of applications: those with speculative parallelismm [4], [5], [6] and those with asynchronous parallelism [7], [8], [9]. We briefly describe these classes of applications in subsections Section II-A and Section II-B. Notice that applications with low sharing benefit from speculation while applications with high sharing are better off with asynchronous parallelism, wherever the application domain can allow for stale values in update.

TABLE I: A Suite of Modern Applications

| Application | Parallelism | | Sharing | |
|---|---|---|---|---|
| | Spec | Async | Low | High |
| Graph Coloring (GC) | √ | | √ | |
| KMeans (KM) | √ | | √ | |
| Single Src Shortest Path (SSSP) | √ | | √ | |
| Maximal Independent Set (MIS) | √ | | √ | |
| PageRank (PR/PR0) | | √ | √ | √ |
| Spam Filter (SF) | | √ | | √ |
| Wave Simulation (WS) | | √ | | √ |
| Heat Simulation (HS) | | √ | | √ |
| 2D Heat Simulation (2DHS) | | √ | | √ |

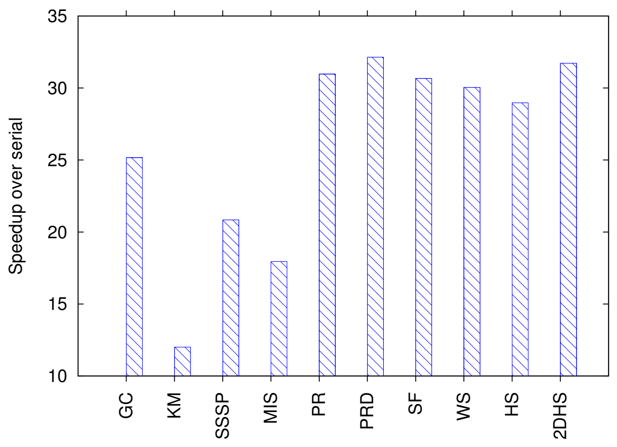


Fig. 1: Speedups of the parallel versions of the benchmarks over their serial versions.

We first present the raw speedups of the parallel versions (without using our techniques) over the serial versions. The experiments were conducted on the dyDSM system [10], which supports speculative and asynchronous execution. dyDSM's uses the the SpiceC [4] model for speculative execution. Figure 1 shows that before using our techniques, the speculative benchmarks achieve 30x speedups while the asynchronous benchmarks achieve about 40x speedup on average. Our goal is to *further* improve the speedups of these parallel benchmarks.

*In the rest of this study, the speedups we present are over the baseline parallel version that does not use separate prefetching or udpate threads. This allows us to objectively study the speedups that can be attributed exclusively to the mechanisms considered in this study.*

In the remainder of this section we describe two forms of parallelism observed in applications listed in Table I and, in each case, we evaluate the benefits from balancing communication and computation. As mentioned above, the baseline used is the parallel version without our techniques. This allows us to evaluate the specific benefit from using dedicated prefetch and update threads and dynamically adapting these resources.

## *A. Speculative Parallelism*

Speculative parallelism is a software technique that parallelizes algorithms by creating parallel threads to perform computations that may exhibit non-deterministic sharing of data [4], [5], [6]. To illustrate speculative parallelism let us consider the example of Graph Coloring. The serial algorithm for graph coloring works as follows – for each node in the graph, if the node is not yet colored, then assign a color to it based on the colors of its neighbors. If such a coloring is not possible, algorithm fails. The pseudo code for the serial graph coloring algorithm is given in Figure 2.

```
0. for(i=0; i<NUM_NODES; i++) {
1.  for(j=0; j<NODES[i].neighbors; j++) {
2.    // decide upon an unassigned color ‘c’
      // to assign to NODES[i]
3.  }
4.  NODES[i].color = c
5. }
```

Fig. 2: Pseudocode for the serial algorithm of Graph Coloring.

In order to parallelize the Graph Coloring algorithm, we can color mutiple nodes from the graph in parallel. However, in such a parallel version, a pair of neighboring nodes may be *concurrently* assigned colors by a pair of computation threads. As shown in Figure 3, the node $N_i$ being colored by thread $T_i$ is the neighbor of the node $N_j$ which is concurrently being colored by thread $T_j$. Therefore, the coloring information of node $N_i$ with thread $T_j$ and node $N_j$ with thread $T_i$ may indicate that the two nodes are not colored. This can lead to incorrect coloring assignment, i.e. the two nodes may be assigned the same color.

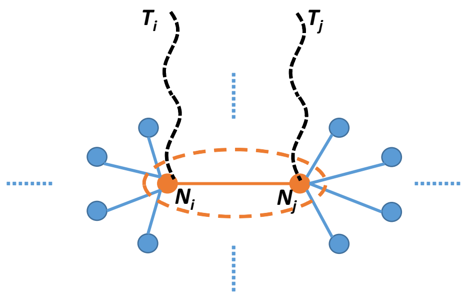


Fig. 3: Parallelizing Graph Coloring requires speculation.

On the face of it, it appears that an algorithm such as graph coloring cannot be parallelized. However, this is where speculation comes in. With speculation, there are two distinct phases: (1) computation and (2) commit. To begin, each computation thread *speculates* that the data it reads will stay current during the computation and proceeds with the computation and stores the results privately. Once the computation is complete, the thread attempts to commit the results in order to make these results visible to other computation threads. As part of the commit phase, a *mis-speculation check* is performed to assert that the data read by the thread is, in fact, still current. The result of a speculative computation is committed *only* if the mis-speculation check succeeds. On failure, the result of the computation is discarded and the computation is scheduled to be re-executed.

