Enabling Parallelization via a Reconfigurable Chip Multiprocessor

Matthew A. Watkins
Cornell University
Computer Systems Laboratory

David H. Albonesi
Cornell University
Computer Systems Laboratory

ABSTRACT

While reconfigurable computing has traditionally involved attaching a reconfigurable fabric to a single processor core, the prospect of large-scale CMPs calls for a reevaluation of reconfigurable computing from the perspective of multicore architectures. We present ReMAPP, a reconfigurable architecture geared towards application acceleration and parallelization. In ReMAPP, parallel threads share a common reconfigurable fabric which can be configured for individual thread computation or fine-grained communication with integrated computation. The architecture supports both fine-grained barrier synchronization and fine-grained point-to-point communication for pipeline parallelization.

The combination of communication and configurable computation within ReMAPP provides the unique ability to perform customized computation while data is transferred between cores, and to execute custom global functions after barrier synchronization. We demonstrate that ReMAPP achieves significantly higher performance and energy efficiency compared to hard-wired communication-only mechanisms, and over what can ideally be achieved by allocating the fabric area to more cores.

1. INTRODUCTION

Reconfigurable computing has traditionally involved attaching a reconfigurable fabric to a single conventional processor core. However, the prospect of large-scale chip multiprocessors (CMPs) with tens to hundreds of cores on a die calls for a reexamination of reconfigurable computing from the perspective of multicore architectures. This paper presents ReMAPP (Reconfigurable Multicore Architecture for Parallel Processing), a general-purpose reconfigurable architecture that accelerates both sequential and parallel workloads and allows parallelization of otherwise sequential applications in an area- and power-efficient manner.

While past reconfigurable architectures have been shown to significantly outperform general purpose architectures for certain application classes, this has come at high area and power costs relative to the overall performance achieved across a broad set of applications. Large-scale CMPs, however, are likely to be heterogeneous in nature, with different areas of the die dedicated to accelerating particular types of applications. Within this context, CMPs offer a more cost-effective way to incorporate reconfigurable fabrics into commodity microprocessors for two reasons. First, the die area dedicated to reconfigurable fabrics may be sized in proportion to the expected proportion of applications that will benefit. Second, the area and power costs of the fabric may be amortized by sharing the fabric among multiple cores, thereby forming a *cluster* of cores+fabric. Through intelligent fabric management, the fabric utilization may be increased, and the overall fabric area and power costs reduced, while achieving nearly the same performance as providing each core with its own, much larger, private fabric [28].

Sharing the reconfigurable fabric among multiple cores creates optimization opportunities not possible with per-core private fabrics. In particular, shared fabric clusters – in addition to increasing fabric area and power efficiency – can be organized on-the-fly in multiple ways to accelerate and help parallelize applications. Figure 1 provides a simplified view of the three ways that the ReMAPP architecture is dynamically organized to these ends. Each figure shows four cores sharing a single reconfigurable fabric. Figure 1(a) depicts four threads (either from the same or different applications), each of which is independently performing a function within the fabric. In Figure 1(b), the fabric is being used for two instances of fine-grain producer-consumer communication with integrated customized computation. Finally, Figure 1(c) depicts four threads synchronizing at a barrier within the fabric with a global function, e.g., a global minimum, computed in the fabric after the synchronization point. The later two organizations allow the parallelization of applications that would not be possible with traditional software techniques due to the fine-granularity of the communication and allow the acceleration of sequential sections of parallel applications.

Having outlined the general principles of ReMAPP, the remainder of this paper presents the architecture of ReMAPP in detail and analyzes its performance and energy efficiency. As the first proposed reconfigurable architecture for accelerating computation and communication, ReMAPP integrates the following features:

- A tightly integrated, row-based reconfigurable fabric to accelerate the computation of threads operating independently (Section 2.1);
- Mechanisms to temporally and spatially share the fabric between multiple cores (Section 2.1);
- Fine-grained inter-core data communication with custom computation (Section 2.2.1);
- Fine-grained barrier synchronization with custom computation (Section 2.2.2).

Unlike previous proposals, ReMAPP supports multiple communication models and also provides the ability to perform customized computation on communicated data. The later provides optimization opportunities not possible with previous communication-only options.

2. REMAPP ARCHITECTURE

ReMAPP pairs a specially designed *Specialized Programmable Logic (SPL)* fabric with multiple cores of a CMP. An example future CMP with integrated SPL is depicted in Figure 2(a). The figure shows a 20 core CMP[1] with two ReMAPP clusters on the

[1] Although relative sizes of the cores and fabric are accurate, this is not intended to represent an actual floorplan.

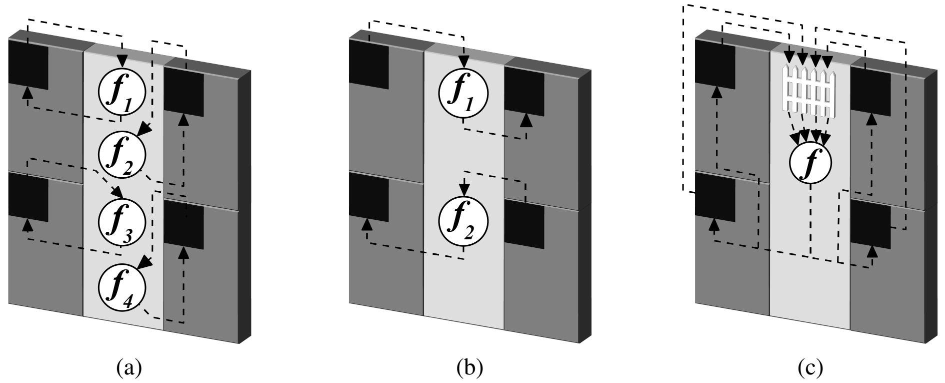


Figure 1: Shared SPL being used for (a) individual computation, (b) producer-consumer communication with computation, and (c) barrier synchronization with computation.

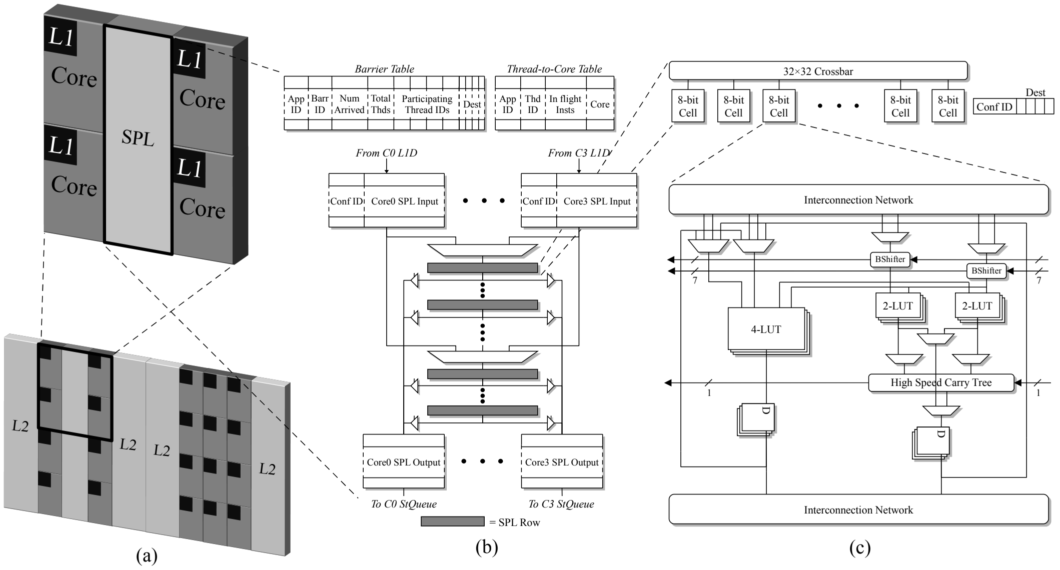


Figure 2: ReMAPP integration in a CMP. (a) Depiction of overall chip, with two ReMAPP clusters and one conventional cluster, and blow-up of one ReMAPP cluster, (b) four-way shared SPL including tables required for communication, and (c) design of SPL row.

