ReMAP: A Reconfigurable Heterogeneous Multicore Architecture

Matthew A. Watkins and David H. Albonesi
Computer Systems Laboratory
Cornell University, Ithaca, NY 14853
maw72,dha7@cornell.edu

***Abstract*—This paper presents ReMAP, a reconfigurable architecture geared towards accelerating and parallelizing applications within a heterogeneous CMP. In ReMAP, threads share a common reconfigurable fabric that can be configured for individual thread computation or fine-grained communication with integrated computation. The architecture supports both fine-grained point-to-point communication for pipeline parallelization and fine-grained barrier synchronization.**

The combination of communication and configurable computation within ReMAP provides the unique ability to perform customized computation while data is transferred between cores, and to execute custom global functions after barrier synchronization. ReMAP demonstrates 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 additional or more powerful cores.

I. Introduction

Reconfigurable computing has traditionally involved attaching a reconfigurable fabric to a single processor core to accelerate sequential applications. 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 multithreaded applications. This paper presents ReMAP (Reconfigurable Multicore Acceleration and Parallelization), a reconfigurable architecture that accelerates the computation and communication of multiple threads within a heterogeneous CMP 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, some of which may realize no benefit from the fabric. 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. As the industry moves to more cores on a die, the proportional cost of incorporating a reconfigurable fabric decreases, as does the proportion of applications needed to justify the presence of the fabric. Second, the area and power costs of the fabric may be amortized over several threads by sharing the fabric among multiple cores, thereby forming a *cluster* of cores+fabric. With intelligent fabric management, such sharing can increase fabric utilization and reduce overall fabric area and power costs, while achieving nearly the same performance as providing each core with its own, much larger, private fabric [32], [33].

Sharing the reconfigurable fabric among multiple cores also creates optimization opportunities not possible with per-core private fabrics. In particular, shared fabric clusters – in addition to amortizing the fabric area and increasing power efficiency – can be organized on-the-fly in multiple ways to accelerate multithreaded applications in addition to the sequential applications that have traditionally been the focus of reconfigurable architectures. Figure 1 depicts a portion of a heterogeneous CMP and provides a simplified view of the three ways that the ReMAP architecture is dynamically organized to accelerate multiple threads. Figure 1(a) depicts four threads, each of which is independently performing a function (each function f_i may be unique or identical) within the fabric without communication. In Figure 1(b), the fabric is being used for two instances of fine-grain producer-consumer communication with integrated customized computation. In each instance, the producer thread feeds inputs into the fabric; the inputs pass through the fabric to perform the function; and the function output, which may be queued using any remaining fabric resources if necessary, is then passed to the consumer thread. 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.

While all of these cases show the individual threads or producer-consumer thread pairs temporally sharing the fabric, to reduce fabric contention, the fabric manager can dynamically partition the fabric, giving each thread a smaller private fabric[1]. Dynamic fabric partitioning can also serve to simultaneously configure the fabric for multiple purposes, e.g., for independent use by one set of threads in one partition, and producer-consumer communication in another.

Unlike previous proposals, ReMAP supports multiple communication models and also provides the ability to perform customized computation on communicated data.

[1]As we explain in Section II, the virtualization of the fabric makes this dynamic division of the fabric transparent to software.

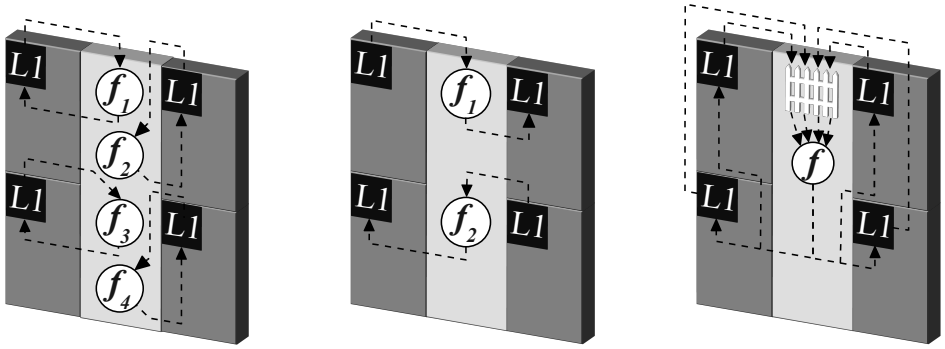


Figure 1. Shared reconfigurable fabric being used for (a) individual computation, (b) producer-consumer communication with computation, and (c) barrier synchronization with computation.

