Remote Store Programming
A Memory Model for Embedded Multicore

Henry Hoffmann, David Wentzlaff, and Anant Agarwal

Tilera Corporation
`hank@alum.mit.edu`, `wentzlaf@tilera.com`, `agarwal@tilera.com`

Abstract. This paper presents remote store programming (RSP), a programming paradigm which combines usability and efficiency through the exploitation of a simple hardware mechanism, the remote store, which can easily be added to existing multicores. The RSP model and its hardware implementation trade a relatively high store latency for a low load latency because loads are more common than stores, and it is easier to tolerate store latency than load latency. This paper demonstrates the performance advantages of remote store programming by comparing it to cache-coherent shared memory (CCSM) for several important embedded benchmarks using the TILEPro64 processor. RSP is shown to be faster than CCSM for all eight benchmarks using 64 cores. For five of the eight benchmarks, RSP is shown to be more than 1.5× faster than CCSM. For a 2D FFT implemented on 64 cores, RSP is over 3× faster than CCSM. RSP's features, performance, and hardware simplicity make it well suited to the embedded processing domain.

1 Introduction

Due to the scaling limitations of uniprocessors, multicore architectures, which aggregate multiple processing cores onto a single chip, have become ubiquitous in many disciplines of computing. One of the key design features of a multicore architecture is its programming model, which must handle inter-core communication. Cache-coherent shared memory is a popular programming model that is supported by several commercial multicores including those from Intel, Cavium, RMI, and Tilera.

In the cache-coherent shared memory (CCSM) model processes communicate by reading and writing a globally accessible address space. This model is popular as it is generally considered easy-to-use, and the ease of use derives from the fact that communication in the CCSM model is accomplished using familiar load and store instructions. In addition, CCSM communication is one-sided and fine-grain, which is easy to schedule and overlap with computation. However, reliance on the abstraction of global, uniformly accessible shared memory makes it difficult for programmers to determine when their code will result in communication, and how much that communication will cost.

The CCSM model also makes it difficult to exploit locality for performance in regularly structured applications, like those typically found in video, image and signal processing. Locality can be especially important for performance on CCSM architectures which distribute the shared cache using directory protocols [1]. Locality will become more important as more cores are integrated onto a single chip because both the probability and penalty of non-local access increases with increasing numbers of cores.

This paper presents the remote store programming (RSP) model, which combines some of the features that make CCSM easy to use while still allowing programmers to control locality in software for performance. In addition, RSP requires only a small set of hardware features and is incrementally supportable in multicore architectures that support standard load and store instructions. Significantly, RSP requires less hardware support than the CCSM model but can achieve higher performance executing regular computations on multicores with a large number of cores. The RSP model can complement cache-coherent architectures by providing an alternative for performance-critical code where locality is an issue. Alternatively, RSP can be implemented as the only programming model on an architecture which may be attractive for multicore architectures targeting regular application domains.

In the RSP model processes have private address spaces by default, but they 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 the standard store instruction to target remote memory, hence the name "remote store programming." Communication in the RSP model is one-sided and fine-grain making it easy to schedule. In addition, consumer processes are guaranteed to read physically close, or local, memory.

The performance of the RSP model is evaluated by emulating it using the TILEPro64 processor. This study demonstrates that the RSP paradigm can achieve efficient parallel implementations on important multicore applications like video, image, and digital signal processing. An RSP implementation of an H.264 encoder for HD video achieves a speedup of 30.5x using 40 processes, while a 2D FFT achieves a speedup of 60x using 64 processes. Additionally, the TILEPro64 allows comparison of RSP to CCSM. While CCSM is generally faster or equivalent to RSP using a small number of cores, RSP achieves anywhere from $1.25\times$ to over $3\times$ the performance of CCSM using 64 cores. The 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. Furthermore, RSP achieves this performance with less hardware support.

The RSP model is similar to some existing programming models like the partitioned global address space (PGAS) model [2], Digital Equipment Corporation's memory channels (MC) model [3], and virtual memory mapped communication (VMMC) as implemented on the SHRIMP processor [4]. All these models combine features of CCSM while allowing users to manage locality in software. The RSP model differs in that it is designed specifically for multicore by including only mechanisms that can be incrementally supported in existing multicore architectures.