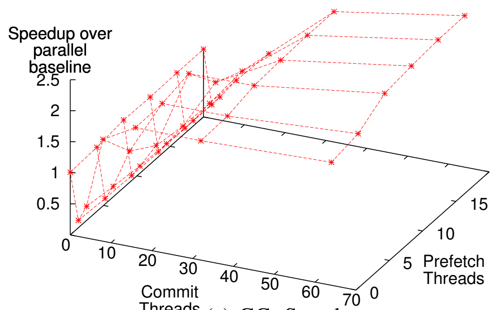
Therefore, in the presence of potential sharing between concurrently executing threads, the key idea behind speculation is to *speculate that such sharing does not occur* and proceed with the computation. Once the results of the computation are ready to be written to memory, a mis-speculation check is performed to assert the absence of such sharing. This can be achieved by resorting to versioning of data, for example. Thus, speculative parallelism only commits those results that are computed from most current values. With speculative parallelism, data commit failures can arise from violating write-read or read-write or write-write dependencies that manifest at runtime. In this work, we use the SpiceC [4] speculative execution model.

Applications generate high-priority *fetch requests* when the node to be processed is needed – these must be serviced immediately and the requesting thread blocks until this request completes. Once a node has been fetched, additional requests are issued to *prefetch* the neighbors. Finally, once the speculative computation is completed, an *update request* is issued to commit the updated private copy of the data. The update thread performs the mis-speculation check and performs the commit asynchronously and off the critical path of the speculative computation.

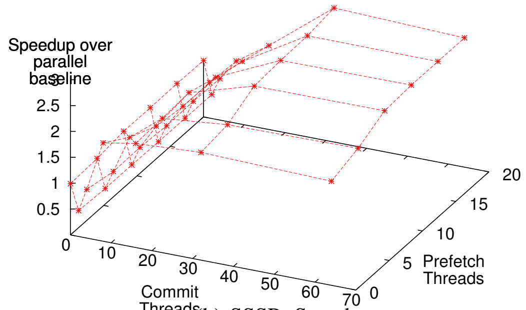
For speculation to be effective, the sharing of nodes between concurrently executing threads should be small. If such sharing is high, the potential benefits from speculation will be offset by repeated re-computations caused by commit failures that result from failed mis-speculation checks. Examples of applications that have low sharing and can benefit from speculative execution include: Graph Coloring (GC), Single Source Shortest Path (SSSP), KMeans (KM) and Maximal Independent Set (MIS).

We now present a study of the speculative benchmarks listed in Figure I. Figure 4 show the speedups Graph Coloring (GC), Single Source Shortest Path (SSSP), KMeans (KM), and Maximal Independent Set (MIS) with varying number of commit and prefetch threads on the lower axes. The baseline for the speedups is the configuration with zero prefetch and zero update threads. We now present the observations for communication.

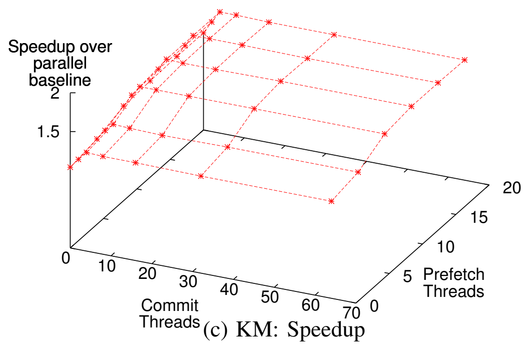
**Communication Strategy.** Asynchronously prefetching the data needed for computation in the near future can definitely help speedup the computations: this is seen in the speedups plotted in Figures 4a, 4b, 4c and 4d. This is especially true when there are a proportionately large number of neighbors to process for each node. In addition, speculation will benefit



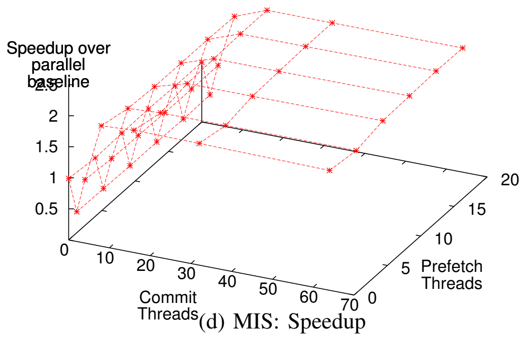
(a) GC: Speedup



(b) SSSP: Speedup

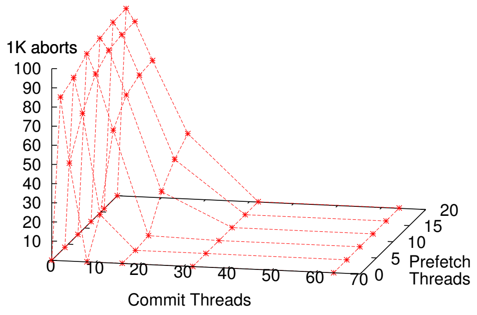


(c) KM: Speedup

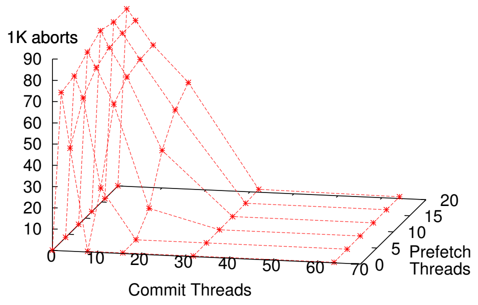


(d) MIS: Speedup

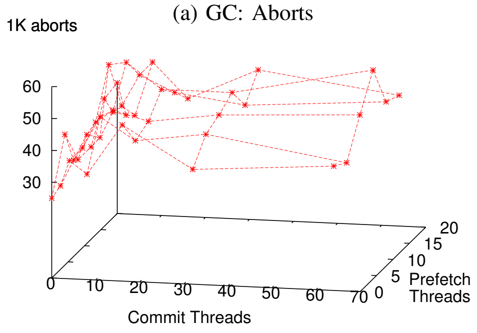
Fig. 4: Figures 4a, 4b, 4c and 4d show the speedups for GC, SSSP, KM and MIS respectively. Prefetch threads make remote objects locally available, hence avoiding remote accesses and reducing the average fetch time. Each benchmark was executed on a cluster of 6 machines, running 4 computation threads.