The latter provides performance improvements beyond what is possible with previous communication-only options and traditional fixed computation alternatives.

In the next section we describe the ReMAP architecture and in Section III provide examples of ReMAP communication. Our methodology is presented in Section IV followed by the evaluation of ReMAP in Section V. We describe related work in Section VI and conclude in Section VII.

II. ReMAP Architecture

ReMAP pairs a specially designed *Specialized Programmable Logic (SPL)* fabric with multiple cores of a CMP. An example ReMAP heterogeneous CMP with integrated SPL is depicted in Figure 2(a). The figure shows a 16 core ReMAP CMP[2] with two SPL clusters on the left. Each cluster consists of four single issue out-of-order processor cores sharing an SPL fabric, which is shown at a high level in Figure 2(b) and explained in more detail in the next section. The fabric is temporally shared in a round-robin fashion among the cores in the same cluster and can be spatially partitioned as needed 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 SPL clusters are implemented. In a large-scale heterogeneous CMP with many tens or hundreds of cores, there may be several SPL 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. Applications are mapped to an SPL cluster during phases that use the fabric and are mapped to other clusters during other phases in order to obtain the best overall performance.

A. SPL Organization

The computational substrate of ReMAP is the highly-pipelined, row-based SPL of [33]. 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 group of 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 [16] and the AMD X2 Dual-Core [1], both of which are implemented in the same 65nm technology assumed for ReMAP) and allows each row to complete the longest permissible computation in a single cycle. Table I shows the relative area and power consumption of the SPL and associated single-issue cores, derived using the methodology of [33].

Table I
Relative area and power of four single-issue out-of-order cores and four-way shared ReMAP fabric.

	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

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 fabric [13]. 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

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

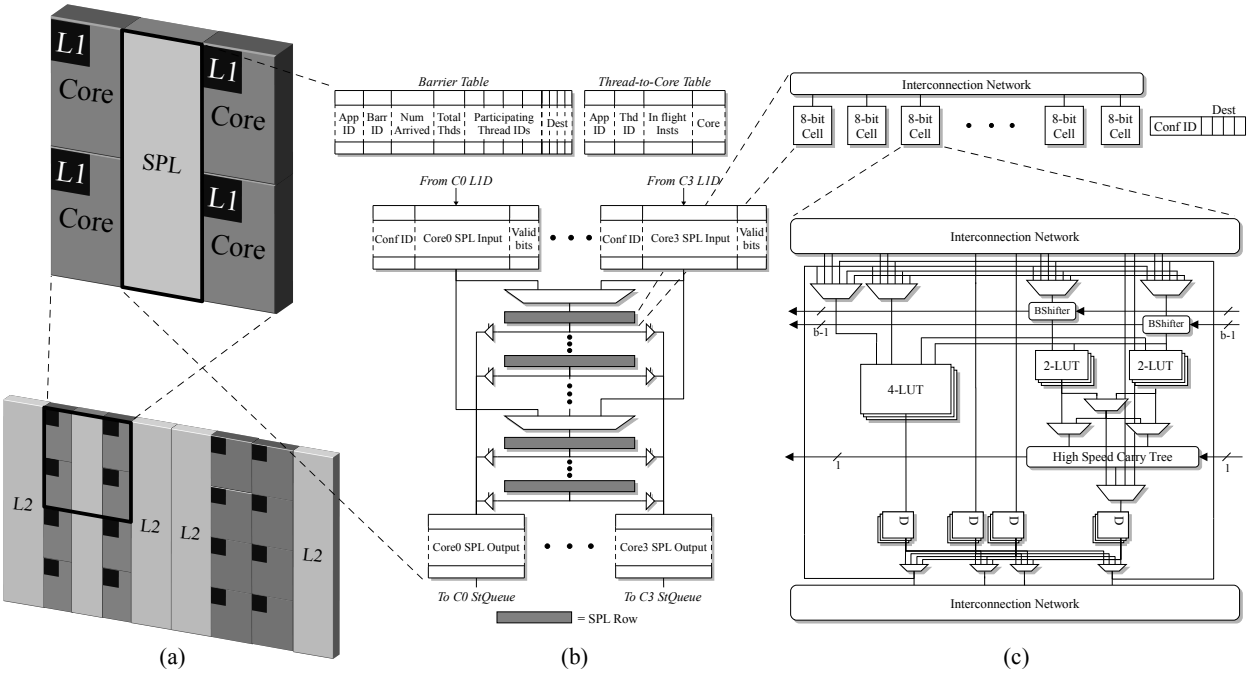


Figure 2. ReMAP heterogeneous CMP. (a) Depiction of overall ReMAP chip, with two SPL clusters and one conventional cluster, and blow-up of one SPL cluster, (b) four-way shared SPL including tables required for communication, and (c) design of SPL row and cell.

