Remote Store Programming: Reflective Memory for Multicore*

Henry Hoffmann, David Wentzlaff, and Anant Agarwal
MIT Computer Science and Artificial Intelligence Laboratory
{hank,wentzlaf,agarwal}@csail.mit.edu

Abstract

This work presents remote store programming (RSP), an instance of the reflective memory model designed to be incrementally supportable on multicores that support loads and stores. To demonstrate the value of RSP, its performance is compared to that of both shared and distributed memory approaches using the TILEPro64 multicore processor. RSP is shown to be as much as $1.76\times$ *faster than distributed memory and over* $5\times$ *faster than shared memory.*

Introduction

There are two dominant memory models for multiprocessors: shared memory (SM) and distributed memory (DM). In the SM model, standard load and store instructions are used to communicate and the hardware is responsible for managing the transfer of data between physically separate memories. The SM model is considered easy to use because it employs the familiar load and store instructions for communication. In the DM model, the hardware provides an explicit communication primitive and it is the software's responsibility to orchestrate communication. The DM model is considered high-performance because it allows software to control the physical location of data and ensure that processors only access physically close data.

The reflective memory (RM) model combines features of both SM and DM, supporting communication through standard load and store instructions while still allowing software to control data locality [3]. The result is a programming model that is easier to use than DM but with similar or better performance.

This talk discusses remote store programming (RSP), an instance of the RM model designed to be incrementally achievable in multicores that support load and store instructions [2]. In RSP programs, processes have private address spaces by default but can give other processes write access to their local memory. Once a producer has write access to a consumer's memory, it communicates directly with the consumer using standard store instructions that target remote memory, hence the name "remote store programming."

Hardware and OS support for RSP

To implement RSP, the hardware and system software must provide the following mechanisms:

Allocation of remotely writable data. Processes must be capable of allocating data that can be written by other processes. This data should be both readable and writable by the allocating process.

Store instructions that target remote data. Processes may execute store instructions where the destination register specifies an address in remote memory. When executing such a store, the core should not allocate a cache-line, but forward the operation to the consumer where the data was allocated. The forwarding is handled in hardware and requires a message be sent to the consumer containing both the datum and the address at which it is to be stored. The consumer receives this message and handles it as it would if the store was issued locally.

Support for managing memory consistency. After a producer process writes data to remote memory, it needs to signal the availability of that memory to the consumer. To ensure correctness, the hardware must provide sequential consistency, or a memory fence operation so that the software can ensure correct execution.

Synchronization instructions may read and write remote data. RSP allows atomic synchronization operations, such as test-and-set or fetch-and-add, to both read and write remote data.

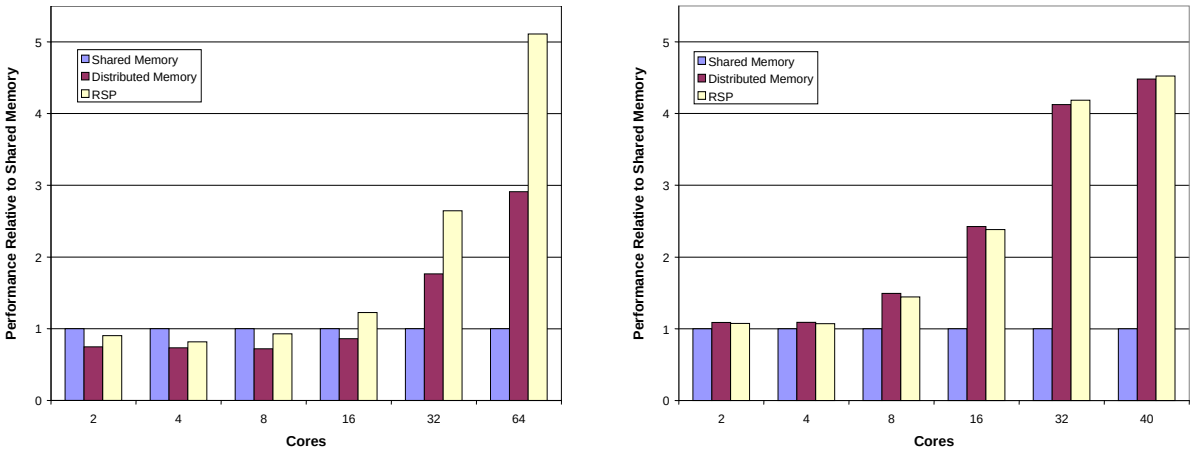
This set of features represents an incremental addition to those required on any multicore. To support loads and stores, a core must send a message to a memory controller to handle cache misses. To support RSP, this capability is augmented so that write misses to remotely allocated data are forwarded not to the memory controller, but to the core that allocated the data. The RSP implementation can use the same network that communicates with the memory controller. The additional hardware support required is logic to determine whether to send a write miss to the memory controller or to another core.