This paper makes the following contributions:

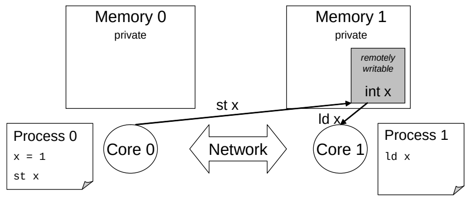


Fig. 1. Illustration of the remote store programming model. There are two cores, each of which executes a process. Process 1 allocates a remotely-writable region of memory to hold the integer `x`. Process 0 writes a new value into `x`, and this new data travels from Process 0's registers to Process 1's cache.

– It identifies the RSP model which is supportable with a small set of hardware features that can be incrementally added to a multicore architecture supporting loads and stores. This model includes features based on existing distributed shared memory models and is particularly suited to embedded processing.
– It describes the high-level features required to support RSP and argues that this model needs less support than CCSM.
– It presents a detailed performance comparison of the RSP and CCSM models using eight embedded benchmarks and finds that RSP out-performs CCSM using large numbers of cores.

The remainder of this paper is organized as follows. Section 2 describes the RSP model. Section 3 discusses the hardware and system software support required to implement the model. Section 4 discusses the methodology used to evaluate RSP and compare it to CCSM using the TILEPro64 multicore processor. Section 5 presents the comparison of RSP and CCSM performance. Related work is discussed in Section 6. Finally, the paper concludes in Section 7.

2 The Remote Store Programming Model

This section discusses programming using the RSP model. The term *process* refers to the basic unit of program execution. This work assumes that there is a one-to-one mapping between processes and processor cores, but that restriction is easily relaxed. A parallel programming paradigm is distinguished by three features: *process model*, *communication* mechanism, and *synchronization* mechanism.

The process model. RSP presents a system abstraction where each process has its own local, private memory. However, a process can explicitly give a subset of other processes write access to regions of its private memory. These regions of memory are referred to as *remotely writable*. The system abstraction for RSP is illustrated in Figure 1. The key idea of the remote store paradigm is that programmers ensure that a process always reads local memory.

The communication mechanism. In an RSP application, processes communicate by writing directly into other processes' memory using the store instruction as the communication primitive. A process that wants to consume data uses a special memory allocation function to allocate remotely writable memory. The consumer process then makes the address of this memory available to the data producer. The producer uses the standard store instruction to write to the remote memory. Once the data is stored remotely, the consumer uses standard load instructions to read the data generated by the producer; however, load instructions are not allowed to target remote memory.

The synchronization mechanism. Processes in an RSP program synchronize using atomic synchronization operations, like test-and-set or fetch-and-add. These synchronization operations are allowed to access remote memory and are the one class of operations that are allowed to read remote memory. One can easily build more advanced synchronization primitives from these operations, so high level synchronization features like mutexes, condition variables, and barriers are available as part of the RSP model.

Given this description, RSP has the following features:

– Familiarity of shared memory programming. Like CCSM, RSP uses standard load and store instructions to communicate.
– Emphasis on locality of reference. RSP encourages programmers to write code in such a way that loads always target local, physically close memory.
– One-sided communication. In RSP programs, data is pushed from the producer to the consumer. Unlike two-sided communication schemes that require a *send* to be accompanied by a *receive*, remote stores do not require acknowledgement in this model. One-sided communication leads to code that is both easier to write and higher performing than a two-sided model.
– No explicit support for bulk transfers. The RSP model does not support a special *put* operation like SHMEM [5] and UPC [6][1]. This omission is designed to encourage programmers to store data remotely as it is produced so that data is transferred from the registers of the producer to the cache of the consumer with no extra buffering or copying.
– No support for remote reads. The RSP model does not support remote loads or *get* operations. This omission is designed to encourage users to structure code such that all reads target local memory, ensuring that loads have minimum latency. RSP focuses on minimizing load latency for two reasons. First, loads are more common than stores. Second, it is easier to tolerate store latency than load latency. One can overlap communication and computation with simple hardware support using remote stores, but such overlap would be hard to achieve for remote loads without more hardware support, like a direct memory access (DMA) engine or hardware prefetching.

[1] The C function `memcpy` can provide the semantics of a bulk transfer function in the RSP model, but the RSP model does not assume any additional bulk data movement mechanisms.

3 Implementation of the RSP Model

The RSP model is designed specifically to be incrementally achievable in multicore architectures that support loads and stores using a small set of hardware features that have a large impact on program performance. In the RSP model data is transfered from the registers of a producer into the cache of a consumer as illustrated in Figure 2(a). The data is not buffered on the producer to be transferred in bulk, but ideally each datum is sent as it is produced. This model results in many small messages and does not attempt to amortize the cost of communication by bundling many messages into a small number of large messages. In trade, RSP programs exhibit good locality of reference, have lower load latencies, and outperform CCSM on highly parallel multicores.

