Computer Science and Artificial Intelligence Laboratory
Technical Report

MIT-CSAIL-TR-2009-017 May 5, 2009

Remote Store Programming: Mechanisms and Performance

Henry Hoffmann, David Wentzlaff, and Anant Agarwal

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

Remote Store Programming: Mechanisms and Performance

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

Abstract

This paper presents remote store programming (RSP). This paradigm combines usability and efficiency through the exploitation of a simple hardware mechanism, the remote store, which can easily be added to existing multicores. Remote store programs are marked by fine-grained and one-sided communication which results in a stream of data flowing from the registers of a sending process to the cache of a destination process. 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 both cache-coherent shared memory and direct memory access (DMA) based approaches using the TILEPro64 processor. The paper studies two applications: a two-dimensional Fast Fourier Transform (2D FFT) and an H.264 encoder for high-definition video. For a 2D FFT using 56 cores, RSP is $1.64\times$ *faster than DMA and* $4.4\times$ *faster than shared memory. For an H.264 encoder using 40 cores, RSP achieves the same performance as DMA and* $4.8\times$ *the performance of shared memory. Along with these performance advantages, RSP requires the least hardware support of the three. RSP's features, performance, and hardware simplicity make it well suited to the embedded processing domain.*

1. Introduction

The emergence of multicore architectures has generated increased interest in parallel programming. While a great deal of research has been done in the field of multichip parallel computing, multicore architectures offer a new set of opportunities and challenges. This paper proposes the remote store programming (RSP) model, which is designed to be easy to use, efficient, and incrementally supportable in multicore architectures that support loads and stores.

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 easy to schedule and consumer processes are guaranteed to read physically close, or local, memory. This locality of reference is important for performance on non-uniform memory access (NUMA) architectures.

The RSP model is similar to both the partitioned global address space (PGAS) [6, 8, 28, 16] and virtual memory mapped communication (VMMC) [11] models which were developed for multichip computer architectures. All three paradigms combine the programming ease of a shared address space with features that allow programmers to ensure that performance critical memory references access physically close hardware. However, the RSP model is distinguished in the following two ways. First, RSP is defined by including only mechanisms that require incremental hardware support in multicores. For example, the RSP model does not include features that require hardware support for cache coherence or direct memory access (DMA), which is commonly used in PGAS and VMMC implementations[1]. Second, RSP programs are characterized by extremely fine grain communication that streams from a source processor's registers to a destination processor's cache. In comparison, multichip programs require data to be buffered in memory on the producing core and then transferred to the consuming core in bulk. These multichip programs are characterized by bulk data transfers from DRAM to DRAM. While the VMMC and PGAS models are well-suited to multichip distributed memory architectures, the RSP model can achieve higher performance on a multicore architecture while requiring less hardware support.

The performance of the remote store programming model is evaluated by emulating it using the TILEPro64 processor [26]. This implementation demonstrates that the RSP paradigm can achieve efficient parallel implementations on important multicore applications like video encod-

[1] While cache-coherence is not required, RSP mechanisms can be easily implemented on an architecture with cache-coherent shared memory.

ing. An RSP implementation of an H.264 encoder achieves a speedup of 24.7x using 40 processes, while a 2D FFT achieves a speedup of 57.7x using 56 processes. Additionally, the TILEPro64 allows comparison of remote store programming to both cache-coherent shared memory and DMA based implementations. Results show that RSP can achieve over five times the performance of shared memory for large numbers of processes. This speedup relative to shared memory is due to its emphasis on locality-of-reference, as RSP programs always access physically close memory and minimize load latencies. In addition, RSP performance is comparable to that of DMA in the worst case and as much as 64 % better in the best case. RSP achieves this performance with less hardware support.

While incrementally supportable in many multicores, RSP may be best suited to emerging multicore digital signal processors (DSPs) such as the three-core chip produced by Texas Instruments [24]. The RSP model is a good match for the regular computations commonly found in DSP applications and it is supportable with a small amount of hardware. Current multichip DSP approaches often communicate using DMA to transfer data between shared DRAM and local scratch-pad memories. Multicore DSPs could communicate using remote stores to write data directly into scratch-pad memory on remote cores, eliminating DMA from intra-core communication.

The remainder of this paper is organized as follows. Section 2 presents the remote store programming model and discusses some of its features and drawbacks. Section 3 discusses the hardware and operating system support required to efficiently implement the RSP paradigm on a multicore architecture. Section 4 describes implementation of the RSP model on the TILEPro64 processor and compares the performance of RSP applications to shared memory and DMA-based approaches. Related work is discussed in Section 5, and the paper concludes in Section 6.