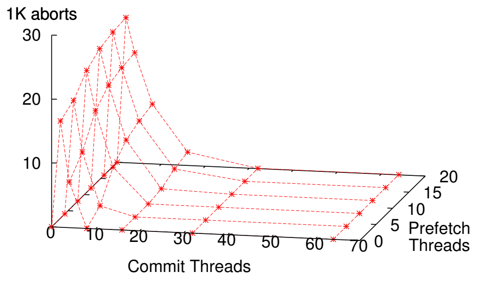
(a) GC: Aborts



(b) SSSP: Aborts



(c) KM: Aborts



(d) MIS: Aborts

Fig. 5: Figures 5a, 5b, 5c and 5d show the number of 1K aborts due to misspeculation for GC, SSSP, KM and MIS respectively. Lesser number of commit threads delay the availability of newly computed object values resulting in higher aborts of computations on stale data values.

from aggressive updates back to the DSM: this is apparent from the increasing speedup with an increasing number of commit threads in Figure 4. Delaying commits increases the mis-speculation rate as the computation threads continue to compute results based on potentially stale values. This can be observed in Figure 5 – the aborts are higher for fewer commit threads and rapidly decrease with increasing commit threads. Further, notice that in most cases when there are fewer commit threads, aggressive prefetching can result in a 10x increase in aborts. This is because prefetching aggressively brings in stale values since the newer values are still in the update queue. As a result, all computations that speculated on prefetched stale values *will* eventually abort, thereby resulting in an overall slow down of the program execution. Therefore, it is imperative that depending on the speed of computation, the number of update threads – both prefetch and commit – should be dynamically balanced as needed to speedup commits.

*Notice that neither the number of prefetch threads nor the number of commit threads can be statically determined in advance: the number of most effective prefetch threads depends on the number of compute threads, the speed of the computation, dynamic network latencies etc. All these point to the need for dynamically adapting these types of requests.*

### B. Asynchronous Parallelism

In this form of parallelism there is little to no synchronization between various concurrently executing threads [7], [8], [9]. Consider the problem of Distributed Spam Filtering where the sharing is total. More specifically, the set of words that indicate presence of spam are fully shared by all computation threads of the Spam Filter program. While this word set may be infrequently updated, the completed spam classification computations need not be thrown away. It suffices to execute future spam classifications using the updated word set. In other words, the result of the current computation is deemed to be 'good enough'. PageRank is another example of an application that falls into this category. The PageRank algorithm computes the rank of a node based on the ranks of its neighbors, until the ranks only change negligibly, that is, until the ranks of nodes converge under some criterion. Now, if a computation thread reads the rank of a neighbor that then becomes stale due to a concurrent update, the results of the computation need not be thrown away. By making this choice, the path to convergence is modified, but the progress towards convergence is not hindered. It suffices to ensure that updates are not lost.

As described in the two examples above, for applications that exhibit asynchronous parallelism, there is no predetermined order of execution that needs to be enforced among concurrently executing threads. Moreover, if data sharing between concurrent threads exists, computations that use stale values can be allowed to commit. That is, in contrast to speculative parallelism, commit failures do not arise in asynchronous parallelism. This relaxation is a key differentiator of asynchronous parallelism from speculative parallelism. As we shall see, this differentiation allows for novel possibilities to achieve speedups. Other examples of applications that belong in this category are: 1D and 2D Heat Simulation and Wave Simulation.

We now present a study of the asynchronous benchmarks listed in Figure I. The baseline for the speedups is the configuration with zero prefetch and zero update threads. We vary the number of commit and update threads. We now present the observations for varying communication.

**Communication Strategy.** Prefetching can help throughout the lifetime of such applications depending on the rate at which objects get replaced from the machine-level object store. As seen in Figure 6, prefetching alone increases the speedup by 37% for PR and PR-HS, 22% for SF, 35% for 2DHS, and 22% for both HS and WS. Also observe that increasing the number of prefetch threads only helps upto a point. After that, there is no benefit from prefetching. Notice once again, that the number of update and prefetch threads cannot be known in advance. This also points to the need for dynamically adapting the number of prefetch and update threads at runtime.

In the next section, we present an adaptive computational model that directly follows from the above observations. We propose, present and evaluate various parameters that can be monitored at runtime. Finally, we evaluate our proposed dynamic model and show that it dynamically achieves speedups close to that of the best hand-optimized parallel versions.