RSP needs hardware and operating system support for the following mechanisms: allocating remotely-writable data, executing store instructions targeting remotely-writable data, maintaining memory consistency, and executing synchronization operations targeting remotely-writable data. These features are discussed in turn.

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

Store instructions targeting remote data. Processes may execute store instructions where the destination register specifies an address in remotely writable memory. The processor executing such a store should not allocate the cache-line, but forward the operation to the consumer processor that allocated the data. This forwarding should be handled in hardware and requires that 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 any other write. In RSP, data that is allocated as remotely writable can only be cached in the allocating processor. This protocol preserves locality of reference by guaranteeing that reads are always local, ensuring minimal load latency.

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 allows one to allocate locks, condition variables, and other synchronization structures in remotely writable memory.

With support for these features a multicore architecture can efficiently implement remote store programs. This set of features represents a small, incremental change over the set of features that would be required on any multicore architecture. On an architecture supporting loads and stores, a core must be able to 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.

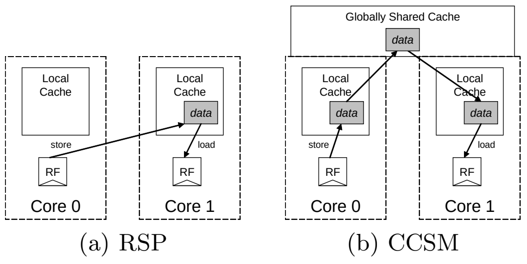


Fig. 2. Communication mechanisms in multicore. The figure illustrates two different mechanisms for sending data from Core 0 to Core 1. RSP transfers data directly from the sender's registers (the box labeled "RF") to the receiver's local memory (cache or scratch-pad). CCSM transfers data through the global address space.

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. Unlike RSP, CCSM hardware transfers data from registers to a local cache and then to a globally shared cache or memory as illustrated in Figure 2(b). To support CCSM one could implement either a snoopy or a directory-based coherence protocol. A snoopy protocol would require a centralized structure which would be difficult to scale to large numbers of cores. Directory-based schemes provide better scalability, but require additional O(P) bits (where P is the number of processors) to store directory information [7] and possibly another network that is dedicated to coherence messages. In addition to the extra hardware structures, a cache coherence protocol requires additional design and verification complexity.

4 Evaluation Methodology

This section presents the approach used to evaluate the remote store paradigm on the TILEPro64 processor. To begin, the TILEPro64 and its implementation of the RSP model are described. Next the eight benchmarks and the parameters used in this evaluation are discussed.

4.1 The TILEPro64

The TILEPro64 processor is a 64 core multicore processor with hardware support for cache-coherent shared memory. Each of the 64 cores is an identical three-wide VLIW capable of running SMP Linux. Each core has a unified 64KB L2 cache, and the L2 caches can be shared among cores to provide an effective 4MB of shared, coherent, and distributed L3 cache. Cores are connected through six low-latency, two-dimensional mesh interconnects [8]. Two of these networks carry user data, while the other four handle memory, I/O and cache-coherence

Application	Input
Bitonic Sort	Integer list of length 128k
Convolution	1920 × 1080 Image
Error Diffusion	4096 × 2048 Image
2D FFT	256 × 256 matrix of complex 16-bit fixed-point values
Histogram	4096 × 2048 Image
Matrix Multiply	Two 512 × 512 matrices of 16-bit fixed-point values
Transpose	1024 × 1024 matrix of integers
H.264	Raw 1280 × 720 video

Table 1. Benchmark applications used to compare RSP to CCSM.

traffic. The TILEPro64 can run off-the-shelf POSIX threads programs under SMP Linux.

The TILEPro64 uses a directory-based cache-coherence scheme with full-map directories. Loads and stores to shared memory which miss in the local L2 cache generate coherence messages that are handled by a directory on a remote core. The latency of these coherence messages is proportional to twice the distance between the accessing core and the core that contains the directory for that memory location. Clearly, if the directory is physically close, the latency is less than if the directory is physically far away. Ideally, one wants to access directories that are physically close to minimize latency.