in Figure 2(b). SPL Load instructions place values into the input queue at a particular data alignment. The SPL similarly writes results to a local output queue that is then written out to the Store Queue using an SPL Store instruction.

The SPL is temporally shared in a time-multiplexed, round-robin fashion among the cores sharing the fabric. The SPL also supports spatial partitioning where the fabric is divided into up to four virtual clusters. 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. Figure 2(b) shows the additional multiplexers and tristate drivers necessary to support both forms of sharing. Figure 2 also shows the barrier and thread-to-core tables, input queue valid bits, and row destination IDs added to support interthread and barrier communication, which are discussed in the upcoming sections.

B. Support for Fine-Grained Communication+Computation

Most multithreaded applications use some form of communication to coordinate their activity. At a high level, communication requires the exchange of information among threads, be it a notification of a thread arriving at a barrier or a producing thread passing results to a consuming thread. ReMAP 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 hard-wired communication-only mechanisms.

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], [24]. 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 output queue to memory (i.e., store queue) (Figure 3(d)).

A small table (the Thread-to-Core Table in Figure 2(b)) maintains a mapping of threads to cores to virtualize the

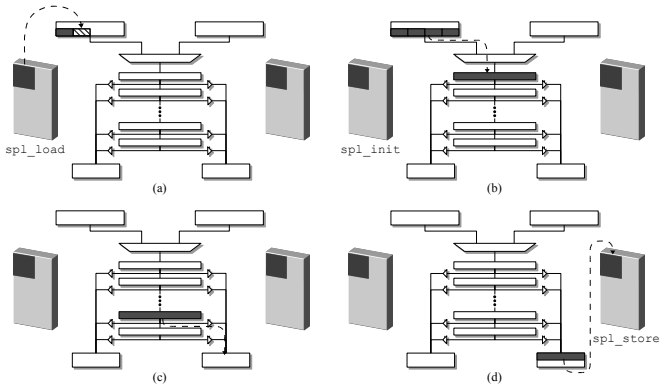


Figure 3. Walk through of intercore communication with integrated computation.

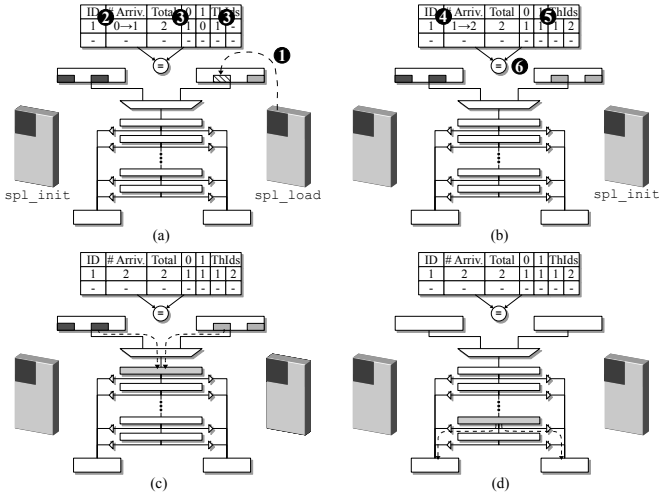


Figure 4. Walk through of barrier synchronization with integrated computation.

selection of the destination core. When an SPL instruction is issued, it obtains the core currently assigned to its destination 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).

