Cooperative Run–time Systems: DSM and RPC Scheduler

György Hernádi
University of Houston
Department of Computer Science
Houston, TX, 77204–3475
e-mail: geo@cs.uh.edu

---

1. INTRODUCTION

My research concentrates on potential interactions between Distributed Shared Memory (*DSM*) systems and schedulers for parallel applications. Its main focus is on the similarities of these two systems and their ability to enhance the performance of one another. A parallel application must incur some overhead for data movement and synchronization. In a DSM system, data access time is non–uniform involving overheads of uncertain magnitude to update the clients local copies of the data. The performance of an application greatly depends on how well data locality is exploited. If the function requiring the data is invoked via a Remote Procedure Call Scheduler (RPCS) which has knowledge of the state of the DSM, then it is possible to attempt an optimization of the schedule which is not possible in non–cooperative DSM, RPCS systems. DMS and RPCS can be seen as solutions for the same problem — data dependency — from two different points of view. The RPCS works with execution of code and the DMS works with data access. To minimize the data traffic, it is easy to see that DSM state information can improve the RPC distribution process. However, the RPCS can also help the DSM system, by providing prefetch hints. Since the bandwidth of the computer networks has improved much faster than latency, in most cases obtaining data in large blocks is more efficient than requesting the same data in several smaller pieces. I am experimenting with intelligence sharing between these two dissimilar run–time systems which have traditionally been treated separately.

2. DISTRIBUTED SHARED MEMORY

In a distributed memory environment, transparent access of shared data involves a lot of messages and causes a severe loss of performance in most cases. While strict consistency is the ideal programming model, experience shows that programmers can often manage quite well with weaker models. *Release consistency* [Kourosh Gharachorloo et al., 1990] requires synchronization of shared accesses and temporary but harmless inconsistencies give a performance edge. Many DSMs like Ivy [Kai Li, 1986] and Mirage [Brett D. Fleisch, Gerald J. Popek, 1989] are page based implementations. False sharing [John K. Bennett, John B. Carter, Willy Zwaenepoel, 1990a] can decrease the performance of page based systems. Write–shared protocol can eliminate false sharing, as in Munin [John K. Bennett, John B. Carter, Willy Zwaenepoel, 1990b] [John B. Carter, John K. Bennet, Willy Zwaenepoel, 1991] and TreadMarks [Cristiana Amza, et. al, 1996]. During release it is determined what data have been changed by comparing each shared page to an unmodified copy (*twin page*). The comparison of the twin and shared address space (*diffing*) can be done faster if the system maintains access bits for data units smaller than the hardware pages. Traditional kernel (external pager) based write detection has the disadvantage of maintaining pages too large for diffing. Smaller pages require software write detection but it has acceptable overhead [Matthew J. Zekauskas, Wayne A. Sawdon, Brian N. Bershad, 1994]. The application source aliases each shared variable and the aliases check the time stamp of the accessed shared pages. The overhead is small (two assembly instructions) because each shared access is trapped in user mode and, if the page is available, the protection code is data parallel with the application. An optimizing compiler for super–scalar processors may fit those instructions in places that otherwise would be left empty.

Finding the processing element that has the requested page can also incur a lot of network traffic. I have designed a special process called the *Shared Memory Server* (SMS) for archiving and distributing shared data. The SMS keeps a most–up–to–date version of the shared address space but it also keeps an appended list of the most recent updates in diffmessage format for each page. When a client requests a page it must provide two time stamps. The first shows

the version of the page the client has, the second is a minimum time stamp that the clients current RPC requires. In many cases the client does not require the most recent version of the page it requests. If the SMS has all the updates that occurred between these two time stamps and together they are smaller than the page, then by sending only those necessary updates the message size is reduced. There are cases when the client needs no update at all. Either there was no update between the two time stamps, or all the updates were made by the same client that requested the update. The SMS must also provide the client a time stamp for each requested page. This time stamp can not be smaller than the requested minimum time stamp but can be greater depending on the status of the page in the SMS.

3. RPC SCHEDULER

Remote procedure calls were originally invented to access special resources. In Linda [Nicholas Carriero, David Gelernter, 1989] [David Gelernter, 1985] *client* processor time is used as a remote resource. The subproblems or work–units are distributed among the client processors during run–time via RPCs. An application that uses the RPC Scheduler consists of a library of C functions and the description of synchronization for these functions. I have adopted the Petri Network (language, compiler and simulator) for this purpose. In the Petri Net when a token enters a place it initiates an RPC to a client. A token can leave a place only after its RPC has been returned. The RPCS has two parts, the *synchronizer* and the *client selector*. The Petri Net places, transitions and tokens are all local RPCS data structures and parts of the tokens are sent to the clients embedded in the RPC messages. The return value of the RPC indicates which link the token will follow in the Petri Network (*decision*).

Synchronization is based on the transitions that are implemented inside the RPCS. The SMS augments each RPC return message with a list of pages that have been accessed by that RPC. The RPCS identifies the particular tokens that made a transition fire and combines their lists of accessed pages. The combined list is then forwarded to each output token of the transition as a prefetch hint.

The client selectors job is to distribute the available tokens among the clients. The two effects that it has to consider are data locality and load balancing. It must find a compromise in order to improve performance. I am going to experiment with many heuristics that can be applied for client selection.

The RPCS has a centralized performance monitor. Each returning RPC contains timing information that the RPCS collects separately for each client and place. Before the RPCS exits, it automatically generates a performance file, that has all the essential data for tuning the application.

REFERENCES

Brett D. Fleisch, Gerald J. Popek,. 1989. Mirage: A coherent distributed shared memory design. *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, 211–223.

Cristiana Amza, et. al,. 1996. TreadMarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer 29,* 2 (February), 18–28.

David Gelernter,. 1985. Generative communication in linda. *ACM Transactions on Programming Landuages and Systems 7*, 80–112.

John B. Carter, John K. Bennet, Willy Zwaenepoel,. 1991. Implementation and performance of munin. *Proc. 13th Symposium on Operating System Principles*, 152–164.

John K. Bennett, John B. Carter, Willy Zwaenepoel,. 1990a. Adaptive software cache management for distributed shared memory architectures. *Proceedings of the 17th International Symposium on Computer Architecture*, 125–134.

John K. Bennett, John B. Carter, Willy Zwaenepoel,. 1990b. Munin: Distributed shared memory based on type-specific memory coherence. *Proceedings of the Second Symposium on Principles and Practice of Parallel Programming*, 168–176.

Kai Li,. 1986. Shared virtual memory on loosely coupled multiprocessors. Ph.D. thesis, Yale University. Ph.D. Thesis.

Kourosh Gharachorloo et al.,. 1990. Memory consistency and event ordering in scalable shared–memory multiprocessors. *Proc. of the 17th Ann. Int'l Symposium on Computer Architecture, ACM*, 15–26.

Matthew J. Zekauskas, Wayne A. Sawdon, Brian N. Bershad,. 1994. Software write detection for a distributed shared memory. *USENIX Proceedings of the First Symposium on Operating Systems Design and Implementation (OSDI)*, 87–100.

Nicholas Carriero, David Gelernter,. 1989. Linda in context. *Communications of the ACM 32*, 444–458.