In addition to standard cache-coherent shared memory, the TILEPro64 allows users to allocate shared memory that is *homed* on the allocating core. On this home core, reads and writes function as usual. However, when cores write remotely homed memory, no cache line is allocated on the remote core. Instead, these writes stream out of the writing core to the home cache without generating any other coherence traffic. This homed memory is used to implement remotely writable memory for remote store programs.

4.2 Benchmark Applications

Table 1 presents the application benchmarks used to compare RSP to CCSM. These benchmarks include representatives from image, video, and digital signal processing. For each benchmark optimized implementations are developed for a single core and then for both the CCSM and RSP paradigms.

For both paradigms, several common optimizations are used. Cache-blocking is used to reduce the number of data cache misses. Flags are used instead of locks whenever applicable (one exception is the histogram benchmark described below) to reduce contention. All stacks and read-only data are allocated as private for both paradigms, meaning that stack accesses and accesses to global constant data will not result in coherence traffic.

In the *bitonic sort* benchmark, a list of integers is sorted by dividing it among processes. Each process sorts its assigned integers independently using quicksort and locally sorted lists are combined through a series of merge steps. Each merge requires a process to exchange data with a partner. In the CCSM implementation the array is stored in global shared memory and all updates are done in place. In the RSP implementation, each process allocates remotely writable memory and data is copied into the partner's address space using remote stores. Barriers synchronize both implementations.

The *convolution* benchmark convolves an input image with a 3×3 mask. Each process is responsible for producing separate rows of the output image. In the CCSM implementation both the input and output arrays are stored in global shared memory. In the RSP implementation, each process allocates remotely writable memory to store its assigned rows of both the input and output images. The RSP implementation requires allocation of additional memory to hold input values on the border of processes' assigned regions. Barrier synchronization is used in both implementations.

The *error diffusion* benchmark performs Floyd-Steinberg Error Diffusion on an input image. Each process is responsible for performing computation on separate columns of pixels, and the computation is done in place. Pixels on the border between processes are shared and these values are both read and written by neighboring processes. In the CCSM implementation, the image is stored in global memory and flags (also stored in global shared memory) are used to synchronize access to the image. In the RSP implementation, each process allocates private memory to store its assigned regions of the image. In addition, each process allocates remotely writable regions of memory to store the shared pixels and flags. When a process produces a value that is to be shared, it stores one copy locally, then stores another copy, and sets a flag in remotely writable memory.

The *FFT* benchmark performs a two-dimensional Fast Fourier Transform (2DFFT) on an input matrix. The benchmark first performs an FFT on each row and then performs an FFT on each column. Both implementations use out-of-place computation for both the row and column FFTs. Before executing the column FFTs, data is transposed in memory so that the consecutive elements in a column are unit distance apart. Each process is assigned a set of row FFTs and a set of column FFTs. In the CCSM implementation, the input, temporary, and output matrices are all stored in global shared memory. In the RSP implementation, each process allocates remotely writable memory to store results of the row FFTs, and other data is stored in private, local memory. As the row FFTs are computed, their results are written directly into this remotely writable memory.

The *histogram* benchmark computes a histogram representing the tonal distribution of an input image. Each process is assigned a separate region of the image. In the CCSM implementation, the image and the histogram are both allocated in global shared memory. As a process works on its part of the input image, it updates the histogram. Each bin of the histogram is guarded by a separate lock. In the RSP implementation, the histogram is distributed across processes. Each process allocates remotely writable memory to store temporary results from other processes. Then, the processes all compute the histogram of their assigned region of the image. Once these local histograms are completed, each process writes the appropriate local values to a remote process which performs a reduction on the local data. In the RSP implementation, a barrier is used to synchronize.

The *matrix multiplication* benchmark multiplies two input matrices to compute an output matrix. Each process is responsible for computing a separate region of the output matrix. In the CCSM implementation all matrices are allocated in global shared memory. In the RSP implementation, each process allocates remotely writable memory to store the rows and columns of the input matrix that are needed to compute the assigned region of the output. In both implementations barriers are used to synchronize.

The *transpose* benchmark performs the transpose of an input matrix. Each process is responsible for producing a separate region of the output matrix. In the SM implementation, both the input and output matrices are allocated in globally addressable shared memory. In the RSP implementation each process allocates remotely writable memory to hold a region of the output matrix. Each process performs its part of the transpose by reading local values and writing them to the appropriate location in remotely writable memory.