2. The remote store programming model

This section discusses programming using the remote store model. The term *process* refers to the basic unit of program execution. A parallel program is one that has more than one process actively performing computation at some point during its execution. A parallel programming paradigm is a framework for coordinating the computation of processes within a parallel program. This work assumes that there is a one-to-one mapping between processes and processors, but that restriction is easily relaxed. A parallel programming paradigm is distinguished by three features:

1. The *process model* - the programmer's "mental picture" of the underlying hardware and system software.
2. The *communication* mechanism - the protocol processes use to transfer data.
3. The *synchronization* mechanism - the protocol processes use to ensure ordering and atomicity constraints.

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 remote store programming is illustrated in Figure 1. The key idea of the remote store paradigm is that programmers ensure that a process always reads local memory.

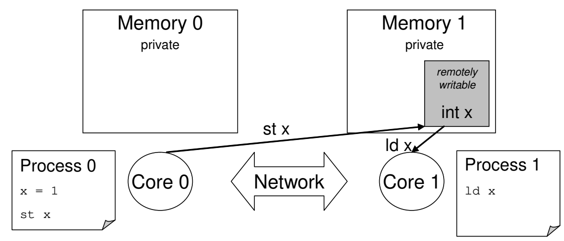


Figure 1. Illustration of the remote store programming model. There are two cores, each of which executes a process. Process 1 allocates a remotely-writable region of memory to hold the integer `x`. Process 0 writes a new value into `x`, and this new data travels from Process 0's registers to Process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

The communication mechanism. In a remote store 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 a remote store 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 shared memory, 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, which leads to high performance on NUMA architectures.
- One-sided communication. In RSP programs, data is pushed from the producer to the consumer using the remote store mechanism. 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 and UPC[2]. This omission is designed to encourage programmers to store data remotely as it is produced. This style allows data to be transferred from the registers of the producer to the cache of the consumer with no extra buffering or copying. In addition, the lack of bulk transfers reduces the hardware burden as no specialized hardware is required.
- 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 expensive hardware, like a DMA engine, to prefetch data into the local cache.

3. Implementation of the RSP model

This section describes the desired hardware and operating system support for the implementation of the remote store programming model on a multicore architecture. 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.

The vision of the RSP model is one in which 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 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, remote store programs exhibit good locality of reference, suffer fewer data cache misses, and outperform shared memory and DMA-based approaches on multicores.

To realize this goal, 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 remote 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. The consumer can support either write allocate or no-write allocate policies. In RSP, data that is allocated as remotely writable can only exist in the cache of 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 op-

[2] 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.

erations, 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 shared 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. 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.

3.1. Comparison to other models

This section illustrates the benefits of the remote store approach by comparing the data movement and hardware support required for other communication models. Cache-coherent shared memory is discussed first, then DMA.

Cache-coherent shared memory 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 cache-coherent shared memory 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 [13] 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. The overall cost for cache-coherence is much greater than that to support remote stores.

To communicate using a DMA, a processor first produces a value in its registers, then stores it to local memory or cache, and finally transfers the data to a remote cache or memory by invoking the DMA engine as illustrated in Figure 2(c). Hardware support for DMA requires dedicated die area for the DMA. The cost in area is small, but adding a DMA engine increases the design and verification time of the hardware. Additionally the DMA is harder to program than cache-coherent shared memory or RSP. Also, one would like to use DMA to transfer data directly from one core to another without affecting the computation taking place on those cores. This is not possible if the local memory on the cores only has a single read and write port

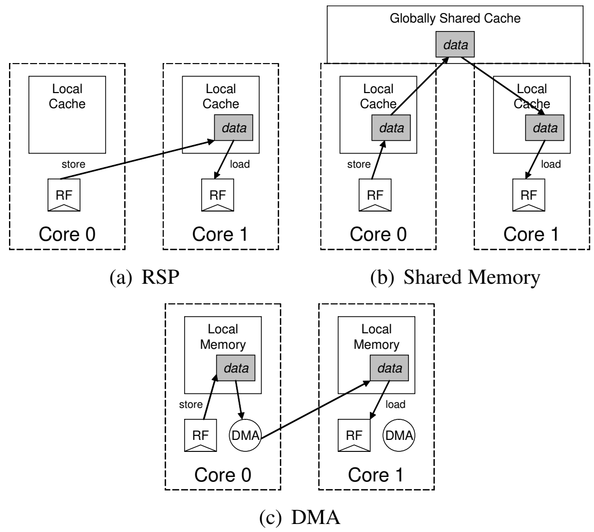


Figure 2. Communication mechanisms in multicore. The figure illustrates three different mechanisms for sending data from Core 0 to Core 1. RSP transfers data directly from the sender's registers (the box labeled "RF") to the receiver's local memory. Cache-coherent shared memory transfers data through the global address space. DMA stores data locally and then copies data from the producer to the consumer.

