Scalable Software Latency Hiding Schemes: Evaluation of the Poststore and Prefetch Options*

Chaitanya Tumuluri and Alok N. Choudhary

121 Link Hall, ECE Department, Syracuse University, Syracuse, NY-13244
Email: *(tumuluri, choudhar)@cat.syr.edu*

Abstract. Most latency hiding studies for Distributed Shared Memory (DSM) systems use *prefetching*, while few explore use of *poststoring*. This study develops, and compares performance gains obtained using, a run-time poststoring scheme (PST) and an application-specific prefetching scheme (PFH). The PST and PFH schemes produced scalable reductions in loop execution times.

1 Introduction

Most Distributed Shared Memory (DSM) systems [1] implement an invalidation based coherence protocol, which frequently trigger expensive remote accesses. Hence, latency hiding features such as **prefetching** and **poststoring**[2] assume critical importance. Many studies have used either hardware [3] or software [4] *prefetching* schemes largely for array-based regular applications with *static* locality. They mostly used *simulations* of the memory system driven by memory access events generated during application execution. Also, their evaluations are restricted to at most 32 processors and do not address scalability issues. We present two latency hiding schemes, PST and PFH, and evaluate their performance on a *dynamic* application (Barnes-Hut algorithm [5]), which uses pointer-based lists of aggregate data structures. As opposed to execution-driven simulations, we directly study the execution performance on the target DSM machine (viz. KSR1). The schemes produced significant (21%) and scalable improvement in the overall loop execution time of the application.

2 Background

KSR1 Architecture: The KSR1 [6] interconnect is a two-level ring hierarchy. Each first level ring connects 32 processor cells and provides sequentially consistent shared memory via an *invalidation* coherency protocol. An onboard split-cache (called *subcache*) and a secondary cache (called *local cache*) are available at each processing node. The data transfer unit is a 128 byte chunk termed

---

* This work was supported in part by an NSF Young Investigator Award CCR-9357840

[2] The *poststore* [2] allows users to implement a source-level *selective-update* protocol.

a *subpage*. The *Prefetch* instruction moves a specific subpage into the local cache. The *Poststore* broadcasts a subpage onto the ring and local caches having place-holders for this subpage update their entries. Often, local caches enroute between the responding and requesting caches having invalid place-holders automatically update their entries. This feature is termed *Automatic Updates.*

The Barnes Hut Algorithm: The application performs an N-body simulation of stars in interacting galaxies [5]. The main data structure is an *octree* whose internal cells represent recursively subdivided partitions of the physical space. The leaf cells of the tree represent the bodies (stars). The computational structure of the Barnes-Hut code is outlined in Figure 1. *The ForceTime constitutes 90% to 95% of the total TrackTime, and is hence the focus of our scalability analysis.* Dynamically changing body interactions defy *static analysis* warrating our runtime PST and PFH schemes.

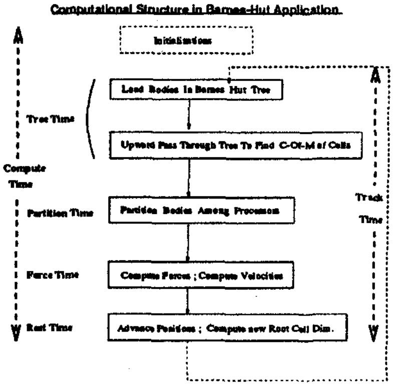
Computational Structure in Barnes-Hut Application
Initializations
Load Bodies In Barnes Hut Tree
Upward Pass Through Tree To Find C-Of-M of Cells
Partition Bodies Among Processors
Compute Forces ; Compute Velocities
Advance Positions ; Compute new Root Cell Dim.
Tree Time
Compute Time
Partition Time
Force Time
Rest Time
Track Time

Fig. 1. Computational Flow

3 Latency Hiding Schemes

We assume that data abstractions ('bodies') are aligned into subpages and subpages mapped to processors during data partitioning are said to be **owned** by those processors. An 'unowned (processor) reference' implies a reference to data within an unowned subpage[3]. Conversely, references to data within subpages owned locally are 'owned references'. Two descriptors per subpage copy, **Owner** and **ExtFlag**, track the ownership and occurrence of unowned references respectively. The **ExtFlag** of a locally owned subpage is reset upon an owned-write and, is set by the first subsequent unowned reference. Descriptor updates produce system-wide coherency invalidation of subpage copies. Processor stalls due to such invalidations are prevented by poststoring the updated subpage.

---

[3] Includes references to a dirty (unshared) local copy of a remotely-owned subpage.

PST Scheme: We assume architectural/runtime system support for monitoring the number of owned (R_o) and unowned (R_{un}) references between two consecutive invocations of the PST runtime system. Upon invocation, the PST scheme computes the measure of remote-processor affinity (A_i) of the subpage (i) as: $A_i = R_{un}/(R_{un}+R_o)$ Also, a running average, called the *Poststore Threshold* ($\bar{A}$), of selected past A_i values is used for making poststoring decisions.