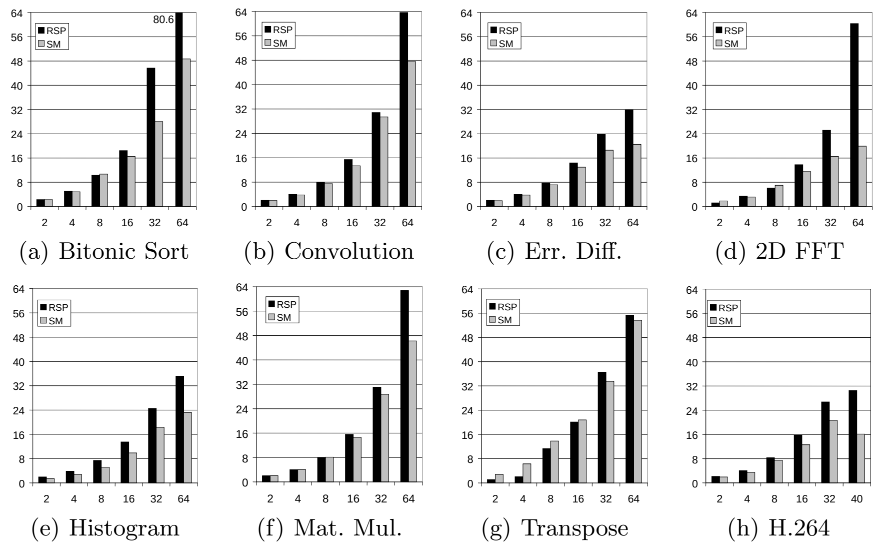
The *H.264* benchmark performs Baseline profile H.264 encoding on raw high-definition video. Both implementations attempt to minimize latency by partitioning the encoding of a frame among multiple processes. Each process is responsible for encoding its assigned region of the frame. To perform this encoding each process needs data from those processes that are assigned neighboring regions of the frame. In the CCSM implementation frames and associated meta-data are stored in global shared memory. In the RSP implementation, each process allocates remotely writable memory to store its assigned part of the frame and the overlapping regions assigned to neighboring processes. These overlapping regions of data are stored locally in the process that created them and then copied to neighboring processes using remote stores. Both implementations synchronize using a combination of flags and barriers. Additionally, both implementations are limited to a maximum of 40 processes.

5 Performance Evaluation

This section evaluates the performance of remote store programming. First, the speedup of RSP implementations of each benchmark is shown and compared to the speedup achieved with CCSM implementations. Next, the load-latency of each of the benchmarks is evaluated.

5.1 Speedup Evaluation

Figure 5.1 illustrates the performance of both the CCSM and RSP implementations of each of the eight benchmarks as the number of cores is varied from 2 to 64. These results all use one process per core. All speedups are computed relative to an optimized single core implementation. Higher bars represent greater speedup and greater performance.



(a) Bitonic Sort (b) Convolution (c) Err. Diff. (d) 2D FFT

(e) Histogram (f) Mat. Mul. (g) Transpose (h) H.264

Fig. 3. Speedup of RSP and CCSM implementations of eight benchmarks. Speedup is shown on the y-axis while number of cores is on the x-axis. (The RSP implementation of bitonic sort achieves a speedup of 80.6 on 64 cores.)

For these benchmarks, RSP achieves greater performance than CCSM using large numbers of cores. Using 16 cores or fewer, RSP is generally higher performing, but there are some exceptions and overall the approaches achieve similar performance. However, using 32 or more cores, RSP begins to clearly out perform CCSM as shown in Figure 4.

Figure 4 shows the ratio of RSP to CCSM performance for each of the eight benchmarks using 2 to 64 cores. Ratios of less than one indicate that RSP is slower while ratios greater than one indicate that RSP is faster (higher bars are better). Figure 4 shows that the performance benefits of RSP are greater for large numbers of cores. When using more than 32 cores, RSP achieves speedup over CCSM for each of the eight benchmarks. When using 64 cores, RSP achieves greater than 1.25x the performance of CCSM for 7 of the 8 benchmarks and greater than 1.5x the performance of CCSM for 5 of 8. For H.264 on 40 cores, RSP achieves greater than 1.8x the performance of CCSM. For the FFT on 64 cores, RSP achieves greater than 3x the performance of CCSM.