because the DMA will compete with the core for memory bandwidth. For the DMA to run with no noticeable effect on the cores involved in the transfer, it would need a second read and write port to the local memories, which would double their size.

4. Performance of remote store programs

This section discusses the performance of the remote store paradigm as implemented on the TILEPro64 processor [26]. To begin, the TILEPro64 and its implementation of the RSP model are described. Next is an in-depth look at two applications, the 2D FFT and H.264 encoding, and a comparison of the performance of remote store implementations to that of both cache-coherent shared memory and DMA-based approaches. Finally, speedup numbers for several different applications are presented.

4.1. The TILEPro64

The TILEPro64 processor is a 64 core multicore processor with hardware support for both cache-coherent shared memory, message passing, and a core-to-core direct memory access (DMA) engine. Each of the 64 cores is an identical three-wide VLIW capable of running SMP Linux. In addition to standard RISC instructions, all cores support a SIMD instruction set designed to accelerate video, image, and digital signal processing applications. Each core has a unified 64KB L2 cache. Further, all 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. Two of these networks carry user data, while the other four handle memory, I/O and coherence traffic. The TILEPro64 can run off-the-shelf POSIX threads [5] programs under SMP Linux. Alternatively, users can transfer data using the core-to-core DMA engine.

The TILEPro64 uses a variation of a directory-based cache-coherence scheme, so loads and stores to shared memory which miss in the local L2 cache generate coherence messages that are handled by 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. Ideally, one wants to access directories that are physically close to minimize latency.

The TILEPro64 DMA interface is similar to that of remote direct memory access (RDMA) as implemented in interconnects like InfiniBand [1], Myrinet [4], and Quadrics [17]. However the TILEPro64 allows data to be transferred from cache to cache over the on-chip network rather than transferring data from one DRAM to another over a local area network.

In addition to standard cache-coherent shared memory, the TILEPro64 allows users to allocate shared memory that is *homed* on the allocating core. On the 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, writes to remotely homed memory 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.

The various mechanisms and the large number of cores available on the TILEPro64 make it an excellent platform for comparing the performance of remote store programs to that of two common techniques: cache-coherent shared memory and DMA. This comparison is made by implementing two applications in each of these three paradigms and presenting the relative performance for each. First the two-dimensional fast Fourier transform (2D FFT) is described and then an H.264 encoder for high-definition (HD) video.