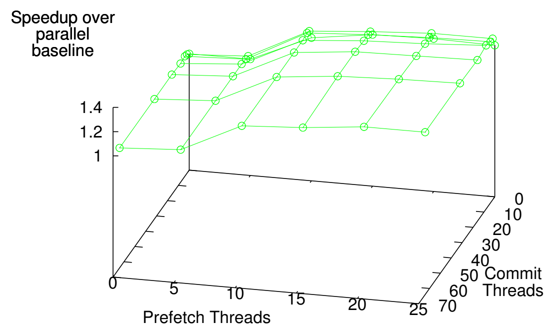
## III. $ABC^2$: AN ADAPTIVE RUNTIME FRAMEWORK

The results of the study of applications in the previous section are summarized in Table II. As we can see, not all applications benefit from prefetching but for those that do, the appropriate number of prefetch threads can vary. Finally, although update threads help tolerate latency in all applications, the appropriate number of update threads varies across applications. Therefore we conclude that to benefit from the latency tolerance mechanisms we support, it is best to develop a runtime model that supports all of the proposed mechanisms and, with the help of runtime monitoring, it adapts their use to meet the needs of the application.

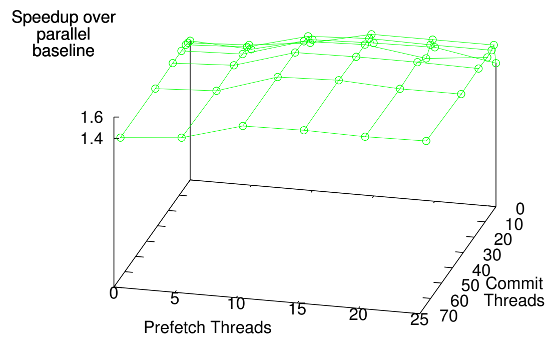
TABLE II: Summary of communication strategies.

| Strategy | Parallelism | |
|---|---|---|
| | Speculative | Asynchronous |
| Number of Prefetch Threads | Varying | Varying |
| Number of Update Threads | Varying | Varying |

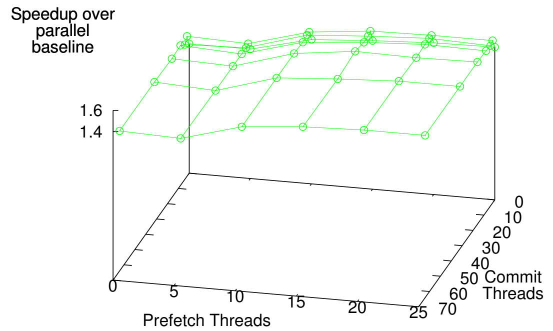
In this section we develop an adaptive framework that performs well for all different types of parallel applications considered without any a priori knowledge of their type or behavior. We identify parameters that are monitored at runtime to guide and control the degree of prefetching and aggressiveness of updates. In the remainder of this section we present the system architecture and design, discuss the parameters monitored at runtime, and finally present $ABC^2$which is the runtime decision making algorithm.



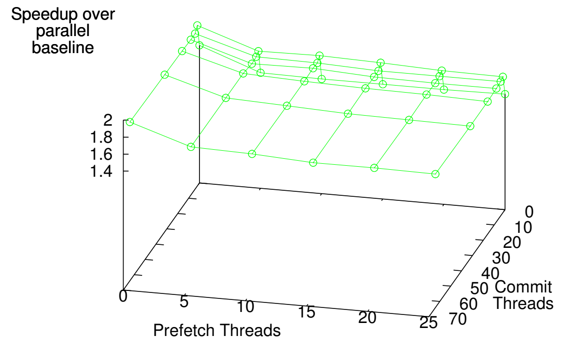
(a) PR: Speedup



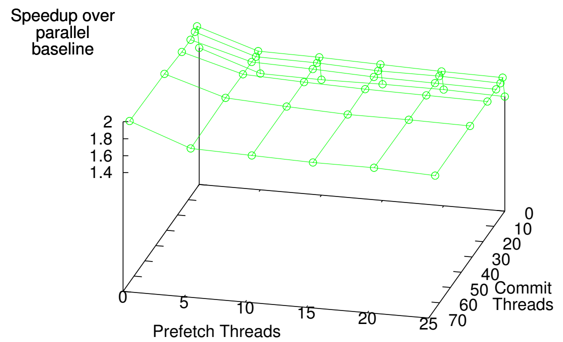
(b) PR-HS: Speedup



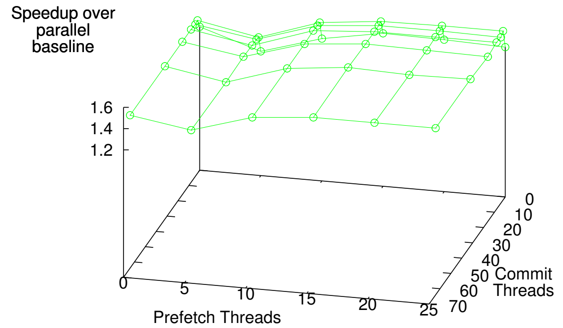
(c) SF: Speedup



(d) WS: Speedup



(e) HS: Speedup



(f) 2DHS: Speedup

Fig. 6: Speedups for PR, PR with high sharing are shown in figures 6a and 6b; speedups for SF and WS are shown in figures 6c & 6d respectively; and speedups for HS and 2DHS are shown in 6e, 6f. Prefetch threads make remote objects locally available, hence avoiding remote accesses and reducing the average fetch time. Each benchmark was executed on a cluster of 6 machines, running 5 computation threads.

### *A. System Design*

In this section we present the system design and describe the roles and responsibilities of its various components.

***DSM.*** We use the object-based dyDSM [10] as the underlying DSM in this work.

***Speculation.*** As described in Section II-A, we have implemented the copy-in, copy-out speculative model from SpiceC [4]. The separate compute and commit phases of this model are in tune with our need to separate computation and communication.