The transpose benchmark defies the general trend in that the RSP and SM implementations achieve comparable performance for all numbers of cores. The reason for the similar speedup numbers is that the transpose benchmark consists of loads of input values followed by stores which put the transposed matrix in place. The benchmark time includes the time required for all stores to complete, so the transpose represents one benchmark where store latency is critical. In this

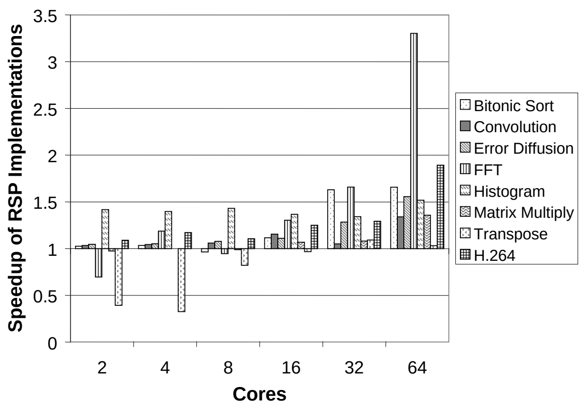


Fig. 4. Performance comparison of RSP and CCSM benchmarks. The speedup of RSP compared to CCSM is shown as a function of the number of cores. Speedups of less than 1 indicate RSP is slower. Speedups of more than 1 indicate RSP is faster. (The H.264 speedup listed for 64 represents the value measured using 40 cores.)

case the low load latency of the RSP implementation is cancelled out by its high store latency.

The relative performance gain of RSP increases with an increasing number of cores. Locality becomes a larger factor in performance with large core count and RSP allows software to control locality while CCSM does not. When using a large number of cores, a cache miss in a CCSM application can result in accessing a cache-coherence directory that is physically far away. In this case many of the distant accesses are loads, and the resulting high load latency has a dramatic effect on performance. However, in the RSP implementation loads do not generate coherence traffic to remote cores and load latency is lower.

5.2 Locality Evaluation

As discussed in Section 2 and Section 3 the RSP model and its implementation emphasize the use of physically close memory with the goal of minimizing load latency. The Tilera simulator allows one to measure the latency of load instructions that access the L2 cache. This statistic keeps track of the time it takes to service an L1 data cache miss.

The average L2 load latency is recorded for each of the eight benchmarks. Figure 5 shows this data expressed as the ratio of RSP load latency to CCSM load latency for each of the eight benchmarks on 2 to 64 cores. A ratio of 1 indicates that both implementations achieve the same load latency. A ratio of less than 1 indicates that the RSP load latency is lower than that of CCSM (lower bars are better).

The L2 load latency generally follows the same trend as the speedup results. Latencies are similar for both implementations when using a small number of

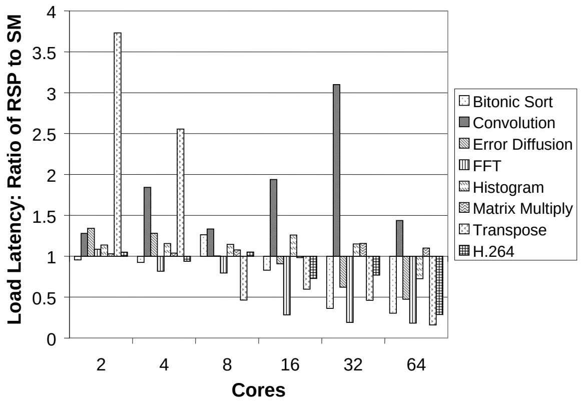


Fig. 5. Ratio of L2 load latency for RSP and CCSM benchmarks. The ratio of the load latencies for RSP and CCSM implementations is shown as a function of the number of cores. Ratios less than 1 indicate RSP is lower latency than CCSM, while ratios greater than 1 indicate RSP load latency is higher.(The H.264 load-latency listed for 64 represents the value measured using 40 cores.)

cores, but the RSP latency tends to be much lower using large numbers of cores. Using 64 cores, RSP load latency is lower for 6 of the 8 applications.

The two applications for which RSP load latency is higher with 64 cores are the convolution and matrix multiplication. Figure 4 shows that the RSP implementations of these benchmarks out perform CCSM despite the higher load latency.

The difference in performance for the convolution is the result of an increased number of data TLB misses on the part of the CCSM implementation. In fact, the CCSM implementation of the convolution produces almost 9 times more data TLB misses than the RSP implementation. The profiling tool does not count time spent in the TLB miss towards load latency (it is accounted for as a separate statistic). This TLB behavior is not an inherent aspect of the CCSM programming model, but rather a random effect due to the combination of the input image size and the way in which processes in this application access globally addressable shared memory. However, page misses are not an issue in the RSP implementation of this application because each process allocates data locally and the amount of local data easily fits in the TLB.