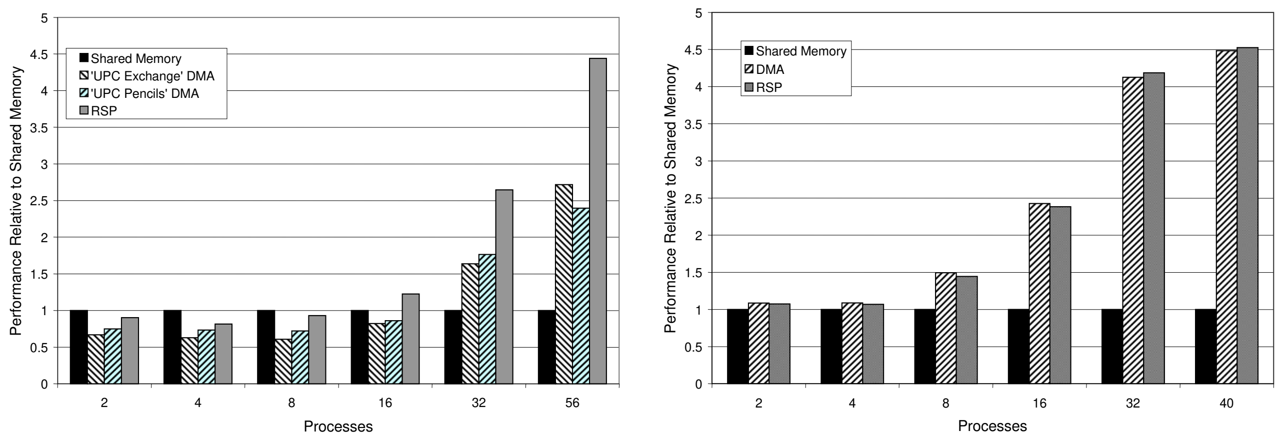
4.2. 2D FFT

To compute a two-dimensional FFT on an $N \times N$ matrix, one first performs an FFT on each row and then performs an FFT on each column [15]. This implementation uses an out-of-place computation for both the row and column FFTs, so the results of the row FFTs are stored in a temporary matrix that is used as input for the column FFTs. Output is written into a third matrix. Before executing the column FFTs, the temporary data is transposed in memory so that the consecutive elements in a column are unit distance apart. This layout results in better cache performance during the column FFTs. In a parallel implementation of the 2D FFT using P processes, each process computes N/P rows and then N/P columns. Barrier synchronization is used between the row and column FFTs to ensure correctness. On a distributed memory architecture, this pattern requires that each process communicates with every other process. The same algorithm and the same barrier construct is used for each of the implementations and the focus is on the difference in data movement.

Cache-coherent shared memory FFT. In this implementation each of the arrays, input, temporary, and output, are allocated in globally addressable shared memory. Data is read and written without regard to the physical location of any of the data or corresponding coherence directories. Using shared memory the transpose is performed by storing the results of the row FFTs to the temporary array as they are produced. As illustrated in Figure 2(b), the data is transferred from the producer to the consumer through the global address space.

DMA-based FFT. A DMA-based approach is used to simulate what a PGAS implementation employing existing multichip techniques might look like on the TILEPro64. Bell et. al. describe a UPC implementation of a three-dimensional FFT which uses one-sided RDMA transactions on a multichip computer architecture [3]. The DMA-based implementations of the 2D FFT on TILEPro64 are patterned after two of those described by Bell. For both DMA implementations, each process allocates private data to hold its assigned rows and columns of the input and output data. Additionally, each process allocates two buffers to hold portions of the temporary matrix. The first buffer is allocated in local memory and the second buffer is allocated in homed memory. The DMA engine transfers data from the local memory to the homed memory. As illustrated in Figure 2(c), a producer process stores data in a local buffer and then the core-to-core DMA engine copies that buffer to the consumer.

Given this setup, two different DMA based approaches are implemented. In the first approach, modeled after Bell's "UPC Exchange" implementation, all row FFTs are computed first, then all DMA transactions are enqueued, the barrier is entered, and finally the column FFTs are performed. Using the TILEPro64's 2D DMA engine and the exchange technique, each core initiates P DMA transactions to move data. The exchange implementation results



(a) 2D FFT

(b) H.264 encode

Figure 3. Performance comparison of cache-coherent shared memory, DMA, 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.

in minimal communication, but offers little opportunity for overlapping communication and computation. The second approach is modeled after Bell's "UPC Overlap Pencils," which was the best performing UPC approach on a variety of multichip architectures. Using the pencils approach, P DMA transactions are enqueued after each row is completed resulting in a total of $N \cdot P$ transactions per core. After completing the final row FFT, the program ensures all DMA transactions have finished and then executes the column FFTs. This second approach requires more individual DMA transactions and thus more overhead, but it allows a greater opportunity for overlapping communication and computation. In both of these approaches the temporary data is transposed locally once it has reached its destination.

Remote Store FFT. In the RSP implementation, each process allocates a private buffer to hold its portion of the input and a separate private buffer to hold its portion of the output. The temporary array is allocated using homed memory to emulate remotely-writable memory. Each process makes its temporary array available to all other processes. As a process completes individual elements of its assigned row FFTs, these elements are written directly into the appropriate regions of remotely writable memory resulting in a total of $N \cdot N/P$ messages per core. In the RSP implementation, data is sent directly from the registers of the producer to the cache of the consumer without any buffering, as illustrated in Figure 2(a). As in the shared memory implementation, the temporary data is transposed as it is stored remotely so there is no separate transposition step.

Both the DMA-based approaches and the RSP approach use locality and one-sided communication. However, because each individual word is sent as a separate message, the RSP implementation has even greater communication overhead than either of the DMA-based implementations described above. Due to its use of ultra-fine-grain communication, RSP provides the maximum opportunity for overlapping communication and computation. Additionally, it requires no communication instructions other than standard stores. Finally, this approach does not require that data is buffered on the producer side. Since no data is buffered, RSP has a smaller cache footprint and fewer memory instructions than the DMA-based approach.