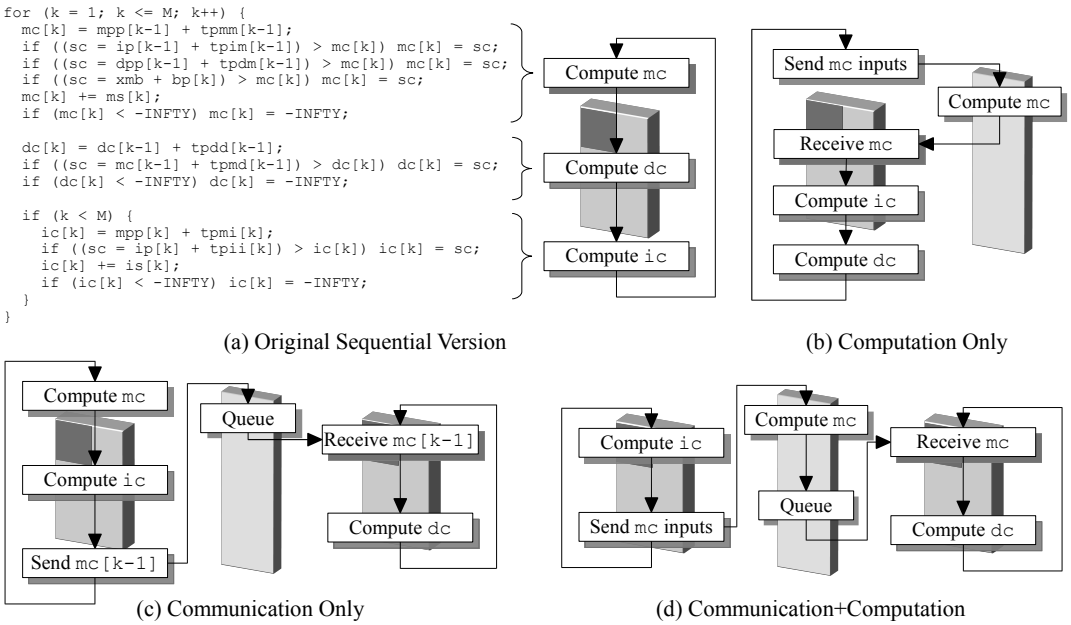
A side benefit of this table based approach is that instructions will not issue to the fabric if the destination thread is not available (not present in the table). This prevents the producing thread from filling up the fabric if the consumer is not present.

Even with the table, however, SPL instructions could accumulate in the fabric if the consumer thread is switched out while data is in flight to it, which would require the consumer to be switched back into the same core to receive the values. To prevent this situation, the thread-to-core mapping table maintains a count of the number of in-flight SPL instructions destined for each core. On a request to switch out, a thread checks the number of in-flight SPL instructions bound for its core. If this is greater than zero, the fabric is blocked from accepting any new instructions destined for that thread and the thread continues to execute until the in-flight counter reaches zero. At this point the thread can be switched out and the fabric unblocked.

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 number of threads increases. This overhead prevents the use of barriers at fine granularities. Various proposals [2], [4], [25], [27] 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, ReMAP may directly synthesize the function into the fabric with the output communicated to the participants' output queues.

To implement barriers in ReMAP, SPL barrier instructions (indicated by a flag in the SPL 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 an SPL initiate instruction, an instruction is issued to the fabric by the SPL 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 indicate 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 (the Barrier Table in Figure 2(b)) with information related to each active barrier. Each table contains as many entries as cores attached to an SPL 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 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



(a) Original Sequential Version

(b) Computation Only

(c) Communication Only

(d) Communication+Computation

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

thread counts are compared to determine when to issue an instruction (❻).

In a system with multiple SPL clusters, a dedicated bus communicates barrier updates among clusters. The bus transmits the barrier ID as well as the associated application ID. With a limit of 256 IDs, the shared bus requires only 16 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 ReMAP controller triggers an exception to switch the missing threads back in. Once all threads are available, the barrier can proceed.

III. 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 from the enhanced communication and additionally benefit from using the computational power of the SPL while communicating.

A. 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.

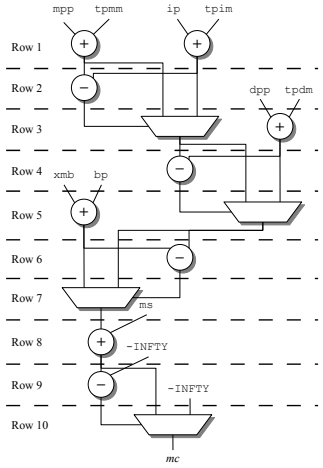
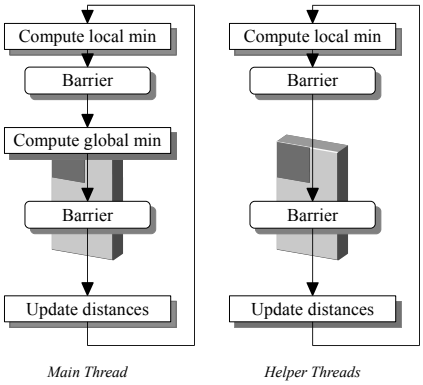


Figure 6. SPL `mc` calculation mapping.