***Thread Pool.*** To eliminate the penalty of creating and terminating new threads at runtime, we employ a *thread pool* model. A large number of threads of each required type are created during the runtime initialization. Unused threads are put to sleep and woken up as needed rather than resorting to polling for work. This prevents idle threads from using CPU resources.

***Prefetching.*** Prefetching is implemented to take advantage of the graph structures being used by the applications. The computation threads enque the objects IDs of the objects that need to be prefetched into the *prefetch queue*. The prefetch threads first deque the object IDs from the prefetch queue, then fetch the object asynchronously and place the object into a *prefetch buffer*. When per-machine replication is used, the per-machine replica store can be used as the prefetch buffer.

When no replication is used, a separate prefetch buffer needs to be used.

***Updates.*** Update threads first dequeue the thread-private memory object from the commit queue and asynchronously *commit* the data back to the DSM. With speculation, the update thread must *atomically* perform the mis-speculation check (discussed in Section II-A) that involves detecting write-read, read-write, and write-write dependencies for all the data in the thread-private memory object and **then** perform the write back to the DSM. Therefore, when speculation is used, the update threads work in a *commit* mode. Finally, for programs with asynchronous parallelism, the update must still perform a commit, but as discussed in Section II-B, it suffices to atomically perform only the write-write dependency check before the update. Note that since the mis-speculation check and update into the DSM need to be performed atomically, batch updates into the DSM can be performed efficiently.

***Computation.*** Computations are performed by compute threads. When speculation is used, the compute threads implement the copy-in, copy-out model to allow concurrent compute threads to execute in isolation. Under this model, the compute threads copy all the data used by the computation into a thread-local private memory. Once the computation is complete, the entire private memory (which also contains the results of the computation) are pushed into a commit queue, to be handled asynchronously by the update threads. Similar mechanism is also used for applications with asynchronous parallelism. Figure 7 summarizes the overview of the prototype. The core components of the framework are:

1) Object based DSM;
2) Speculation via SpiceC's copy-in, copy-out model;
3) Separate computation and commit phases; and
4) Prefetch, Compute, and Update thread pools.

The numbering in Figure 7 indicates the flow of execution. The compute thread first populates the prefetch queue with the IDs of objects to be prefetched and returns to its computation task. The prefetch threads asynchronously fetch the necessary data from the DSM into the prefetch-buffer. When the compute thread needs a data item, it first looks up the prefetch buffer and brings it into the per-machine replication stores as appropriate; the compute thread then continues with its task. Once completed, the compute thread pushes the results of the computation into the update queue and starts working on the next computation. The update threads dequeue the data from the update queue and commit or discard the results of the computation depending on the success or failure of the mis-speculation check.

## *B. The* ABC$^2$*Algorithm*

For each of the decisions the runtime needs to make, we consider the various parameters that we evaluated in Section II. To vary the number of prefetch threads, we propose to monitor the length of the prefetch queue, which contains the IDs of objects that need to be prefetched. Finally, to vary the number of update threads, we monitor the length of update queue, which contains the results of computations that need to be written back to the DSM.

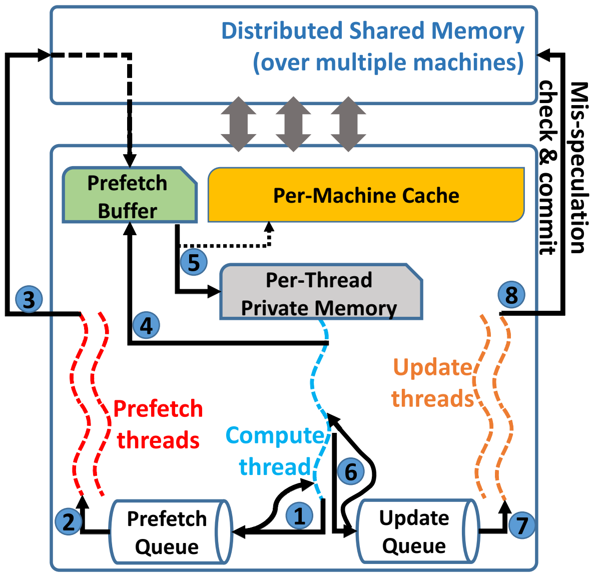


Fig. 7: The $ABC^2$system design showing the DSM, Prefetch, Compute & Update threads.

```
Adapt Prefetching:
 // At the start of the prefetch task:
 0. if(prefetch_q.length > PQ_Threshold):
 1.    wakeup_more_prefetchers()
 2. else:
 3.    sleep_extra_prefetchers()

Adapt Updates:
 // At the start of update task:
 0. if(update_q.length > UQ_Threshold):
 1.   wakeup_more_udapters()
 2. else:
 3.   sleep_extra_updaters()
```

Fig. 8: The $ABC^2$Algorithm.

The adaptive $ABC^2$algorithm monitors the various parameters listed above to adapt the computation model and communication resources. There are two independent parts to the algorithm, both of which are initiated simultaneously at start of the application. The two independent parts adaptively control the prefetch threads, and update threads. The listing in Figure 8 presents the adaptive algorithm. To adapt prefetching and updates, more threads are used depending upon the current queue sizes. In these experiments, the maximum number of prefetch threads was experimentally bounded at 25 and the number of update threads was bounded at 64.

Each of the parameters like prefetch and update queue lengths are monitored as needed in Figure 8. We now briefly discuss the benefits and generality of monitoring these parameters.