For matrix multiplication the difference in performance is explained by the worst case load latency. To compute average load latency, the latency of all loads on all cores is averaged; however, for the CCSM matrix multiplication one core has a consistently higher load latency than the others. In the CCSM implementation, Core 0 has an average load latency of 13 cycles, while the average for all cores is 10.1. This difference is due to the fact that core 0 accesses directories that are, on average, farther away than those accessed by other cores. In contrast, the RSP implementation has a maximum per-core load latency of 11.5 cycles with an average of 11.1. Although the CCSM matrix multiply has a lower average load latency, the maximum is higher and the performance of the application is determined by the slowest core.

On the whole, RSP fulfills its promise of exploiting locality to minimize load latency. The predictability of load latency and TLB behavior under RSP is an advantage in embedded systems that tend to require repeatable performance to meet real-time requirements.

6 Related Work

Both the partitioned global address space (PGAS) model and RSP combine the familiarity of CCSM with explicit control over locality for performance[2]. The PGAS model is designed to be implemented in high-level languages such as Unified Parallel C [6], Titanium [9], and Co-Array Fortran (CAF) [10]. The RSP model is designed to be implemented in hardware and can serve as the lowest-level communication primitive for an architecture. In this sense, the two approaches are complementary. A PGAS implementation can benefit from targeting RSP to achieve higher performance than would be available through standard CCSM mechanisms. A high-level PGAS language targeting RSP would make the efficiency of RSP available to a greater number of programmers.

Despite the similarities, there are some differences in the way programs are written using the PGAS and RSP models. While the PGAS model can target multicore, it was designed for multichip parallel computers with physically distributed, non-uniform memory access (NUMA) memory architectures like clusters and supercomputers. On these architectures programs typically perform best when total communication is reduced, the remaining communication is bundled into a small number of large messages, and communication and computation is overlapped. These optimization techniques affect the interface as most PGAS implementations include *put* and *get* (or similar) operations that are used to transfer large buffers between local and remote DRAMs and efficient programs are structured to make such infrequent, large transfers.

In contrast, the RSP model targets multicore architectures which support shared address spaces built using relatively powerful on-chip networks to connect physically distributed caches on the same chip. The network makes it possible for processors to transfer data from cache to cache (e.g. IBM Cell [11]), registers to cache (e.g. TILEPro64 as described above [8]), or even from registers to registers (e.g. Raw [12]). The RSP model is designed to support fine-grained communication on these types of multicore architectures. Specifically, RSP is designed to encourage programmers to structure code so that data is transferred from registers to cache as data is produced without buffering or bulk transfer.

The reflective memory model also combines the familiarity of CCSM with mechanisms that allow users to control locality [13]. This multichip model supports a paradigm in which writes in one process' address space appear (or are "reflected") in another address space. Unlike the PGAS model, reflective memory systems are designed to efficiently support individual writes as a communication primitive. The two reflective memory implementations that share the most in common with RSP are virtual memory mapped communication (VMMC) [14] as implemented on the SHRIMP processor [4] and DEC's Memory Channels [3].

Like RSP, VMMC [14] uses writes to transfer data between processors' virtual address spaces. Using the "automatic update" option of VMMC, both the producer and the consumer allocate a data buffer to communicate. The producer writes its local copy of the memory, the data is stored locally, and the writes are put on a system bus. The consumer snoops this bus and intercepts writes that also map to its address space. Unlike VMMC, the RSP implementation uses messages and can be implemented on a mesh network without requiring a snoopy protocol or a centralized bus. Furthermore, RSP does not require the producer to keep a separate copy of the data in its own local memory.

The most significant difference between Memory Channels (MC) and RSP is that MC allows pages of the shared address space to be mapped to a processor for read or write access, but not read/write. RSP allows read/write mappings for home nodes. All 8 benchmarks discussed in this paper make use of read/write mappings, and disallowing this, as in MC, would add code complexity, copy operations, or both to RSP applications. Unlike RSP, MC requires the OS to "pin" shared pages to communicating processors. This restriction limits the number of sharers to the size of the page table limiting scalability and may effectively waste a page table entry for processes that communicate infrequently. Furthermore, RSP can be implemented on heterogeneous cores while MC is restricted to homogeneous clusters. Finally, MC has hardware support for broadcast/multicast. While RSP lacks this support, it does not require the additional hardware and only 2 of the 8 applications could make use of multicast.