left. Each cluster consists of four single issue out-of-order processor cores sharing a SPL fabric, which is shown at a high level in Figure 2(b). The fabric is temporally shared in a round-robin fashion among the cores in the same cluster and can be spatially partitioned to reduce contention among the threads. Contention is further reduced by limiting the degree of fabric sharing, which also limits the maximum wire delay. In this particular example, the proportion of applications that benefit from the fabric is such that two shared fabric clusters are implemented. In a large-scale heterogeneous CMP with many tens or hundreds of cores, there may be several ReMAPP clusters as well as many other different cluster types, such as the traditional many-core cluster shown on the right hand side of Figure 2(a), on the die.

2.1 ReMAPP Organization

The computational substrate of ReMAPP is the highly pipelined, row-based SPL of [28]. The SPL is composed of 24 rows, in which each row contains 16 cells and each cell computes 8 bits of data. Figure 2(c) shows the row and cell designs. The major cell components are a main 4-input look-up table (4-LUT), a set of 2 2-LUTs plus a fast carry chain to compute carry bits (or other logic functions if carry calculation is not needed), barrel shifters to properly align data as necessary, flip-flops to store results of computations, and an interconnect network between each row. Within a cell, the same operation is performed an all 8 bits. These 8-bit cells are arranged in a row to form a 16×8-bit row. Each cell in a row can perform a different operation on its set of inputs and 24 of these rows are grouped together to form the overall SPL fabric. The SPL is clocked at a fixed 500 MHz. This is one-quarter the 2 GHz core frequency (the same as the Pentium Core2 Duo [17] and the AMD X2 Dual-Core [1], both of which are implemented in the same 65nm technology assumed for ReMAPP) and allows each row to complete the longest possible computation in a single cycle. Table 1 shows the relative area and power consumption of the SPL and associated single-issue cores.

	SPL Rows	Total Area	Peak Dyn. Power	Total Leak. Power
Four Cores	N/A	1.00	1.00	1.00
4-way Shared SPL	24	0.51	0.14	0.67

Table 1: Relative area and power of four single-issue out-of-order cores and four-way shared ReMAPP fabric.

The row-based nature of the fabric allows hardware requirements to be indicated by the number of rows needed to implement a function. If the number of rows required by a function exceeds the physical number on chip, the function can be virtualized over the

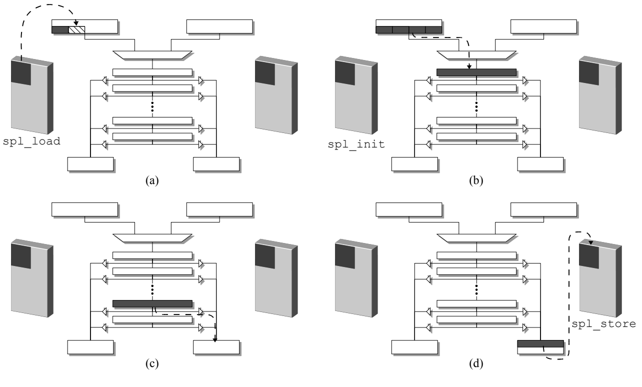


Figure 3: Walk through of intercore communication with integrated computation.

fabric [15]. Virtualization uses the same physical row to execute multiple virtual rows of the function. This comes at a possible loss in throughput but guarantees that all functions can be executed, even if fewer rows are available than originally anticipated.

The SPL is integrated with the processor core as a reconfigurable functional unit and interfaces to the memory system via a queue-based decoupled architecture as shown in Figure 2(b). Special SPL load instructions place values into the input queue at a particular data alignment. In addition to storing the input data itself, each byte in the input queue includes a valid flag indicating that the position contains data loaded for the current instruction. These flags are needed for barrier synchronization when input data from multiple queues is merged. For output, the SPL similarly writes to a local output queue that is then written out to the Store Queue using a special SPL store instruction.

The SPL supports both spatial partitioning, in which the fabric is divided into multiple virtual clusters and each core accesses a single virtual cluster, and temporal sharing, in which a partition is shared among multiple cores in a time multiplexed fashion. Spatial partitioning reduces contention from sharing threads, but also reduces the amount of resources available to each core, possibly leading to degraded throughput due to increased virtualization. The fabric can be divided in up to 4 virtual clusters and a simple round-robin schedulers selects the instruction to issue each shared virtual cluster. Figure 2(b) shows the additional multiplexers and tristate drivers necessary to support both forms of sharing.

2.2 Support for Fine-Grained Communication with Computation

At a high level, communication requires the exchange of information between threads, be it a notification that a thread has arrived at a barrier or a producing thread passing results to a consuming thread. ReMAPP facilitates fine-grain communication among threads sharing the fabric, creating new opportunities for parallelization that are too costly using conventional software-based methods. Moreover, the ability to perform computation within the fabric during communication provides additional benefits over communication-only mechanisms.

2.2.1 Fine-Grained Interthread Communication+Computation

Fine-grained interthread communication enables threads to communicate with each other much more frequently than would be possible using the traditional memory system. Such fine-grained communication is typically targeted at pipelined/streaming applications [7, 22]. To perform this type of communication, a queue is established between the two communicating threads. The producing thread places data into the queue and the consuming thread reads data from the queue. Unless the queue is full/empty, the two threads can continue to produce/consume data without concern for how the other thread is progressing.

Since the fabric is shared between multiple cores, sending data to a different core simply requires sending the fabric output to the output queue of the consuming core. The input and output queues provide queuing slots and the pipelined fabric serves as both a computational substrate and as additional *on-demand* queue slots.

Figure 3 details the steps involved in interthread communication with custom computation. First, the producing thread loads data into its input queue (Figure 3(a)). Once all of the necessary data is loaded, the producer issues an SPL instruction (Figure 3(b)). The data progresses through the SPL to perform the computation programmed into the fabric. Once any computation is complete, the results are bypassed to the output queue of the consuming core (Figure 3(c)). Finally, the consuming core stores the data from the queue to memory (i.e., store queue) (Figure 3(d)).

Two features ease intercore communication via ReMAPP. First, in order to fully utilize the queuing capacity of the fabric, instructions that have completed their computation but cannot yet be allocated an output queue slot continue to progress through the rows of the fabric, simply passing their output data through to the next row. This continues until either an output queue slot becomes available, at which point the data is immediately written to the output queue (bypassing any remaining rows in the fabric) or the instruction reaches the end of the fabric, at which point it stalls. When the fabric is stalled, instructions immediately following the stalled instruction stall as well. Bubbles in the SPL pipeline, however, are allowed to collapse and so some progress may continue to be made even if the head instruction is stalled. If the entire fabric is full, then the producing thread will stall if it attempts to issue additional SPL instructions.