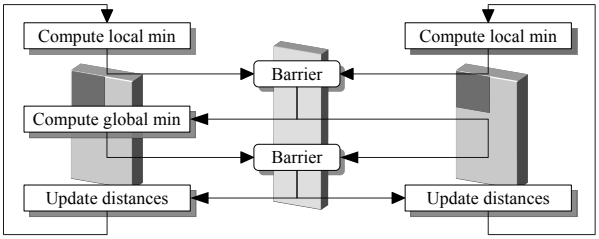
We first look at how the SPL can accelerate a portion of the computation, specifically the calculation of `mc`. As shown in Figure 5(b), the core loads the input values needed to compute `mc` into the fabric, the SPL computes the value of `mc`, and the core receives the result. After receiving `mc`, 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

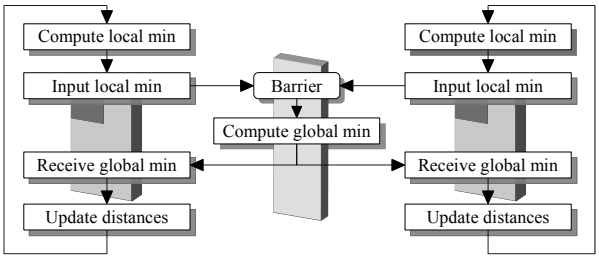
```
for(int k = 0; k<NUM_NODES; k++){
  getLocalMin(); //get min dist of my available nodes
  localMins[myId] = myLocalMin; //place min in global location
  barrier();
  mem_fence();
  if(myId == 0){
    globalMin = localMin[0];
    for(int n = 1; n<NTHREADS; n++){
      if(localMins[n] < globalMin)
      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.

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 combined in the SPL. The producer thread computes `ic` and loads the inputs needed for `mc`. The SPL computes the value of `mc` 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 V-B).

B. Barrier Synchronization+Computation Example

To show the operation of ReMAP 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 speedup for small to moderate graph sizes. By using the SPL 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 is to replace the software barriers with ReMAP barriers, as shown in Figure 7(b). As with previous dedicated barrier techniques [2], [27], replacing the software barriers with ReMAP barriers provides significant performance improvements. Performance can be further improved beyond that possible with previous techniques by using the computational power of the SPL to compute the global minimum within the fabric. Figure 7(c) shows this optimization for the case where all threads share a single SPL 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

Table II
ARCHITECTURE PARAMETERS.

	OOO1	OOO2
Fetch/Decode/Rename Width	2	4
Issue/Retire Width	1	2
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	2/1
Branch Units	1	2
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	

the extra barrier, performance is still improved over using ReMAP for communication only (see Section V-C).

IV. EVALUATION METHODOLOGY

We use a modified version of SESC [26] 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 for the single-issue (OOO1) and dual-issue (OOO2) out-of-order cores used in the evaluation are shown in Table II. We use Wattch, Cacti, and HotLeakage to model power.

A. Benchmarks

We use benchmarks from the SPEC 2000 [28] and 2006 [29], MediaBench [18], MiBench [10], and Livermore Loops [19] suites along with the Unix utility *wc* to analyze the three usage modes from Figure 1. A complete list of the benchmarks used for each operation mode, the functions we optimize in each, and the percentage of total program execution time consumed by the functions are listed in Table III. *Cjpeg* makes use of two operation modes, computation-only and computation+communication, and is evaluated with other communicating workloads. We execute two 250 million instruction SimPoints [21] for SPEC workloads with reference inputs and run all other workloads to completion.

To evaluate barrier synchronization we create parallel versions of the listed applications. Two of the benchmarks, specifically Livermore Loop 3, which is transformed to operate on integers, and Dijkstra's algorithm, include computation in the fabric after synchronization. In Dijkstra's Algorithm, computation is performed during the synchronization operation (as in Figure 1(c)). *LL3* makes use of two ReMAP modes of operation: performing computation on the data within the loop (Figure 1(a)), and using the SPL to accelerate synchronization between iterations (Figure 1(c)).