After computing the A_i, if the *ExtFlag* of subpage i is set: (a) *ExtFlag* for subpage i is reset, (b) $\bar{A}$ is updated: $\bar{A} = (\bar{A} + A_i)/2$ and, (c) subpage i is poststored. Else if *ExtFlag* is unset: Subpage i is poststored only if $A_i > \bar{A}$.

PFH Scheme We implemented a history-based [3], domain-specific [4]), temporal lookahead [7] PFH scheme. PFH tries to anticipate and prefetch: (a) unowned references to leaf-cells and, (b) previously unreferenced internal cells. The PFH scheme defines *CurrDist* as the distance in physical space between successive bodies updated in the ForceTime loop. *AvgDist* is defined as a moving average of the *CurrDist* values. An explicit history of octree cells/bodies traversed by the first body is recorded. When a body's *CurrDist* exceeds the *AvgDist* value, the history is updated where this body's traversal deviates from the stored history. A deviant reference to an internal cell triggers a prefetch of its unowned octree descendants.

4 Scalability Evaluation Setup

RCTS Rule: Realistic Constant Time Scaling (RCTS) [8] of the number of processors by k, requires scaling the number of bodies n by $\sqrt{k}$, to maintain a constant execution time. Subsequently, the accuracy parameter θ and the timestep t are scaled to ensure that error contributions from all sources in the scaled simulation are proportional to those in the unscaled simulation: If n scales by a factor s, then θ scales to $\frac{\theta}{\sqrt[4]{s}}$ and t scales to $\frac{t}{\sqrt[4]{s}}$. The base case is a 4 processor run for which ($n = 4k, \theta = 0.8, t = 0.05$). The scaled problem sizes (see Table 1), are derived via the RCTS rule.

Simulation Sizes & Parameters::RCTS Simulation									
NPROC	4	24	40	56	64	80	96	112	120
NBODY n	4096	10034	12952	15325	16384	18312	20066	21674	23170
THETA θ	0.8	0.64	0.6	0.58	0.55	0.55	0.54	0.53	0.51
TIME t	0.05	0.04	0.037	0.036	0.034	0.034	0.033	0.033	0.032

Table 1. Simulation Sizes and Parameters under RCTS Scaling

Constant Problem Size Scaling Rule: The application parameters were fixed over all runs to be: ($n = 32768, \theta = 0.53, t = 0.018$).

5 Performance Analysis

The results for four different sets of runs are presented. One set of runs use the uninstrumented **Normal** version of the code. The second uses the **PST** scheme while the third is for the **PFH** runs. The last, labeled **PFHPST**, uses both the PFH and PST schemes. *The PST, PFH and PFHPST timings include the overheads of the latency hiding schemes.*

5.1 Application Behavior

This section examines the scalability of the Norm and latency hiding runs.

Constant Time Scalability: The RestTime, in Fig. 2, represents synchronization etc., while the TreeTime equals (TreeTime + PartitionTime) of Fig. 1. The ForceTime decreases from representing about 90% to approximately 20% - 30% of the TrackTime in the Norm runs. The TreeTime and RestTime together increasingly dominate the Tracktime (10% to 60% of TrackTime). The finer data partitions in larger runs reduce the ForceTime, but also increase remote accesses and lock contention thereby increasing the RestTime. The TrackTime components scaled much better in the the latency hiding versions. Fig. 3 compares the versions for an 80 processor run. The PST, PFH and PFHPST versions reduce the (RestTime + TreeTime) while maintaining a constant ForceTime. Hence, PST and PFH versions produced a 21% reduction while PFHPST produced a 10% reduction in the TrackTime over the Norm run.

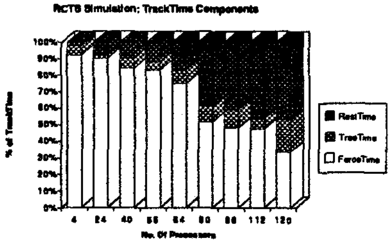

Fig. 2. TrackTime Components:Norm RCTS Simulation

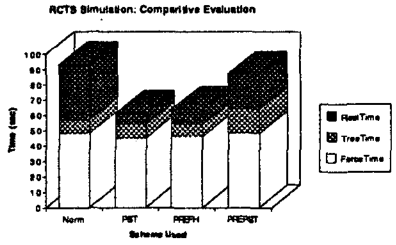

Fig. 3. Latency Hiding::80 Processor, 18k Particles RCTS Simulation

Constant Problem Size Scaling: Again, the TrackTime components for Norm runs do not scale well (Fig. 4). The ForceTime component decreases from representing almost 99% of the TrackTime to less than 50%. Fig. 5 compares the Norm and latency hiding versions for a 112 processor run. The PST, PFH and PFHPST schemes produced a 35%, 37% and 50% reduction over the 112 processor Norm run respectively. In summary, the PST and PFH schemes successfully reduce latencies arising from invalidation-misses and locality variations. They also scale very well in performance.