The second feature is a small table to maintain a mapping of threads to cores in order to virtualize the selection of the destination core (see Thread-to-Core Table in Figure 2(b)). When an SPL instruction is issued, it obtains the core currently assigned to its destination thread (which may be either itself or another thread) from the table and stores its results to the appropriate output queue upon completion. In our proposed 4-way shared fabric, each table has four entries. Each entry contains the thread and application ID currently running on that core as well as a count of the number of in flight instructions destined for that core. Assuming a limit of 256 thread and application IDs and a maximum of 24 in flight instructions (as the fabric has 24 rows), each per-SPL table requires a 11.5B CAM (16 bits for IDs, 5 bits for number of in flight instructions, and 2 bits for hard coded core ID).

A side benefit of this table based approach is that instructions will not issue to the fabric if the destination thread is not available (not present in the table). This prevents the producing thread from filling up the fabric if the consumer is not present, which could impact other threads sharing the fabric. If both threads are present but not well balanced, it is possible the fabric could still be full most of the time. However, assuming that the program is even remotely well written, the consumer would still be consuming values, even if at a slow rate, and, because of the SPL's round robin issue policy, other threads would continue to be able to utilize the fabric.

2.2.2 Barrier Synchronization+Computation

Barriers are one of the most common synchronization operations. However, with a typical memory-based implementation, the overhead of executing a barrier can be significant, especially as the

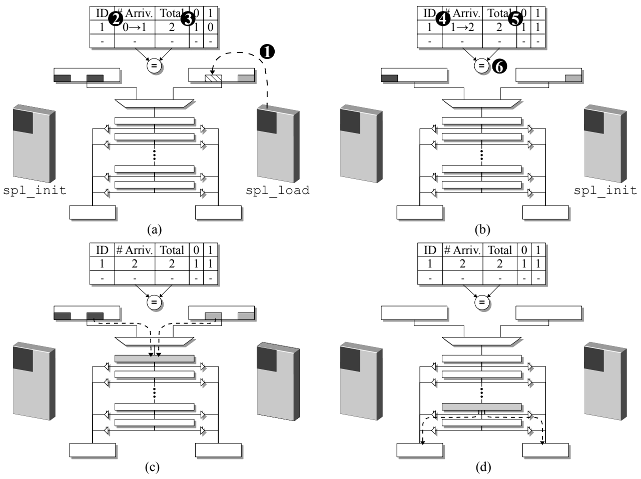


Figure 4: Walk through of barrier synchronization with integrated computation.

number of threads increases. This overhead prevents the use of barriers at fine granularities. Various proposals [2, 4, 23, 25] have suggested dedicated mechanisms to reduce this overhead, thereby allowing parallelization of applications that would not otherwise be possible. In cases where a barrier is followed by a serial function that is performed by one of the threads and the output communicated to all participating threads, ReMAPP may directly synthesize the function into the fabric with the output communicated to the participants' output queues.

To implement barriers in ReMAPP, SPL barrier instructions (indicated by a flag in the ReMAPP function configuration), must not be allowed to issue to the fabric until all participating cores have arrived at the barrier. To achieve this, each core participating in the barrier loads some value(s) into its SPL input queue (Figure 4(a)❶). Once the loads from all of the cores have reached the head of their respective input queues and all threads have indicated arrival at the barrier by executing a SPL initiate instruction, an instruction is issued to the fabric by the ReMAPP controller, and the loaded values from each core are passed into the fabric (Figure 4(c)). The valid bits associated with every byte in the input queues are used to determine which values from each core should be loaded into the fabric. The global function programmed into the fabric is performed, the results are placed into the output queue of each participating processor (Figure 4(d)), and the processor stores the data as appropriate. A memory fence is executed following the stores to ensure that no subsequent memory operations are performed until the barrier is complete.

To determine that all threads have arrived at the barrier, each SPL cluster maintains a table with information related to each active barrier. Each table (see Barrier Table in Figure 2(b)) contains as many entries as cores attached to a ReMAPP cluster, as each could be participating in a different barrier. The table keeps track of the total number of threads, the number of arrived threads, and the number of cores that are participating in the barrier. The number of arrived thread and participating cores are updated each time a thread arrives (Figure 4 ❷, ❸, ❹, and ❺) and the total and arrived threads are compared to determine when to issue an instruction (❻).

In a system with multiple ReMAPP clusters, each cluster communicates updates on the number of arrived threads with all other clusters (even though other clusters may not have participating threads). An alternative is to have clusters only monitor those barriers in which they have threads actively participating. This, however, requires that clusters obtain the number of currently arrived threads from another cluster each time a locally new barrier arrives, which increases the complexity of the intercluster network. Since the table is localized and is small in either case, whereas the increase in interconnect complexity has global impact, we choose to track all active barriers in every cluster to reduce interconnect overhead at the cost of an increase in table size. A dedicated bus communicates barrier updates among clusters. The bus transmits the barrier ID as well as the associated application ID (as different applications might use the same barrier ID). With a limit of 256 IDs, the shared bus requires only 12 data lines plus control. Each table entry requires 8 bytes: 16 bits for IDs; 4 for number of arrived threads; 4 for total number of threads; 4 to indicate participating cores; 32 for participating thread IDs; and 4 to indicate if each participating thread is currently active. In a four cluster (16 core) system, this requires a 128B table for each cluster.

All threads participating in a barrier must be actively running in order for all input data to be available. Each table entry maintains a list of the IDs of the local threads that are participating in the barrier as well as a bit indicating if they are actively running. If a barrier is ready to be released but not all participating threads are active, the ReMAPP controller triggers an exception to switch the missing threads back in. Once all threads are available, the barrier can proceed. Since ReMAPP barriers are primarily intended for fine grain synchronization, switching out a thread that arrives early should be avoided in any event for performance reasons.

3. COMMUNICATION EXAMPLES

We propose using the SPL to perform both fine-grained interthread communication and fine-grained barrier synchronization. In this section we show example applications that benefit not only from the enhanced communication, but also receive additional benefits due to the computational power of ReMAPP that could not be achieved with communication alone.

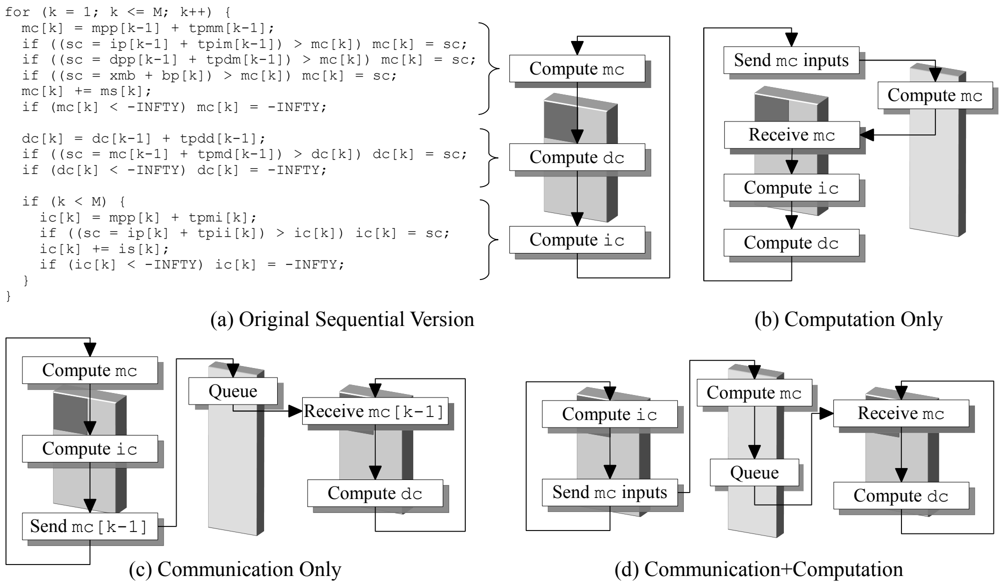
3.1 Interthread Communication+Computation Example

To illustrate interthread communication, we show an example parallelization of the SPEC2006 application *456.hmmer*. We optimize the inner loop of the `P7Viterbi` function, which implements the dynamic programming Viterbi algorithm. The original code for the optimized section is shown in Figure 5(a) along with a flow chart summarizing the computation being performed. This high level description will be used to show how the function is optimized for computation alone, communication alone, and for the computation+communication case.