FFT Performance. The performance of each of these implementations is measured by executing a 2D FFT on a 256×256 matrix using varying numbers of processes[3]. The results are shown in Figure 3(a) with performance normalized to that of the shared memory implementation. From the figure one can see that the shared memory implementation is fastest for small numbers of processes. With few processes, data is never far away, so locality does not have a pronounced effect on performance. Therefore, the overhead incurred in the RSP and DMA implementations to ensure locality results in a performance loss compared to shared memory. In the worst case this performance loss is as much as 50% for DMA-base approaches, while the RSP implementation is only about 17% slower than shared memory.

As the number of processes grows, locality becomes in-

[3]Although not required, in all the following experiments, eight cores are reserved for the operating system and the maximum number of cores available to the application is fifty-six. Additionally, the FFT performance of all implementations using 32 and 56 cores is measured using the cycle-accurate simulator. The simulator is used to avoid an interaction with the page table that distorts performance in favor of remote store programming.

creasingly important for performance and both the DMA and RSP approaches out-perform shared memory. With sixteen processes, RSP just outperforms shared memory, while DMA lags behind. However, with 32 and 64 processes, the benefit of locality in both the remote store and DMA implementations starts to become clear. With this many processes, a cache miss in the shared memory FFT 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 case of both RSP and DMA, loads do not generate coherence traffic to remote cores. The emphasis on locality results in greater performance for both RSP and DMA implementations of the FFT when compared to shared memory using large numbers of processes.

While both the DMA and RSP implementations focus on locality for performance, the RSP approach is much more efficient than the DMA based one. The RSP implementation is over 64% faster than the best DMA based approach even though it generates more communication. The advantages of the RSP approach are three-fold. First, the RSP FFT does not require a buffer to store temporary data on the producer side, resulting in fewer memory accesses and fewer cache misses. Second, because the RSP approach communicates at an extremely fine granularity, it can almost completely hide the latency of communication. Third, in the remote store approach the matrix transpose is performed as part of the communication, while in the DMA-based approach transposition requires a separate step.

The next section demonstrates that the RSP approach provides performance not only on kernel applications like the FFT, but also on large scale applications.

4.3. H.264 encoding

The viability of remote store programming in a large scale application is demonstrated by experimenting with several implementations of a Baseline profile H.264 encoder for high-definition, 720p video [14]. This H.264 implementation attempts to minimize the average encoding latency of each frame 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. As with the FFT, the differences in implementations is limited solely to differences in data movement and the same algorithm is used for all three implementations.

Cache-coherent shared memory H.264 Encoder. In this implementation, all video frames are allocated in globally shared memory. When a process needs data produced by a neighbor, it reads that data from the global address space. In this implementation, processes do not need to know who produced which data item, they simply need to know when it is safe to read data. As with the shared memory FFT, data in the H.264 encoder is read and written without regard to the physical location of any of the data or the corresponding coherence directories. As illustrated in Figure 4(a), the frame data is shared through the global address space.

DMA-based H.264 Encoder. In this implementation, each process allocates local, homed memory to hold its assigned region of the frame and additional local memory to hold the values that will be produced by neighboring processes. As each process produces data it is stored locally and later copied to the homed memory of each process that needs to access it. These copies are performed using DMA and DMA transactions are scheduled to maximize the overlap of communication and computation. As illustrated in Figure 4(b), a producer process stores frame data in a local buffer and then uses the core-to-core DMA engine to copy that buffer to multiple consumers.

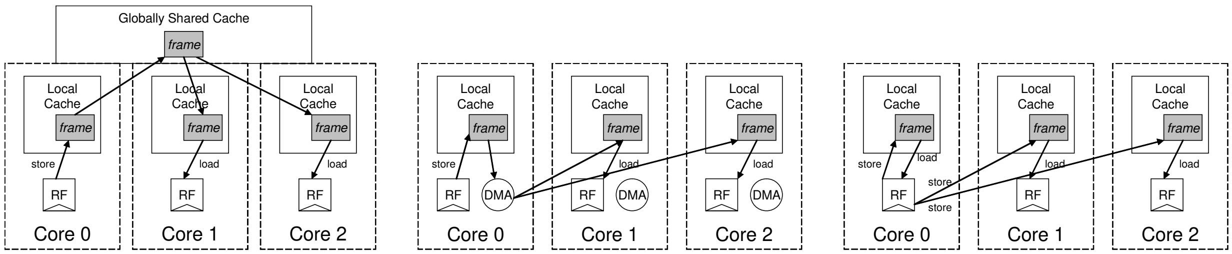
RSP H.264 Encoder. The RSP implementation is similar to the DMA-based implementation. Each process again allocates memory to hold its local data and the data produced by neighbors and each process copies data explicitly to the neighbors that need it. The difference is that the RSP implementation performs the copy by storing directly to the remote memory rather than using DMA as illustrated in Figure 4(c).