The RSP model

In an RSP program a consumer process allocates local memory and then gives write permission to a remote producer process. The RSP model has several distinctive characteristics. First, it uses load and store instructions to communicate. Second, it provides no support for bulk transfers, an omission designed to encourage programmers to store data to remote memory as soon as it is produced. Third, the model does not support remote loads or reads ensuring that load instructions always target local, physically close memory and guaranteeing minimum load latency.

Given these characteristics, communication in the RSP model is fine-grained and one-sided where data is pushed from the producer to the consumer. Data is sent from the registers of a producer to the cache of a consumer without the overhead of buffering or copying. Such communication does not require explicit scheduling for efficiency and it is easily overlapped with computation. By allowing remote stores but not remote loads, the model trades long store latency for minimal load latency. Load latency is minimized for two reasons: 1) loads are more common than stores and

*This work is supported by Quanta Computer and Tilera Corporation.



(a) **2D FFT** (b) **H.264 video encode**

Figure 1: Performance comparison of shared memory, distributed memory, and RSP. (a) shows performance of the 2D FFT, while (b) shows the performance of the H.264 encoder. All performance is normalized to that of cache-coherent shared memory.

2) it is easier to tolerate store latency than load latency as store-to-use time is typically much greater than load-to-use time.

RSP performance

The performance of RSP is evaluated by emulating it using the TILEPro64 processor [4]. This implementation demonstrates that the RSP paradigm can support high-performance embedded multicore applications like digital signal processing, video encoding and wireless communication. An RSP implementation of a 2D FFT achieves a speedup of $69.7\times$ using 64 cores, an H.264 encoder achieves a speedup of $24.7\times$ using 40 cores, while the RSP implementation of the BDTI Communication Benchmark (OFDM) achieves a speedup of $54.1\times$ using 56 cores [1][1]. In all cases, the speedups are calculated by comparing the parallel performance to that of hand-optimized serial code. RSP obtains these speedups because its fine-grained communication can be completely overlapped with computation.

Additionally, the TILEPro64 supports a variety of mechanisms that allow comparison of remote store programming to both shared and distributed memory applications. The SM implementation exercises the TILEPro64's directory-based cache-coherence protocol, while the DM implementation makes use of the two-dimensional DMA engine. The TILEPro64 allows emulation of RSP by supporting the allocation of*homed* memory. Such memory is shared supports read/write access on the core which allocates it; however, if a remote core access this memory, no cache-line is allocated. Therefore, homed memory can be used to emulate remotely writable memory if a producer does not attempt to read from it.

The performance of each of the three models is compared for a 256×256 2D FFT and an H.264 encoder for high-definition video. The relative performance of the models is illustrated in Figure . Results show that RSP can achieve over $5\times$ the performance of SM using sixty-four cores. This speedup relative to shared memory is due to RSP's emphasis on locality-of-reference, as RSP programs always access physically close memory and minimize load latencies. RSP programs also generate far less coherence traffic than SM programs. In addition, RSP performance is comparable to that of DM in the worst case and as much as $1.76\times$ better in the best case. The best case performance advantage is due to the fact that RSP requires less buffering and copying than the DMA-based approach.

Significantly, RSP achieves this performance despite requiring less hardware support than either the cache-coherent shared memory or DMA approaches. Furthermore, RSP programs are easier to write than DMA-based programs because RSP communication does not have to be explicitly scheduled like DMA transfers.

[1]Using RSP, the TILE64 processor achieved the highest compute performance of any programmable processor on this benchmark.

References

[1] BDTI communications benchmark (OFDM). http://www.bdti.com/products/services_comm_benchmark.html/.

[2] H. Hoffmann, D. Wentzlaff, and A. Agarwal. Remote Store Programming: Mechanisms and Performance. Technical Memo MIT-CSAIL-TR-2009-017, MIT, May 2009.

[3] M. Jovanovic and V. Milutinovic. An overview of reflective memory systems. *IEEE Concurrency*, 7(2):56–64, 1999.

[4] TILEPro64 processor product brief. http://www.tilera.com/pdf/ProductBrief_TILEPro64_Web_v2.pdf.