We first look at how the SPL can be used to accelerate a portion of the computation, specifically the calculation of `mc`. As shown in Figure 5(b), the core loads the input values needed to compute `mc` into the fabric, the SPL computes the value of `mc`, and the core receives the result. After receiving `mc`, the core computes the values of `dc` and `ic` and repeats the loop. Figure 6 shows the general functionality performed within each row of the SPL for the optimized section.

The next implementation creates a producer/consumer thread pair that uses the SPL solely for communication (Figure 5(c)). The producer thread is responsible for calculating the values of `mc` and `ic` and sending the value of `mc` from the previous iteration to the consumer through the SPL. The consumer receives this value and uses it to compute `dc`.

Finally, Figure 5(d) shows how computation and communication can be integrated in the SPL. The producer thread computes `ic` and loads the inputs needed for `mc`. The SPL computes the value of `mc`



```
for (k = 1; k <= M; k++) {
  mc[k] = mpp[k-1] + tpmm[k-1];
  if ((sc = ip[k-1] + tpim[k-1]) > mc[k]) mc[k] = sc;
  if ((sc = dpp[k-1] + tpdm[k-1]) > mc[k]) mc[k] = sc;
  if ((sc = xmb + bp[k]) > mc[k]) mc[k] = sc;
  mc[k] += ms[k];
  if (mc[k] < -INFTY) mc[k] = -INFTY;

  dc[k] = dc[k-1] + tpdd[k-1];
  if ((sc = mc[k-1] + tpmd[k-1]) > dc[k]) dc[k] = sc;
  if (dc[k] < -INFTY) dc[k] = -INFTY;

  if (k < M) {
    ic[k] = mpp[k] + tpmi[k];
    if ((sc = ip[k] + tpii[k]) > ic[k]) ic[k] = sc;
    ic[k] += is[k];
    if (ic[k] < -INFTY) ic[k] = -INFTY;
  }
}
```

(a) Original Sequential Version

(b) Computation Only

(c) Communication Only

(d) Communication+Computation

Figure 5: Parallelization of SPEC 2006 *456.hmmer* `P7Viterbi`.

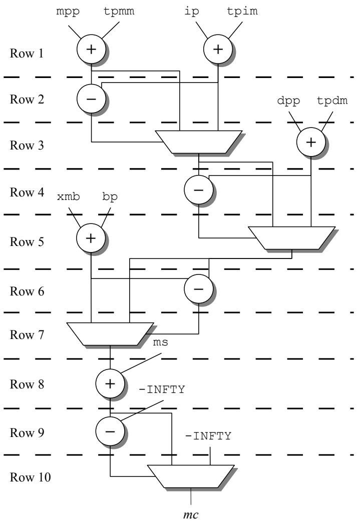


Figure 6: Mapping of `mc` calculation to SPL.

and sends it to the consumer. The consumer receives this value and uses the value of `mc` from the previous iteration to compute `dc`. Computing `mc` in the fabric reduces the amount of work for the producer, which better balances the threads and further improves the performance of the parallelization (see Section 5.1).

3.2 Barrier Synchronization+Computation Example

To show the operation of ReMAPP barrier synchronization, we consider a parallel version of Dijkstra's Shortest Path Algorithm. Parallel versions of Dijkstra's Algorithm have previously been proposed. These algorithms, however, tend to provide limited or no speedups for small to moderate graph sizes. By using ReMAPP to perform the barrier synchronization, we can improve the synchronization while also using the fabric to perform computation during the barriers to further improve performance.

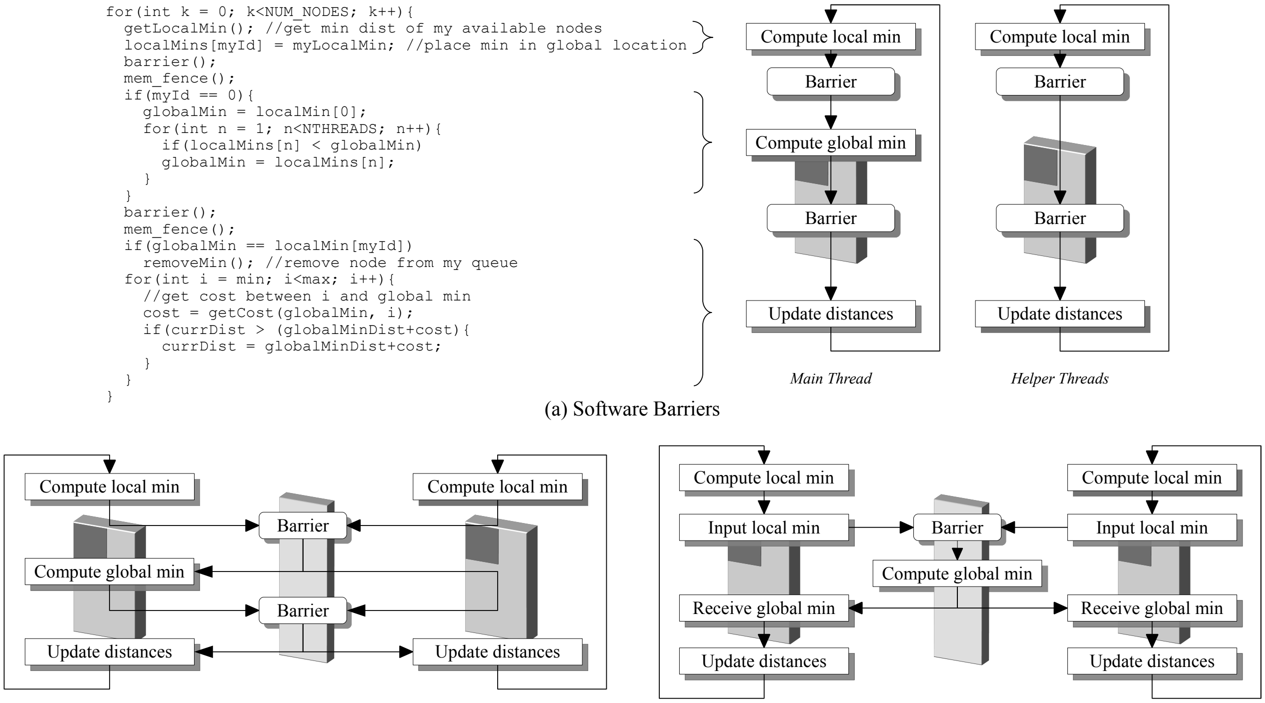
In the parallel version of Dijkstra's Algorithm, each thread is given a portion of the entire graph to maintain. Figure 7(a) shows pseudocode of the basic parallel algorithm and the high level flow of the main and helper threads. The code consists of three sections, delineated by code before, between, and after the two barriers. In the first section, each thread determines the minimum value of all unvisited nodes among its subset and places this value in a global location. In the next section, the main thread computes the global minimum from these local minimum values and makes this value globally available. Finally, each thread reads the global minimum and updates the distances for all of its nodes.