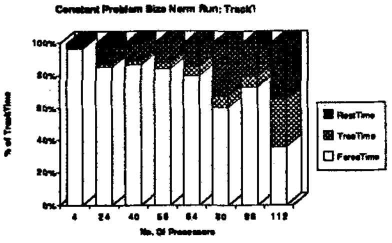

Fig. 4. Norm Run TrackTime Components: Constant Problem Size Scaling

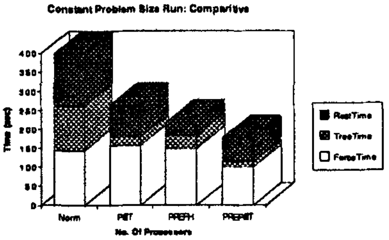

Fig. 5. Constant Problem Size TrackTime Components: Latency Hiding vs. Norm Runs

5.2 KSR1 Machine Behavior

Local Cache Performance: In this section, the plots of the subpage misses and stall times present the maximum values over the processors in each run.

Effect of Scaling Working Set: The scaling rules used, produced successively smaller partitions at each processor [8], therefore increasing remote accesses. This is seen in the increasing number of local cache misses in Fig. 7 with a noticeable peak in the 80 processor run. The half-ring peak is seen to a lesser extent, at the 112, 24 and 56 processor runs as well since their working sets contain enough locality to offset the effects of *automatic updates*.

Effect of Latency Hiding Schemes: The effects of the PST version is a smoothing effects on the half-ring peaks, as in Fig. 7. However, the increased remote accesses are not very well predicted by the PFH scheme for larger runs. Combining the PFH and the PST schemes should therefore yield better results as seen by the PFHPST run in Fig. 7. The interested reader is referred to [9] for further details.

Stall Time Analysis: The StallTime (Fig. 6) is a good indicator of the computation to communication (CC) ratio of the execution. This is because the main memory misses stall the processor while communicating data requests, to effectively lower the CC ratio. Thus, the Stall times shown in Fig. 6 correlate well with the local cache subpage miss curves. The lower local cache misses in the latency hiding runs produce lower stall times. This effect is pronounced in the larger runs where the stall time proportion of the TrackTime in the Norm runs is high (90% in 120 processor run). Thus, latency hiding assumes critical importance for increasing CC ratio via overlapping communication for satisfying remote accesses with the computation.

6 Conclusions

Both the PST and PFH schemes produced significant (20% in 80 proc. run) reductions in the TrackTime over the Norm versions. However, the PST scheme

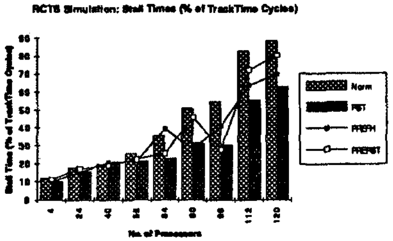

Fig. 6. Processor StallTime :: RCTS Simulation

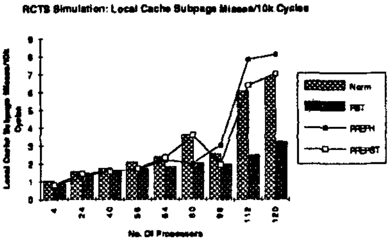

Fig. 7. Subpage Misses in Local Cache::RCTS Simulation

is application-independent and allows users to selectively implement an *update* coherency semantics. The successful use even in applications using pointer-based data structures further adds to PST's attractiveness. Hence, the main result is that *poststoring* (as opposed to prefetching) in general, and our PST scheme (as opposed to the PFH scheme) in particular, is the better path toward significant and scalable performance improvements.

References

1. J. Kuskin *et al.*, "The Stanford FLASH Multiprocessor," in *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pp. 302–313, 1994.
2. E. Rosti *et al.*, "The KSR1: Experimentation and Modelling of Poststore," in *Proceedings of the 1993 Sigmetrics Conference on Measures and Modelling of Computer Systems*, pp. 74–85, 1993.
3. J. W. C. Fu and J. Patel, "Data Prefetching in Multiprocessor Vector Cache Memories," in *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pp. 54–63, 1991.
4. T. Mowry and A. Gupta, "Tolerating Latency Through Software-Controlled Prefetching in Shared Memory Multiprocessors," *Journal of Parallel and Distributed Computing*, vol. 12, pp. 87–106, June 1991.
5. J. P. Singh, W. Weber, and A. Gupta, "SPLASH: Stanford Parallel Applications for Shared Memory," *Computer Architecture News*, vol. 20, pp. 5–44, March 1992.
6. K. S. R. Corporation, *KSR1 Principles of Operations*, 1992.
7. J. Baer and G. R. Sager, "Dynamic Improvement of Locality of Virtual Memory Systems," *IEEE Transactions on Software Engineering*, vol. SE-2, pp. 54–62, March 1976.
8. J. P. Singh, J. L. Hennessy, and A. Gupta, "Implications of Hierarchical N-body Techniques for Multiprocessor Architecture," Technical Report CSL-TR-92-506, Stanford University, 1992.
9. C. Tumuluri and A. N. Choudhary, "Exploitation of Latency Hiding on the KSR1, Case Study: The Barnes Hut Algorithm," Technical Report CTC94TR176, Cornell Theory Center, 1994.