B. ReMAP Programming

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], [34] and good partitionings for pipelined applications [15], [20]. We believe our design could leverage this prior art in an actual implementation.

Table III
BENCHMARK DETAILS.

Benchmark	Functions Optimized	% Exec Time
Computation Only		
g721dec	fmult	48%
g721enc	fmult	46%
mpeg2dec	store_ppm_tga, conv422to444, conv420to422	63%
mpeg2enc	dist1	70%
gsmtoast	Calculation_of_the_LTP_parameters Weighting_filter	54%
gsmuntoast	Short_term_synthesis_filtering	76%
462.libquantum	quantum_toffoli, quantum_cnot	40%
Communication+Computation		
wc	wc	100%
unepic	read_and_huffman_decode	22%
cjpeg	rgb_ycc_convert, jpeg_fdct_islow	50%
adpcm	adpcm_decoder	99%
300.twolf	new_dbox_a	30%
456.hmmer	P7Viterbi	85%
473.astar	regwayobj::makebound2	33%
Barrier Synchronization		
Livermore Loop 2 (LL2)		100%
Livermore Loop 3 (LL3)		100%
Livermore Loop 6 (LL6)		100%
Dikjstra's Algorithm		100%

V. RESULTS

We first evaluate ReMAP in the context of a heterogeneous CMP executing entire programs and then evaluate the performance of the optimized regions to show the sources of the improvements.

A. ReMAP in a Heterogeneous CMP

The ReMAP system is composed of clusters of four OOO1 cores plus a 24-row SPL, coupled with clusters of OOO2 cores (Figure 2(a)). An alternative is a cluster of four OOO2 cores with a dedicated communication network (*OOO2+Comm*), similar to previous proposals [7], [24]. Assuming zero hardware cost for the communication network, an *OOO2+Comm* cluster consumes approximately the same area as an SPL cluster. In the ReMAP configuration, regions of code that utilize the SPL are run on the SPL cluster while other regions are run on an OOO2 core. The migration overhead is accounted for by draining in-flight instructions and stopping execution for 500 cycles to model the time necessary to context switch all state to the new core (determined by running the requisite code in the simulator).

We compare these two schemes using workloads that use the SPL for computation alone and those that use the SPL for computation+communication (results for barrier synchronization are discussed separately in Section V-C).

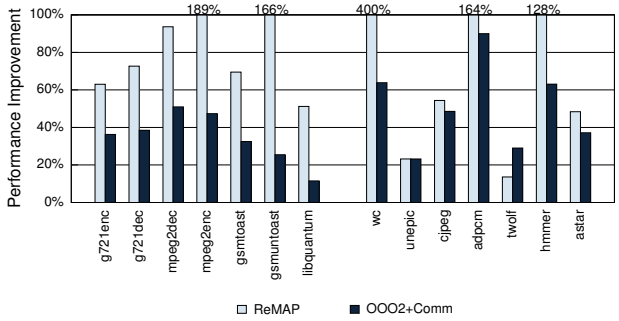


Figure 8. Performance relative to single threaded baseline.

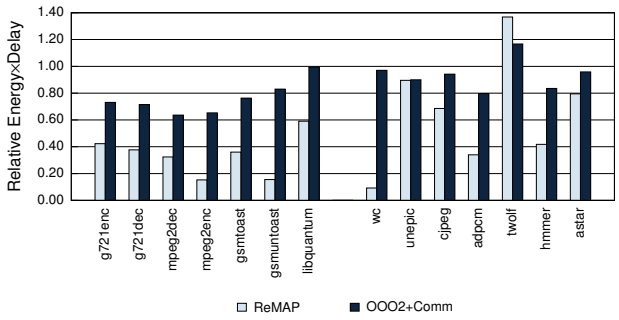


Figure 9. Energy×delay relative to single threaded baseline.

Computation-only workloads are run concurrently with other computation-only workloads to account for SPL contention. Communicating workloads are run separately, but are given access to only half of the SPL, under the assumption that the other half is in use by another communicating pair.

The performance improvement of the two configurations relative to executing the original sequential code on a OOO1 core is shown in Figure 8. ReMAP performs as well or better than *OOO2+Comm* in all but one case. On average ReMAP performs 49% better than *OOO2+Comm* for computation-only workloads and 41% better for communicating workloads. In the one exception, *twolf*, the time duration of the sequential regions are so short that the migration cost outweighs the benefit of executing these regions on the 2-way issue core. The performance benefit of executing on the 2-way issue core during the sequential regions (with *OOO2+Comm*) outweighs the benefit of executing on the SPL during the parallel sections.