The first optimization that can be made is to replace the software barriers with ReMAPP barriers, as shown in Figure 7(b). As with previous dedicated barrier techniques [2, 25], replacing the software barriers with ReMAPP barriers provides significant performance improvements. Performance can be further improved beyond that possible with previous techniques by using the computational power of ReMAPP to compute the global minimum within the fabric. Figure 7(c) shows this optimization for the case where all threads share a single ReMAPP cluster. Each thread computes its local minimum as before and then loads this value into the SPL. While performing the barrier, the SPL computes the minimum of the input values. Each participating core receives the global minimum from the SPL and updates the distances for its nodes. Since the SPL outputs the global minimum directly, one of the barriers is eliminated.

If the threads are spread across multiple clusters, the fabric still helps compute the minimum; however, this operation is performed in multiple stages and requires an extra barrier to ensure proper execution. The first stage computes regional minimum values (minimum values of all cores in a single cluster). The second barrier ensures that all clusters have finished storing these results. At the final barrier each cluster loads the regional minimum values and the fabric computes the final global minimum. Despite the extra barrier, performance is still improved over using ReMAPP for communication only (see Section 5.2).

4. EVALUATION METHODOLOGY

We use a modified version of SESC [24] to evaluate our proposed communication schemes. We assume processors implemented in 65 nm technology running at 2.0 GHz with a 1.1V supply voltage. The major architectural parameters are shown in Table 2.



```
for(int k = 0; k<NUM_NODES; k++){
  getLocalMin(); //get min dist of my available nodes
  localMins[myId] = myLocalMin; //place min in global location
  barrier();
  mem_fence();
  if(myId == 0){
    globalMin = localMin[0];
    for(int n = 1; n<NTHREADS; n++){
      if(localMins[n] < globalMin)
      globalMin = localMins[n];
    }
  }
  barrier();
  mem_fence();
  if(globalMin == localMin[myId])
    removeMin(); //remove node from my queue
  for(int i = min; i<max; i++){
    //get cost between i and global min
    cost = getCost(globalMin, i);
    if(currDist > (globalMinDist+cost){
      currDist = globalMinDist+cost;
    }
  }
}
```

(a) Software Barriers

(b) Barrier Only

(c) Barrier+Computation

Figure 7: Parallelization of Dijkstra's Shortest Path Algorithm.

Fetch/Decode/Rename/ Issue/Retire Width	2/2/2/1/1
Branch Predictor	gshare + bimodal
RAS Entries	32
BTB Size	512B
Integer/FP Registers	64/64
Integer/FP Queue Entries	32/16
ROB Entries	64
Int/FP ALUs	1/1
Branch Units	1
LD/ST Units	1
L1 Inst Cache	8kB 2-way, 2-cycle access
L1 Data Cache	8kB 2-way, 2-cycle access
L2 Cache	1MB per core, 10-cycle access
Coherence Protocol	MESI
Main Memory Access Time	100 ns

Table 2: Architecture parameters.

	Function Optimized	% Exec Time
wc	wc	100%
unepic	read_and_huffman_decode	22%
cjpeg	rgb_ycc_convert	21%
adpcm	adpcm_decoder	99%
300.twolf	new_dbox_a	30%
456.hmmer	P7Viterbi	85%
473.astar	regwayobj::makebound2	33%

Table 3: Interthread communication benchmarks.

4.1 Benchmarks

We use a combination of MediaBench and SPEC benchmarks and the unix utility `wc` to evaluate producer-consumer communication. Table 3 shows the benchmarks we use, along with the function we optimized in each, and the percentage of total program execution time consumed by that function. We run four versions of each application: the original source, a single-threaded version that uses the SPL for computation; a dual-threaded version that uses the SPL only for communication; and a dual-threaded version in which computation is performed on the data communicated through the SPL (as in Figure 1(b)). We run the MediaBench and *wc* workloads to completion. For the SPEC workloads, we run a set number of calls to the optimized function such that in the base case at least 500 million instructions of the function are executed.

We use parallel versions of Dijkstra's Algorithm and Livermore Loop 3 in which the loop operates on integers to evaluate ReMAPP barrier synchronization. In Dijkstra's Algorithm, computation is performed during the synchronization operation (as in Figure 1(c)). *LL3* makes use of two ReMAPP modes of operation: performing computation on the data within the loop (Figure 1(a)), and using the SPL to accelerate synchronization between iterations

The SPL is equipped with enough on-chip configuration storage such that, for our workloads, once loaded, a configuration never needs to be reloaded. Our simulations estimate that this initial load time would consume less than 1000 cycles. Given that the SPL functions are executed millions of times over hundreds of millions of cycles, this configuration overhead is insignificant.

4.2 ReMAPP Programming

At the moment we modify our workloads by hand to create the producer/consumer pairs and SPL mappings. Previous work has shown that compilers can produce good mappings for reconfigurable architectures [3, 5, 14, 29] and good partitionings for pipelined applications [16, 19]. We believe our design could leverage this prior art in an actual implementation.

The procedure for identifying and mapping functionality to the SPL is shown in Figure 8. The profiling phase identifies the most important regions of execution. Each of these identified regions is evaluated for SPL suitability in terms of the number of inputs and outputs required, the type of required operations (e.g., integer addition/subtraction, Boolean operations, and conditional selection), and inter-operation dependencies. SPL mappings, similar to that

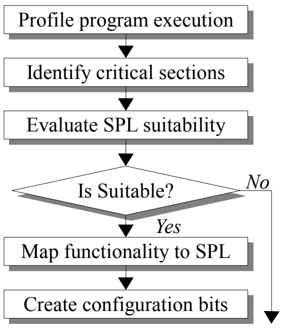

Profile program execution
Identify critical sections
Evaluate SPL suitability
Is Suitable?
No
Yes
Map functionality to SPL
Create configuration bits

Figure 8: Procedure for mapping functions to SPL.

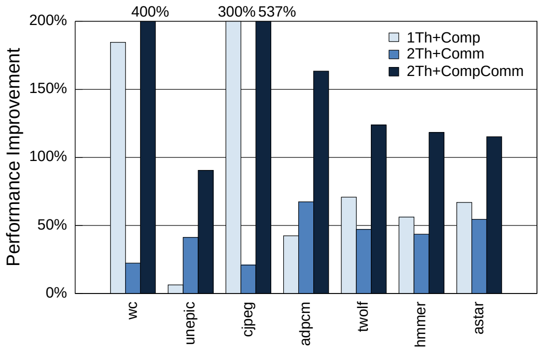


Figure 9: Performance improvement of optimized functions relative to performance of single threaded baseline.

shown in Figure 6, are then generated for the selected regions, and these mappings are transformed into SPL configuration bits.

5. RESULTS

5.1 Interthread Communication+Computation

Figure 9 compares the performance improvements achieved over the single threaded baseline with a single thread using the SPL for computation (*1Th+Comp*), dual threads with the SPL used for communication (*2Th+Comm*), and dual threads with the SPL used for computation and communication of the results to the consumer thread (*2Th+CompComm*). We assume that only half the SPL is available, i.e., the SPL is spatially partitioned so that the other two cores can use the other half for other purposes. Similar to previous work [28], we confirm that using the SPL for computation provides significant performance improvements (104% on average for our workloads), and focus on a comparison of the latter two cases.