1) *Prefetch-queue length*: To turn on or turn off prefetching on demand, we simply leverage the presence of the prefetch queue. We will always have at least one

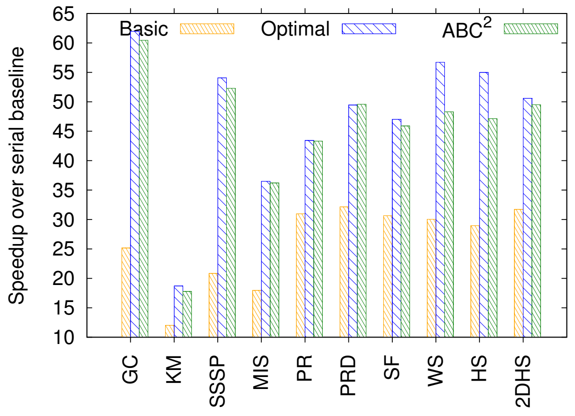


Fig. 9: Speedups without and with *ABC*$^2$.

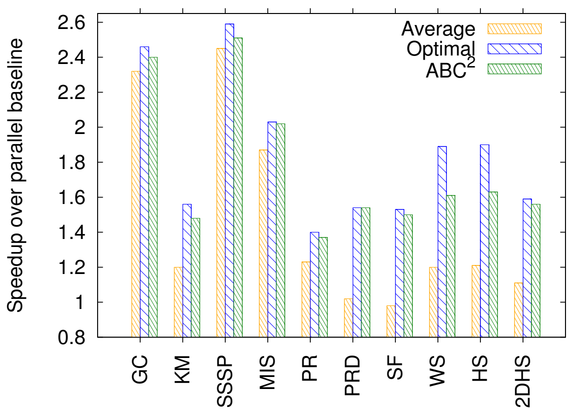


Fig. 10: Performance evaluation of *ABC*$^2$.

prefetcher active. If this prefetcher observes many pending data objects to be prefetched, by querying the prefetch queue length, it will wake up more prefetchers. If prefetchers find no work to do, they simply go to sleep until woken up again.

2) *Update-queue length*: Update threads are dynamically adjusted in a manner similar to that used for prefetch threads, with the exception that the update threads monitor the length of the update queue.

### C. *Evaluation of* ABC$^2$

We now evaluate the *ABC*$^2$framework on a cluster of 6 8-core Dell T410 machines each with 8 GB memory, running Ubuntu 10.04, Kernel v2.6.32-21. We use the *dyDSM* [10] as the underlying DSM. Our goal is to compare the speedups achieved by the adaptive framework with those of the fastest configurations from the study in Sections II-A and II-B. The speculative benchmarks are run with 4 compute threads while the asynchronous benchmarks are run with 5 compute threads, while adaptively varying prefetch and update threads. In this evaluation, we consider the following different configurations:

- `Basic`, which is the basic configuration *without* prefetch or update threads: without *ABC*$^2$, prefetching and updates are handled by the computation thread itself.
- `Optimal`, which is the fastest configuration from the motivation in Section II-A and Section II-B.
- `Average`, where the maximum number of prefetch and update threads is set to the respective average numbers as measured in the *ABC*$^2$version.

We show that for each benchmark, our adaptive framework (a) automatically selects the optimal fastest configuration seen in the study above; and (b) achieves speedups comparable to the fastest configuration.

***Speedup over serial baseline.*** We first show that parallel benchmarks running on 6 machines with tt *ABC*$^2$ achieve significant speedups over the `Basic` version without *ABC*$^2$. The speedups in this experiment are baselined to the serial versions of their benchmarks. Both the serial and parallel versions fetch data from the DSM and update data back into the DSM. In the serial versions of the benchmarks, the network communication is handled by the computation thread itself. Figure 9 compares the speedups of the `Basic`, `Optimal` and *ABC*$^2$. First, we see that without *ABC*$^2$, the `Basic` versions of speculative benchmarks, on an average, achieve speedups of 18x while the asynchronous benchmarks achieve 30x speedup. Next, we see that the best achievable speedups with prefetch and update threads from the study in Section II is significantly higher: an average of 40x for speculative benchmarks and 47x for asynchronous benchmarks. On an average, this is an improvement over the `Basic` version by 22x for speculative benchmarks and 17x for asynchronous benchmarks. This clearly shows that *ABC*$^2$algorithm is beneficial for performance. This apparently very large increase is because these benchmarks are mostly network-bound due to the DSM communication. Therefore, providing dedicated resources for network IO in the form of prefetch and update threads hides the network communication latency while simultaneously allowing the computations to make progress with overall speedup.

***Speedup over parallel baseline.*** We now evaluate the speedup benefits of *ABC*$^2$ for the *parallel* versions of the benchmarks. The baseline in this experiment is the parallel benchmark running on 6 machines without dedicated prefetching or update threads; the fetches and updates are handled by the computation thread itself. This baseline allows us to evaluate the specific benefit of the *ABC*$^2$ algorithm dynamically adapting the prefetch and update threads for parallel programs. Therefore, the speedups presented here are over and above the speedups achieved by the parallel program without adaptive communication. Figure 10 compares the speedups of the `Optimal` configuration from Section II with the speedups achieved by the *ABC*$^2$ algorithm. We see that the *ABC*$^2$algorithm achieves speedups between 1-6% of that achieved in the study, except in the case of WS and HS, where *ABC*$^2$achieves speedups within 15% of that in the study. As seen in Figures 6d and 6e, WS and HS have optimal running times with zero prefetchers, while the adaptive algorithm uses about 3 prefetchers for both WS and HS, as seen in Figure 11. The prefetching overhead results in lower speedups in these two cases.