H.264 Performance. The performance of each of these implementations is measured by executing the H.264 encoder using various numbers of processes[4]. Figure 3(b) presents the performance of the H.264 encoder relative to shared memory. Again, the impact of locality on performance is clear. In this case, both the DMA and RSP based approaches outperform shared memory from the beginning. While the difference in performance is slight for two and four process implementations, the eight process implementation already shows a 50% improvement for the DMA and RSP approaches. For larger numbers of processes the performance benefit is even more pronounced and both the DMA and RSP implementations achieve almost 4.5 times the performance of shared memory using 40 processes.

As noted above, both the RSP and DMA implementations of the H.264 encoder require extra copy operations to update neighboring processes. These copies are not necessary in the shared memory implementation. This copied data is written once and read repeatedly. The results indicate that the overhead of these additional copy operations is more than compensated for by the resulting locality of the reads.

In the case of the H.264 encoder, the DMA and RSP implementations achieve almost the same performance. For

[4]None of the three implementations benefit from additional processes beyond forty.



(a) Shared Memory (b) DMA (c) RSP

Figure 4. Communication patterns in the H264 encoder. The figure shows how the frame data is sent from a producer on core 0 to multiple consumers on cores 1 and 2. Shared memory transfers data through the global address space. DMA copies data from the producer to each of the consumers. RSP buffers data locally and then copies to the consumers by issuing store instructions targeting the receiver's local memory.

the FFT, RSP is faster than DMA because it requires less buffering and allows greater overlap in communication and computation. However, in the H.264 encoder each data item is used by multiple consumers. This usage pattern requires that data be buffered on the producer side so that it can be copied to multiple locations. Both the DMA and RSP implementations need the same amount of buffering in this case. Also, in the H.264 encoder there is a much higher ratio of computation to communication so it is easier to overlap the two.

Given the need for buffering and the ratio of computation to communication in the H.264 encoder it is not surprising that both the DMA and RSP implementations achieve similar performance. However, the RSP implementation does not require hardware support for a DMA engine making it a much cheaper micro-architectural alternative for the same performance level. If a multicore implementing RSP provided support for a multicast remote store, it would be possible to implement the H.264 encoder with multicast RSP and avoid the extra copying. Such an implementation may outperform DMA.

4.4. Speedup of remote store applications

In addition to the applications described above, four other applications have been implemented on the TILEPro64 using the remote store paradigm. This section presents the speedup results of all six of these applications.

The first two benchmarks are the 2D FFT and the H.264 encoder, both of which are discussed in the previous section. Benchmark three is radix sort [9], which is measured sorting an array of $N = 1048576$ using P processes. Benchmark four is one-dimensional Jacobi relaxation performed on an array of length $N = 57344$. Benchmark five is full-search motion estimation for video encoding [12]. This benchmark divides 720×480 frames of pixels into 16×16 regions called macroblocks, and then attempts to find the most similar 16×16 region in a reference frame. Benchmark five is the BDTI Communications Benchmark (OFDM) [2]. This benchmark implements a simplified multi-channel orthogonal frequency division multiplexing wireless receiver.

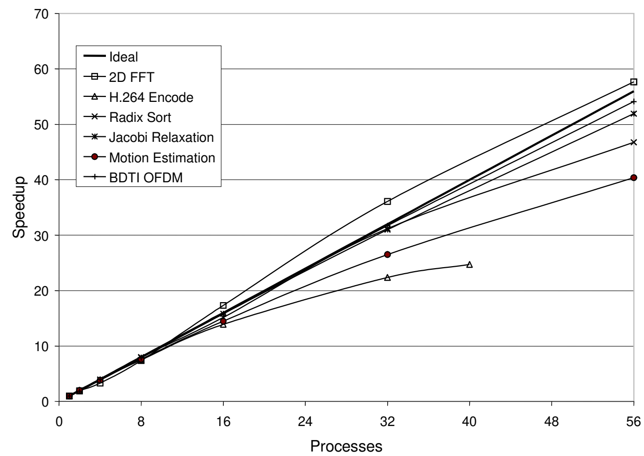


Figure 5. Speedup of remote store applications.