Using the SPL for producer-consumer communication alone provides a 42% improvement in performance for the optimized region relative to the single core baseline. By adding the speedups obtained from communication and computation alone, we would expect to achieve an average speedup around 146% when the two techniques are combined. The results for *2Th+CompComm*, however, show an average improvement of 221%. Before delving into the reasons for this behavior in the next section, we first quantitatively compare other options with the ReMAPP approach.

To confirm that similar results could not be achieved using a simple dual-threaded software solution (due to the fine-grain nature of the communication), we also ran the benchmarks with software queues, with and without SPL computation. The software versions experienced more than a 175% slowdown on average relative to the single threaded baseline in both cases.

The *2Th+Comm* option also indicates the performance that would be achieved with previously proposed dedicated producer-consumer communication hardware [7, 22]. Assuming zero hardware cost for these previous proposals, we ideally scale the *2Th+Comm* results by 1.5X to account for replacing the SPL with more powerful cores. The *2Th+CompComm* implementation outperforms this ideally scaled producer-consumer alternative by 59% on average, showing the clear performance benefit of integrated SPL computation and communication.

5.1.1 Benchmark Performance Factors

We analyze the benchmarks to identify the factors that contribute to the performance improvements for combined SPL communication+computation. Primary among these factors is that the combination of SPL computation and communication reduces the amount of time between successive SPL requests relative to using either technique in isolation, often by 2X or more. This increased access rate improves performance by increasing the amount of concurrent processing in the SPL.

Relative to the single threaded case with SPL computation, splitting the application into a producer/consumer pair and performing the computation during the communication means that each core is now responsible for approximately half of the SPL instructions (either the loads or the stores). This reduces the number of instructions that both threads need to process, which can lead to reduced pressure on the ROB and other related structures. This leads to fewer pipeline stalls and therefore better performance. By splitting the threads we can also place sections of code with poor branch or load performance in their own thread to reduce or eliminate their impact on performance. In *unepic*, for example, the consumer is responsible for a section of code with both an unpredictable branch as well as a pointer chasing load. By placing just this code in the consumer and the rest in the producer, the consumer can start processing these unpredictable instructions earlier. This reduces the impact of the unpredictability of these instructions and improves performance.

Compared to just communicating data through the SPL, performing computation on the data while in flight to the consumer provides a number of sources of improvement. For one, the added computation removes instructions from one or both threads. This can help to better balance the work done by both threads and allow for more efficient pipelining. Both *astar* and *adpcm*, for example, are consumer bound with just communication. By performing computation in the SPL, computation previously performed by the consumer is now performed in the SPL, leading to more balanced producer/consumer threads. These more balanced threads spend less time waiting on a full/empty SPL queue, which improves performance. Removing instructions from one or both of the threads can also reduce pressure on the ROB and related structures, again improving performance. *Cjpeg* and *unepic* are two such examples that see reduced ROB stall time from performing computation during communication. Finally, moving computation inside the SPL can improve branch prediction in one or both threads by moving conditional operations into the SPL. *Adpcm*, *twolf*, and *wc* are three cases that see such a reduction in misprediction rate. The improved branch prediction improves processor efficiency which again improves performance.

5.1.2 Energy Efficiency Results

While adding SPL communication or computation improves performance relative to a single threaded implementation, energy efficiency may degrade given the energy costs of the extra core and SPL. Figure 10 shows energy×delay (ED) of the three SPL implementations relative to the single threaded baseline without SPL.

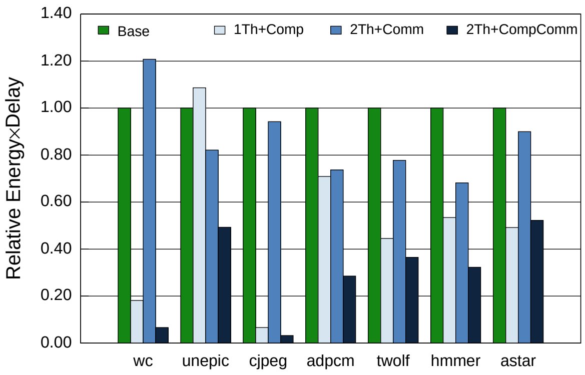


Figure 10: Energy×delay relative to single threaded baseline.

Using the SPL to perform computation in a single thread reduces ED in all but one case. In the one exception the added power consumed by the SPL outweighs the performance improvement. Similarly, communication alone reduces ED in most case but must contend with the extra energy consumed by the both second core and the SPL. Using the SPL for custom computation during communication is the only option to reduce ED relative to the single core baseline for all benchmarks. The net result is an average 70% ED savings over the single threaded baseline without SPL.

5.2 Fine-Grain Barrier Synchronization

We evaluate the performance of software (SW) versus ReMAPP barriers for our two applications when executing 2, 4, 8, and 16 threads. Figure 11 shows the performance for SW and ReMAPP barriers (with and without computation where appropriate) for the 8 and 16 threaded cases (2 and 4 threads show similar trends and are omitted for graph clarity).

Similar to other fine-grained synchronization techniques [2, 23, 25], performing barriers via ReMAPP significantly improves performance over SW barriers. For the *LL3*, the parallel ReMAPP versions start outperforming the sequential code for much smaller vector lengths. For instance, with 16 threads, ReMAPP barriers start outperforming the sequential case at a problem size between 32 and 64 whereas SW barriers only start outperforming the sequential case at a size between 64 and 128, demonstrating the benefits of finer-grain synchronization using ReMAPP barriers.

It should be noted that the restructuring of the code required to parallelize *LL3* alone has a significant impact on performance even without barrier overhead. This can be seen by comparing the performance of the Seq and SW-p1 lines in Figure 11(a)[2]. The SW-p1 case executes the parallel version of the code without any barriers. Despite having no barriers, it still performs notably worse than the sequential version. This means that the parallel versions have to overcome not only the degradation resulting from the barriers, but also the inherent loss due to how the loops must be parallelized.

In *dijkstra*, both the software and ReMAPP barriers outperform the single threaded case for all iteration sizes. ReMAPP barriers outperform software barriers for the same number of threads in all cases. In some cases, ReMAPP barriers also outperform software barriers with two or four times the number of threads.

5.2.1 Fine-Grain Barrier Synchronization with SPL Computation

The computational capabilities of the SPL can provide additional speedups not possible with fast synchronization alone. This computation is either performed as part of the barrier operation, as is done in *dijkstra* (see the discussion in Section 3.2), or in a separate SPL function that only performs computation, as in *LL3*. The execution time and performance improvement relative to barriers alone of barriers+computation for the two benchmarks are shown in Figures 11 and 12, respectively.

For *dijkstra*, where the computation is performed as part of the barrier, the benefits of adding computation are most pronounced as the number of threads increases and at finer synchronization granularities. This is due to the fact that thread synchronization, which is the portion of code accelerated by ReMAPP computation, tends to consume more time with smaller problem sizes and as the number of threads increases. In the 16 threaded case, adding computation provides up to a 9% improvement versus ReMAPP barriers alone.

In *LL3*, on the other hand, where the computation is a separate function, the most benefit is received with smaller number of threads and/or coarser synchronization granularities. This is due to the fact that in either of these cases each thread has more work to do between barriers. This means that the computation section makes up a larger percentage of the execution time and so speeding it up provides greater relative benefit. When there are an extremely small number of loop iterations per thread, the Barrier+Comp case can actually perform worse than synchronization alone as there are not enough SPL instructions to take advantage of the pipelined nature of the fabric. This can be seen in Figure 12(a) for small problem sizes with 8 and 16 threads. In each of these cases each thread has only 2 or 4 iterations to perform and so little pipelining occurs. For the larger problem sizes, however, the performance improvement is significant, ranging from 15-26%.