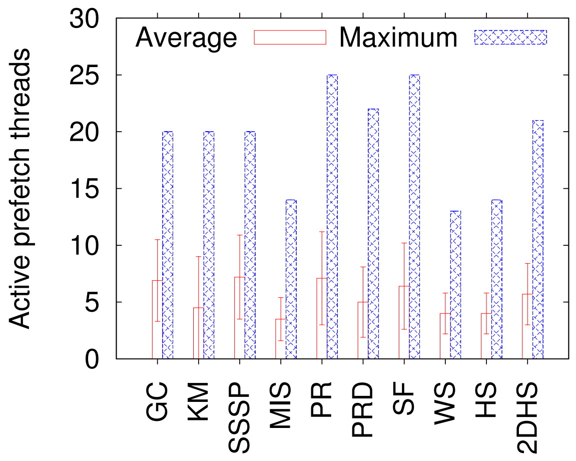


Fig. 11: Concurrently active *prefetch* threads.

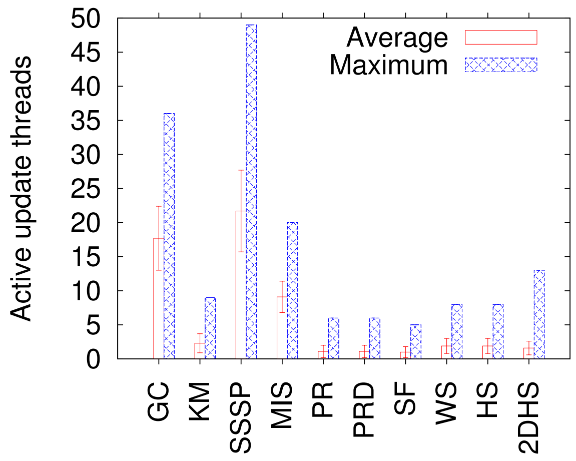


Fig. 12: Concurrently active *update* threads.

***Adaptive Prefetching and Updates.*** Finally, we evaluate the specific improvements accrued by `ABC`$^2$ adaptively varying prefetch and update threads. Comparing the speedups achieved using the average numbers for prefetch and update threads obtained from Figures 11 and 12 with the speedups obtained by the *ABC*$^2$, we see that *ABC*$^2$always gives better speedups close to the optimal in the study. This indicates that a higher maximum number for prefetch and update threads employed by *ABC*$^2$is important and useful in achieving better speedups.

Figures 11 and 12 show the average and maximum number of concurrently active prefetch and update threads as measured by the *ABC*$^2$algorithm. We see that the *ABC*$^2$algorithm uses, on an average, 5 prefetch threads. On an average, all the asynchronous benchmarks use just 2 update threads while the speculative benchmarks use 16 update threads, except for KM, which uses only 2 update threads. The update phase of asynchronous benchmarks is significantly shorter compared to the speculative benchmarks: for GC, SSSP and MIS, since a node's value is calculated based on its neighbors, the mis-speculation check involves the node and all its neighbors. Therefore, speculative updates are network-bound and time consuming, but not CPU-bound. Hence, with fewer update threads, more commit requests can get queued to the update queue. But as can be seen from the `Adapt Updates` section in Figure 8, when the update queue length is greater than a threshold, the *ABC*$^2$algorithm wakes up more update threads. This does not happen often with asynchronous benchmarks since there is no mis-speculation check in their updates. With KM, the mis-speculation check only involves the single cluster data; there is no notion of neighbors in this case. Therefore, the mis-speculation check is fast and hence fewer number of update threads are sufficient to keep the update queue length below the set threshold. Also, the maximum numbers show that occasionally the *ABC*$^2$algorithm uses a maximum of 25 prefetch threads and 49 update threads. The vertical bars show the standard deviation around the average number of prefetch and update threads. The average, standard deviation and the maximum are indicative of the dynamically varying number of prefetch and update threads in the *ABC*$^2$algorithm. The standard errors for the average number of prefetch and update threads are 0.0036 and 0.0054, indicating a very high confidence in the average numbers shown.

## IV. Related Work

Distributed Shared Memory or DSM is a prominent choice of memory models for programming distributed systems. In essence, this software layer superimposes an abstraction of shared memory over the physical memories from multiple machines in a cluster. This allows programmers to write applications using the familiar abstraction of shared memory systems. Additionally, the scalability of clusters coupled with the shared memory abstraction makes DSMs an attractive option for scalable distributed computing without resorting to specialized programming models or frameworks.

Dynamic data prefetching in the context of distributed systems has also been studied [11], [12] in various research and engineering communities. Our work differs by providing a runtime that monitors dynamic parameters and dynamically turns on/off prefetching as needed. Further, our designation of separate threads on each machine specifically for prefetching allows us to dynamically scale prefetching on demand. This further allows for optimal allocation of physical CPU cores to computation and communication needs of the application. There are other strategies based on Markov models [13] etc., that do not need any input from the user, but those models are not the primary focus of this evaluation.

DSMs are not the only programming/memory model for distributed parallel programming. MapReduce [14] is a popular programming model that consists of two distinct phases of computation: map and reduce. The map operation distributes work across the cluster while the reduce operation aggregates the results from the across cluster. However, MapReduce cannot be applied to every program that can be parallelized, thereby limiting it to a smaller set of applications than possible with DSM. Another approach is to use distributed memory (DM) (in contrast to distributed *shared* memory (DSM)) model. With DM, the programmer has to explicitly manage and move data between various compute nodes. HipG [15] is an example of a graph processing framework that uses DM. HipG contains DM extensions similar to those proposed in SpiceC [4] and works in two phases, like MapReduce. The Message Passing Interface (MPI) [16] is a popular DM programming model. Compared to DSM based systems, the programmer burden in programming for these systems is very high. For example, with the exception of transferring atomic data types and their arrays, simply creating the wrappers to allow serialized communication of incrementally complex data structures can be daunting enough to deter the use of this system for larger programs.