Figure 5 shows the speedup achieved by RSP applications for all six benchmarks. Again, the performance improvement is due to the advantages of RSP in emphasizing (1) locality of reference in minimizing load latency, and (2) fine-grained communication that allows overlapping computation and communication. For all six of these benchmarks, load latency is critical for performance and the RSP model guarantees that loads target local memory ensuring minimal load latency. Additionally, as the RSP paradigm results in fine-grained communication, there is ample opportunity to overlap computation and communication. For example, the FFT stores a value to remote memory and then immediately begins computing the next value without waiting for acknowledgement or issuing other communication instructions.

5. Related work

This section compares remote store programming to previous work on programming models for parallel architectures.

The two best known paradigms for parallel computing are shared memory (with common implementations in the POSIX threads library [5] and Open MP [7]) and message passing (with common implementations in PVM [22] and MPI [21]). In shared memory programs, processes communicate by using standard load and store instructions to read and write data in globally accessible memory. In message passing programs, processes communicate through a two-sided exchange which transfers data between processes private address spaces. Shared memory is generally considered easier to use because of the familiar communication mechanism, while message passing is generally considered higher performance because it requires that programmers carefully consider data layout and locality in their programs.

The partitioned global address space (PGAS) model combines the familiarity associated with shared memory programming and the focus on locality of reference associated with message passing programming. The PGAS model has been implemented in both the SHMEM library [19] and several languages including Unified Parallel C [8], Titanium [28], and Co-Array Fortran (CAF) [16]. The model and its implementations target multichip parallel computers with physically distributed, non-uniform memory access (NUMA) memory architectures like clusters and supercomputers. In these architectures, processors have their own DRAM, and multiple processors (and DRAMs) are connected by a network.

Although the PGAS programming model conceptually uses load and store instructions as communication primitives, an individual load or store to a remote DRAM is expensive. PGAS implementations 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. (Although, as discussed above in the FFT implementation, sometimes the desire to overlap communication conflicts with the desire to minimize total communication.) These optimization techniques, in turn, effect 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.

The RSP model, however, targets multicore NUMA architectures. These architectures typically have many processors connected by a powerful on-chip network. The on-chip network makes it possible for processors to transfer data from cache to cache (e.g. IBM Cell [18]), registers to cache (e.g. TILEPro64 as described above [27]), or even from registers to registers (e.g. the register-mapped networks of the Raw architecture [23]). The remote store programming model is specifically designed to support fine-grained communication on these types of multicore architectures. As discussed in Section 4, such communication can produce better performance than the bulk communication common to multichip PGAS implementations. Furthermore, this performance is achievable with less hardware support.

Like RSP, the virtual memory mapped communication (VMMC) model [11] allows processes to transfer data directly between the producer's and consumer's virtual address space. In fact, the "automatic update" option of VMMC is semantically similar to RSP in the case where data has one producer and one consumer. In both cases the producer writes to a memory region and these writes show up in the consumer's memory. The difference between RSP and VMMC is the hardware mechanisms used by each. VMMC is implemented by writing data to memory on the producer. The consumer snoops the memory bus, sees these writes, and stores its own copy of the data. 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.

Several multichip hardware architectures provide support for the remote store programming model. These include distributed shared memory architectures like the Cray T3D [10], T3E [20] and the TMC CM5 [25]. All three of these processors are scalable multichip computers that support shared memory programming using physically distributed memories. While these machines provide most of the hardware required to support RSP, it would likely result in inefficient application code. For example, while the T3E supports single word transfers to remote memory, network bandwidth is optimized when when transfers are 16 KB or larger [20]. The necessity of transferring large data buffers for performance makes PGAS interfaces, which explicitly support such transfers through *put* and *get* operations, a better match for the architecture. RSP is better suited to distributed shared memory multicores where the powerful on chip networks can stream individual words from one core's registers to another's cache.

6. Conclusion

The remote store programming 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 a two-dimensional FFT and an H.264 video encoder on a multicore exhibit equal or better performance than common multichip techniques like shared memory and DMA.