5.2.2 Energy Efficiency Results

Figure 13 shows energy×delay (ED) for the two synchronization workloads relative to the single threaded case. In general, the break even point for ED – the point at which the ED of the parallel case drops below the sequential case – for both SW and ReMAPP barriers requires a larger problem size (coarser grained synchronization) than the performance break even point. This occurs since, especially at very fine granularities, the performance improvement achieved by increasing the number of threads is not ideal (i.e., doubling the number of threads does not halve the run time). ReMAPP barriers always achieve better ED than their SW counterparts, despite the additional energy consumed by the SPL which is not present with SW barriers.

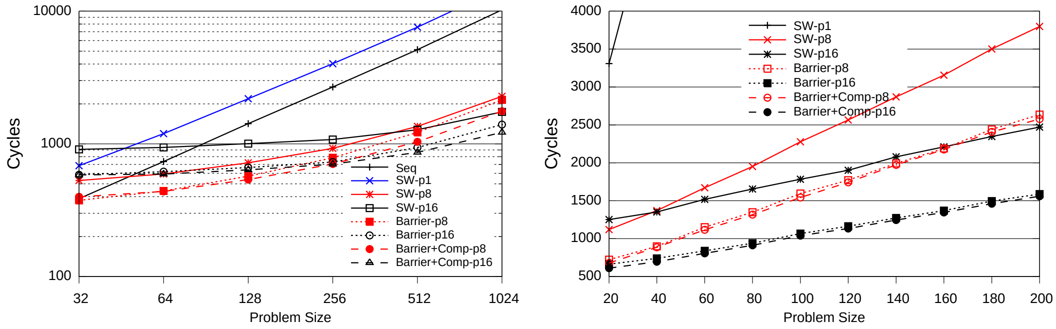
We also evaluate the ED achieved by replacing the SPL with additional cores and dedicated fine-grain barrier support [2, 25]. We simulate a system where each SPL is replaced by two additional cores (yielding a total of 24 cores for the case that originally had 16 cores+SPL) and the cores are connected with a dedicated barrier network that incurs no hardware cost. We find that, compared to such a homogeneous cluster, ReMAPP barriers+computation achieve up to 25.9% and 62.7% lower ED for *dijkstra* and *LL3*, respectively, demonstrating the benefits of ReMAPP custom computation with fine-grain barrier synchronization.

6. RELATED WORK

6.1 Reconfigurable Processors

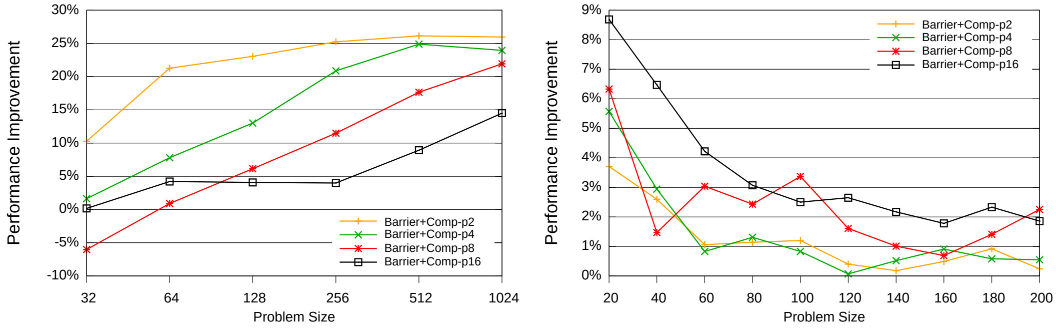
A number of research efforts [6, 9, 13] have investigated the high level integration of a reconfigurable fabric on-chip. All of these, however, only investigate the integration with a single core, although Garcia and Compton [13] state that their technique could be extended to a multicore system.

[2] *Dijkstra* does not have a SW-p1 line because the code can be parallelized without restructuring.



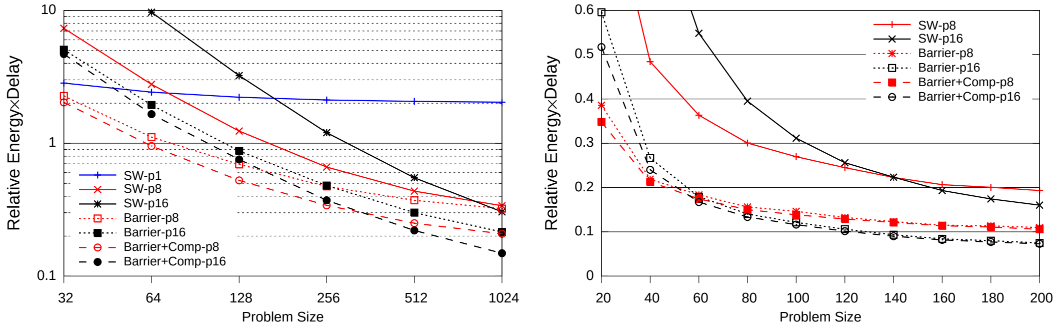
(a) LL3 (b) Dijkstra

Figure 11: Per iteration execution time for (a) Livermore loop 3 and (b) Dijkstra's Algorithm.



(a) LL3 (b) Dijkstra

Figure 12: Performance improvement of barriers+computation over barriers alone for (a) *LL3* and (b) Dijkstra'a Algorithm.



(a) LL3 (b) Dijkstra

Figure 13: Energy×Delay for (a) Livermore loop 3 and (b) Dijkstra's Algorithm relative to sequential execution.

Our previous work [28] identifies a number of features of past reconfigurable proposals that are highly amenable to incorporating reconfigurable fabrics within CMPs. That work proposed a fabric architecture, summarized in Section 2.1, tailored to sharing and analyzed its benefit for computation acceleration only (the scenario of Figure 1(a)). Our work explores how shared reconfigurable fabrics can be architected to accelerate multithreaded applications through fine-grain producer-consumer communication and fine-grain barrier synchronization with integrated custom computation.

Reconfigurable computing has recently gained increasing attention from industry. Both Intel and AMD allow tighter integration of FGPAs with general purpose processors through HyperTransport, QuickPath, and licensing of front side bus technology [11, 12]. Convey Computer's HC-1 pairs an Intel processor with a reconfigurable coprocessor and allows different instruction sets to be loaded into the coprocessor [8]. There is also growing industry interest in on-chip integration of reconfigurability in future CMPs [10].

6.2 Fine-Grained Interthread Communication

StreamIt [16, 26] is a programming language and compiler infrastructure aimed at easing the use of pipeline parallelization. Decoupled Software Pipelining (DSWP) addresses hardware options for implementing fine-grain communication [22, 21], automatic extraction of streaming threads [19], data parallelization of pipeline stages [20], and speculative DSWP [27].

Caspi et al. [7] propose SCORE, a stream computing model targeted at reconfigurable systems. Their design incorporates a single CPU and multiple reconfigurable blocks and streaming occurs between reconfigurable blocks over a dedicated interconnect. In our work, communication occurs between CPUs and the shared reconfigurable fabric is used to perform the communication.

Furthermore, none of this prior work evaluates the energy efficiency implications of streaming. Energy usage is a non-trivial concern given the fact that streaming tends to provide less than ideal speedups.