Traditional clusters were built from commodity single core CPUs and the DSMs proposed for those systems were not designed to exploit the presence of multiple cores. For instance, systems like ORCA, Shasta, TreadMarks and Emerald [1], [2], [17], [3] were all successful DSM systems for clusters of uniprocessor machines. This work is focused not on the DSMs *perse*, but rather on novel approaches to exploit new opportunities afforded by multi-core machines. Specifically, we explore the dynamic balance between communication and computation for speculative and asynchronous applications on DSMs. Prior systems do not provide explicit support for speculative or asynchronous parallelism, like we do. Finally, to the best of our knowledge, none of the older systems monitored dynamic, runtime characteristics of data and applications to automatically *adapt communication and computation* for optimal performance. To tolerate latencies, prior work has explored lazy vs. eager release consistency models [17]. Other work has looked at dynamically adapting between single and multiple writer protocols [18], [19] or adapting to dynamic sharing patterns, which again relies on some release consistency model [18] in the context of regular applications. This work focusses on irregular applications; for speculative applications, we rely on the SpiceC [4] model, which is a lazy release, multiple-writer protocol. Asynchronous applications by definition require no strict consistency models; we use the multiple writer model for both speculative and asynchronous applications, with primary focus on $ABC^2$: Adaptively Balacing Communication and Computation.

## V. Conclusion

In this paper, we motivate the need to delicately balance computation and communication for applications with speculative and asynchronous parallelism. To address this problem, we propose to enable fine-tuned balance between computation and communication: we propose the separation of concerns into prefetch, compute and update threads. To dynamically adapt the system based on the runtime application and data characteristics, we proposed a scheme to monitor data sharing, hit-rates and fetch-times. An evaluation of the $ABC^2$model shows that the adaptive scheme achieves performance close to that of the optimal case.

## VI. Acknowledgements

This research was supported in part by a Google Research Award and NSF grants CCF-1318103, CCF-0963996 and CCF-0905509 to the University of California, Riverside.

## References

[1] H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum, "Orca: A language for parallel programming of distributed systems," *IEEE Transactions on Software Engineering*, vol. 18, pp. 190–205, 1992.
[2] E. Jul, H. Levy, N. Hutchinson, and A. Black, "An object-oriented language and system that indirectly supports dsm through object mobility," *University of Washington Technical Report*, 1988.
[3] D. J. Scales, K. Gharachorloo, and C. A. Thekkath, "Shasta: a low overhead, software-only approach for supporting fine-grain shared memory," in *Architectural Support for Programming Languages and Operation Systems (ASPLOS)*, 1996, pp. 174–185.
[4] M. Feng, R. Gupta, and Y. Hu, "SpiceC: scalable parallelism via implicit copying and explicit commit," in *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2011, pp. 69–80.
[5] J. Oplinger, D. Heine, S.-W. Liao, B. A. Nayfeh, M. S. Lam, and K. Olukotun, *Software and hardware for exploiting speculative parallelism with a multiprocessor*. Computer Systems Laboratory, Stanford University, 1997.
[6] L. Rauchwerger and D. A. Padua, "The lrpd test: Speculative run-time parallelization of loops with privatization and reduction parallelization," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 10, no. 2, pp. 160–180, 1999.
[7] S. H. Roosta, *Parallel Processing and Parallel Algorithms: Theory and Computation*, 2000.
[8] G. L. Steele Jr, "Making asynchronous parallelism safe for the world," in *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 1989, pp. 218–231.
[9] L. Liu and Z. Li, "Improving parallelism and locality with asynchronous algorithms," in *ACM Sigplan Notices*, vol. 45, no. 5. ACM, 2010, pp. 213–222.
[10] S. C. Koduru, M. Feng, and R. Gupta, "Programming large dynamic data structures on a dsm cluster of multicores," in *7th International Conference on PGAS Programming Models*, 2013, p. 126.
[11] A. Dash and B. Demsky, "Automatically generating symbolic prefetches for distributed transactional memories," *Middleware 2010*, pp. 355–375, 2010.
[12] H. Liu and W. Hu, "A comparison of two strategies of dynamic data prefetching in software dsm," in *Parallel and Distributed Processing Symposium., Proceedings 15th International*. IEEE, 2001, pp. 6–pp.
[13] D. Joseph and D. Grunwald, "Prefetching using markov predictors," in *ACM SIGARCH Computer Architecture News*, vol. 25. ACM, 1997, pp. 252–263.
[14] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.
[15] E. Krepska, T. Kielmann, W. Fokkink, and H. Bal, "A high-level framework for distributed processing of large-scale graphs," in *Distributed Computing and Networking*. Springer, 2011, pp. 155–166.
[16] T. M. Forum, "Mpi: Message passing interface," 1993. [Online]. Available: http://www.mcs.anl.gov/research/projects/mpi/index.htm
[17] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel, "Treadmarks: Shared memory computing on networks of workstations," *IEEE Computer*, vol. 29, no. 2, pp. 18–28, 1996.
[18] L. R. Monnerat and R. Bianchini, "Efficiently adapting to sharing patterns in software dsms," in *High-Performance Computer Architecture, 1998. Proceedings., 1998 Fourth International Symposium on*. IEEE, 1998, pp. 289–299.
[19] C. Amza, A. L. Cox, S. Dwarkadas, L.-J. Jin, K. Rajamani, and W. Zwaenepoel, "Adaptive protocols for software distributed shared memory," *Proceedings of the IEEE*, vol. 87, no. 3, pp. 467–475, 1999.