Leverich et al. performed a similar study comparing CCSM to streaming memory for multicore [15]. This study found similar performance for the two models even though stream programming allows a user to explicitly control locality. In contrast, the results presented here show that allowing a user to control locality can have significant benefits for large numbers of cores. There are two major differences in the approaches that may account for the different findings. First, the Leverich study uses a snooping, bus-based coherence protocol, while the study presented here uses a directory scheme built on a mesh network. The bus-based scheme may provide better performance for small numbers of cores, but has limited scalability to large multicores. Second, and perhaps most importantly, the Leverich study limits the comparison to a maximum of sixteen core chips, where the study presented here includes performance using 32 and 64 core processors. In fact, for RSP the most significant performance gains are found when using 32 or more cores.

7 Conclusion

The RSP model is designed to provide high performance and ease-of-use while requiring only incremental hardware support in multicore architectures. As demonstrated, RSP implementations of eight benchmarks exhibit better performance than shared memory for large numbers of cores. RSP can augment directory-based cache-coherence schemes for multicores with many processors. Standard shared memory techniques can be used for code that is highly dynamic in its memory access patterns, while RSP could be used for performance critical sections of regularly structured code. Alternatively, RSP can be used as the only paradigm to provide a convenient and efficient programming model on multicore DSPs.

References

1. Shan, H., Singh, J.P.: A comparison of MPI, SHMEM and cache-coherent shared address space programming models on a tightly-coupled multiprocessors. Int. J. Parallel Program. **29**(3) (2001) 283–318
2. Carlson, W., El-Ghazawi, T., Numric, R., Yelick, K.: Programming with the PGAS model. In: IEEE/ACM SC2003. (2003)
3. Gillett, R.B.: Memory channel network for PCI. IEEE Micro **16**(1) (1996) 12–18
4. Blumrich, M.A., Dubnicki, C., Felten, E.W., Li, K.: Protected, user-level dma for the shrimp network interface. In: In Proceedings of the Second IEEE Symposium on High-Performance Computer Architecture. (1996) 154–165
5. Quadrics: SHMEM Programming Manual. Quadrics Supercomputers World Ltd, Bristol, UK (2001)
6. Chauvin, S., Saha, P., Cantonnet, F., Annareddy, S., El-Ghazawi, T.: UPC Manual. (May 2005) http://upc.gwu.edu/downloads/Manual-1.2.pdf.
7. Hennessy, J.L., Patterson, D.A.: Computer Architecture: A Quantitative Approach. Morgan Kaufmann, San Mateo, CA
8. Wentzlaff, D., Griffin, P., Hoffmann, H., Bao, L., Edwards, B., Ramey, C., Mattina, M., Miao, C.C., Brown III, J.F., Agarwal, A.: On-chip interconnection architecture of the Tile Processor. IEEE Micro **27**(5) (2007) 15–31
9. Yelick, K., Semenzato, L., Pike, G., Miyamoto, C., Liblit, B., Krishnamurthy, A., Hilfinger, P., Graham, S., Gay, D., Colella, P., Aiken, A.: Titanium: A high-performance Java dialect. In: In ACM. (1998) 10–11
10. Numrich, R.W., Reid, J.: Co-array Fortran for parallel programming. SIGPLAN Fortran Forum **17**(2) (1998) 1–31
11. Kistler, M., Perrone, M., Petrini, F.: Cell multiprocessor communication network: Built for speed. Micro, IEEE **26**(3) (May-June 2006) 10–23
12. Taylor, M.B., Lee, W., Miller, J., Wentzlaff, D., Bratt, I., Greenwald, B., Hoffmann, H., Johnson, P., Kim, J., Psota, J., Saraf, A., Shnidman, N., Strumpen, V., Amarasinghe, S., Agarwal, A.: Evaluation of the Raw Microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams. In: International Symposium on Computer Architecture. (June 2004)
13. Jovanovic, M., Milutinovic, V.: An overview of reflective memory systems. IEEE Concurrency **7**(2) (1999) 56–64
14. Dubnicki, C., Iftode, L., Felten, E., Li, K.: Software support for virtual memory-mapped communication. (Apr 1996) 372–381
15. Leverich, J., Arakida, H., Solomatnikov, A., Firoozshahian, A., Horowitz, M., Kozyrakis, C.: Comparing memory systems for chip multiprocessors. SIGARCH Comput. Archit. News **35**(2) (2007) 358–368