6.3 Fine-Grained Synchronization

Beckmann and Polychronopoulos [2] and Shang and Hwang [25] both provide hardware mechanisms for performing barriers using dedicated interconnect and hardware tables. IBM's Cyclops architecture [4] provides dedicated hardware support for barriers through a special purpose register and wired-OR. Sampson et al. [23] propose barrier filters to eliminate the dedicated interconnect required in most barrier synchronization proposals. The Multi-ALU Processor [18] provides an explicit barrier instruction in the ISA and supports register to register communication between clusters.

7. CONCLUSIONS

We propose ReMAPP, a shared reconfigurable architecture designed to accelerate otherwise sequential regions of code by enabling parallelization through multiple forms of fine-grained communication. In contrast to previous fine-grain communication approaches, ReMAPP enables custom functions to be integrated with communication. Combining computation with communication provides an average 221% performance improvement for fine-grained interthread communication. In the case of barrier synchronization, performing computation during the barrier provides up to a 9% performance improvement over the SPL barrier alone. ReMAPP achieves the aforementioned performance improvements while also improving energy × delay.

8. REFERENCES

[1] Advanced Micro Devices. AMD Athlon X2 Dual-Core Details. `http://www.amdcompare.com/us-en/desktop/-details.aspx?opn=ADH2350IAA5DD`, 2007.

[2] C. J. Beckmann and C. D. Polychronopoulos. Fast Barrier Synchronization Hardware. In *Supercomputing '90: Proceedings of the 1990 ACM/IEEE Conference on Supercomputing*, pages 180–189, 1990.

[3] M. Budiu and S. C. Goldstein. Fast Compilation for Pipelined Reconfigurable Fabrics. In *Proc. 1999 ACM/SIGDA 7th Int'l Symposium on Field Programmable Gate Arrays*, pages 195–205, Feb. 1999.

[4] C. Caşcaval, J. Castaños, L. Ceze, M. Denneau, M. Gupta, D. Lieber, J. Moreira, K. Strauss, and H. Warren Jr. Evaluation of a Multithreaded Architecture for Cellular Computing. In *Proc. 8th IEEE Symposium on High Performance Computer Architecture*, pages 311–321, 2002.

[5] T. Callahan, J. Hauser, and J. Wawrzynek. The Garp Architecture and C Compiler. *Computer*, 33:62–69, Apr. 2000.

[6] J. Carrillo and P. Chow. The Effect of Reconfigurable Units in Superscalar Processors. In *Proc. 2001 ACM/SIGDA 9th Int'l Symposium on Field Programmable Gate Arrays*, pages 141–150, 2001.

[7] E. Caspi, M. Chu, R. Huang, J. Yeh, J. Wawrzynek, and A. DeHon. Stream Computations Organized for Reconfigurable Execution (SCORE). In *Proceedings of the 10th Int'l Workshop on Field-Programmable Logic and Applications*, pages 605–614, Aug. 2000.

[8] Convey Computer. The Convey HC-1 Computer. White Paper, Nov. 2008.

[9] M. Dales. Managing a Reconfigurable Processor in a General Purpose Workstation Environment. In *Proc. of the Design, Automation, and Test in Europe Converence and Exhibition*, pages 980–985, 2003.

[10] J. Emer. An Evolution of General Purpose Processing: Reconfigurable Logic Computing. International Symposium on Code Generation and Optimization Keynote Address, Mar. 2009.

[11] M. Feldman. FPGA Acceleration Gets a Boost from HP, Intel. *HPCWire*, Sept. 2007.

[12] M. Feldman. Reconfigurable Computing Prospects on the Rise. *HPCWire*, Dec. 2008.

[13] P. Garcia and K. Compton. A Reconfigurable Hardware Interface for a Modern Computing System. *Proc. 2007 IEEE Symposium on Field–Programmable Custom Computing Machines*, pages 73–84, April 2007.

[14] S. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R. Taylor. PipeRench: A Reconfigurable Architecture and Compiler. *Computer*, 33:70–77, 2000.

[15] S. Goldstein, H. Schmit, M. Moe, M. Budiu, and S. Cadambi. PipeRench: A Coprocessor for Streaming Multimedia Acceleration. In *Proc. 26th IEEE/ACM Int'l Symposium on Computer Architecture*, pages 28–39, May 1999.

[16] M. I. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, A. A. Lamb, C. Leger, J. Wong, H. Hoffmann, D. Maze, and S. Amarasinghe. A Stream Compiler for Communication-Exposed Architectures. In *Proc. 10th ACM Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 291–303, Oct. 2002.

[17] *Intel Core 2 Extreme Processor X6800 and Intel Core 2 Duo Desktop Processor E6000 and E4000 Sequences*, 2007. Intel Datasheet: 313278-004.

[18] S. W. Keckler, W. J. Dally, D. Maskit, N. P. Carter, A. Chang, and W. S. Lee. Exploiting Fine-Grain Thread Level Parallelism on the MIT Multi-ALU Processor. In *Proc. 25th IEEE/ACM Int'l Symposium on Computer Architecture*, pages 306–317, June 1998.

[19] G. Ottoni, R. Rangan, A. Stoler, and D. August. Automatic Thread Extraction with Decoupled Software Pipelining. In *Proc. IEEE/ACM 38th Annual Int'l Symposium on Microarchitecture*, pages 105–118, Nov. 2005.

[20] E. Raman, G. Ottoni, A. Raman, M. J. Bridges, and D. I. August. Parallel-Stage Decoupled Software Pipelining. In *CGO '08: Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 114–123, Apr. 2008.

[21] R. Rangan, N. Vachharajani, A. Stoler, G. Ottoni, D. August, and G. Cai. Support for High-Frequency Streaming in CMPs. In *Proc. IEEE/ACM 39th Annual Int'l Symposium on Microarchitecture*, pages 259–272, Dec. 2006.

[22] R. Rangan, N. Vachharajani, M. Vachharajani, and D. August. Decoupled Software Pipelining with the Synchronization Array. In *Proc. 13th IEEE/ACM Int'l Conference on Parallel Architectures and Compilation Techniques*, pages 177–188, Oct. 2004.

[23] J. Sampson, R. Gonzalez, J.-F. Collard, N. P. Jouppi, M. Schlansker, and B. Calder. Exploiting Fine-Grained Data Parallelism with Chip Multiprocessors and Fast Barriers. In *Proc. IEEE/ACM 39th Annual Int'l Symposium on Microarchitecture*, pages 235–246, Dec. 2006.

[24] SESC Architectural Simulator. `http://sourceforge.net/projects/sesc`, 2007.

[25] S. Shang and K. Hwang. Distributed Hardwired Barrier Synchronization for Scalable Multiprocessor Clusters. *IEEE Trans. Parallel Distrib. Syst.*, 6(6):591–605, 1995.

[26] W. Thies, M. Karczmarek, and S. P. Amarasinghe. StreamIt: A Language for Streaming Applications. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, pages 179–196, 2002.

[27] N. Vachharajani, R. Rangan, E. Raman, M. J. Bridges, G. Ottoni, and D. I. August. Speculative Decoupled Software Pipelining. In *Proc. 16th IEEE/ACM Int'l Conference on Parallel Architectures and Compilation Techniques*, pages 49–59, Sept. 2007.

[28] M. Watkins, M. Cianchetti, and D. Albonesi. Shared Reconfigurable Architectures for CMPs. In *Proc. 18th IEEE Int'l Conference on Field Programmable Logic and Applications*, Sept. 2008.

[29] Z. A. Ye, N. Shenoy, and P. Banerjee. A C Compiler for a Processor with a Reconfigurable Functional Unit. In *Proc. 2000 ACM/SIGDA 8th Int'l Symposium on Field Programmable Gate Arrays*, pages 95–100, Feb. 2000.