Although this paper distinguishes between the RSP and PGAS models, note that these two models are not mutually exclusive. Many PGAS interfaces are actually implemented as languages and it would be possible for the back-end of a PGAS (or any other) parallelizing compiler to target a multicore RSP implementation for high performance.

Consider two possible uses of the RSP model in emerging multicore architectures. First, RSP could be used to augment directory-based cache-coherence schemes for multicores with many processors. Standard shared memory techniques could 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. Second, RSP could be used as a convenient and efficient programming model on multicore DSPs. These architectures are designed for the same applications that map well to the RSP paradigm: regular, numerical computation with high performance requirements. RSP may be a more convenient and less costly interface for these processors than the DMA-based interfaces that dominate multichip DSP architectures.

Acknowledgements

This work is supported by Quanta Computer and Tilera Corporation.

References

[1] I. T. Association. http://www.infinibandta.org.

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

[3] C. Bell, D. Bonachea, R. Nishtala, and K. Yelick. Optimizing bandwidth limited problems using one-sided communication and overlap. *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 10 pp.–, April 2006.

[4] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W. king Su. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, 15:29–36, 1995.

[5] D. R. Butenhof. *Programming with POSIX threads*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[6] W. Carlson, T. El-Ghazawi, R. Numric, and K. Yelick. Programming with the pgas model. In *IEEE/ACM SC2003*, 2003.

[7] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon. *Parallel programming in OpenMP*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[8] S. Chauvin, P. Saha, F. Cantonnet, S. Annareddy, and T. El-Ghazawi. *UPC Manual*. May 2005. http://upc.gwu.edu/downloads/Manual-1.2.pdf.

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition*. The MIT Press, 2001.

[10] Cray research, inc., CRAY T3D system architecture overview, 1993.

[11] C. Dubnicki, L. Iftode, E. Felten, and K. Li. Software support for virtual memory-mapped communication. pages 372–381, Apr 1996.

[12] B. Furht and B. Furht. *Motion Estimation Algorithms for Video Compression*. Kluwer Academic Publishers, Norwell, MA, USA, 1996. Performed By-Joshua Greenblatt.

[13] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Mateo, CA.

[14] ITU-T. H.264: Advanced video coding for generic audiovisual services.

[15] C. V. Loan. *Computational Frameworks for the Fast Fourier Transform*. SIAM, Philadelphia, 1992.

[16] R. W. Numrich and J. Reid. Co-array fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, 1998.

[17] F. Petrini, W. chun Feng, A. Hoisie, S. Coll, and E. Frachtenberg. The quadrics network: high-performance clustering technology. *Micro, IEEE*, 22(1):46–57, Jan/Feb 2002.

[18] D. Pham, S. Asano, M. Bollinger, M. N. Day, H. P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Riley, D. Shippy, D. Stasiak, M. Suzuoki, M. Wang, J. Warnock, S. Weitzel, D. Wendel, T. Yamazaki, and K. Yazawa. The Design and Implementation of a First-Generation CELL Processor. In *Proceedings of the IEEE International Solid-State Circuits Conference*, Feb. 2005.

[19] Quadrics. *SHMEM Programming Manual*. Quadrics Supercomputers World Ltd, Bristol, UK, 2001.

[20] S. L. Scott. Synchronization and communication in the t3e multiprocessor. In *ASPLOS-VII: Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, pages 26–36, New York, NY, USA, 1996. ACM.

[21] M. Snir and S. Otto. *MPI-The Complete Reference: The MPI Core*. MIT Press, Cambridge, MA, USA, 1998.

[22] V. S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2:315–339, 1990.

[23] M. B. Taylor, W. Lee, J. Miller, D. Wentzlaff, I. Bratt, B. Greenwald, H. Hoffmann, P. Johnson, J. Kim, J. Psota, A. Saraf, N. Shnidman, V. Strumpen, S. Amarasinghe, and A. Agarwal. Evaluation of the Raw Microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams. In *International Symposium on Computer Architecture*, June 2004.

[24] TMS320C6474 multicore digital signal processor, Oct. 2008. http://www.ti.com/lit/gpn/tms320c6474.

[25] Connection machine cm-5 technical summary, thinking machines corporation,, 1992.

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

[27] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J. F. Brown III, and A. Agarwal. On-chip interconnection architecture of the tile processor. *IEEE Micro*, 27(5):15–31, 2007.

[28] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A high-performance java dialect. In *In ACM*, pages 10–11, 1998.