Figure 9 shows energy×delay (ED) for the two configurations relative to the single threaded baseline. ReMAP provides better ED than both the baseline and *OOO2+Comm* configurations in all but one case. The one exception is again *twolf*, where both alternatives achieve worse ED than the baseline, indicating that *twolf* should be run as a single thread on a simple core if energy is a significant concern. *OOO2+Comm* also generally provides better ED than the baseline, but only marginally so in cases such as *cjpeg*, *astar*, and *libquantum*. With the exception of *twolf*, ReMAP provides 45% better performance on average and 35% lower energy consumption on average than *OOO2+Comm*.

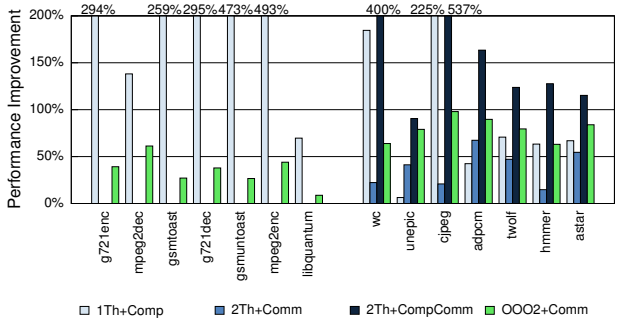


Figure 10. Performance improvement of optimized functions relative to performance of single threaded baseline.

B. Analysis of Optimized Regions

We now analyze the code regions optimized for ReMAP to see the source of the above improvements. Figure 10 shows the performance improvements relative to the single threaded baseline of a single thread using the SPL for computation (*1Th+Comp*) and (where appropriate) dual threads with the SPL used for communication (*2Th+Comm*) and dual threads with the SPL used for computation+communication (*2Th+CompComm*). We also show the dual threaded case running on OOO2 cores with idealized communication hardware (*OOO2+Comm*). Using the SPL for computation (*1Th+Comp*) provides significant performance improvements (289% and 105% on average for computation-only and communicating workloads, respectively).

Focusing on the workloads employing communication, using the SPL for producer-consumer communication alone provides a 38% improvement in performance for the optimized region relative to the single core baseline. Combining both computation and communication (*2Th+CompComm*) increases the average performance improvement to 223%. It is only with the combination of computation and communication that ReMAP outperforms the *OOO2+Comm* alternative in all cases (by 79% on average), showing the clear benefit of integrating SPL computation and communication.

To confirm the need for hardware-based communication, we also ran the benchmarks with software queues, with and without SPL computation. Software queues degraded performance by more than a 180% on average relative to the OOO1 baseline.

1) Contributing Factors: We analyzed 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, by splitting the application into a producer/consumer pair we can place sections of code with poor branch or load performance

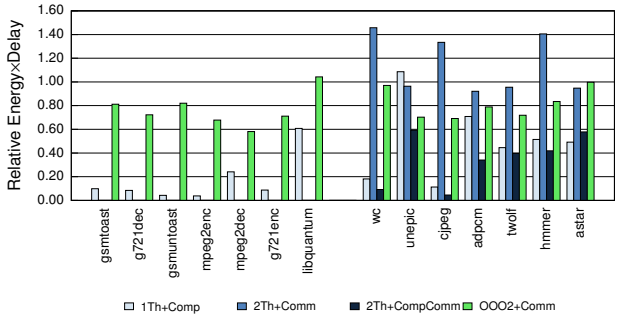


Figure 11. Energy×delay of optimized fuctions relative to single threaded baseline.

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. With ReMAP communication we can also perform computation during the communication, meaning 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.

Compared to just communicating data, performing computation on the data while in flight to the consumer provides multiple sources of improvement. For one, the SPL computation removes instructions from one or both threads. This can 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 examples that see reduced ROB stall time with integrated computation. Finally, moving computation inside the SPL can improve branch prediction in one or both threads by moving conditional operations into the SPL. *Adpcm* and *wc* are two cases that see such a reduction in misprediction rate. The improved branch prediction improves processor efficiency which again improves performance.

2) Energy Efficiency Results: Figure 11 shows the ED of the three SPL implementations and the *OOO2+Comm* alternative relative to the single threaded baseline without SPL. While adding computation or communication in isolation reduces ED in many cases, neither is able to provide enough performance benefit to overcome the added power consumed by the extra core and SPL in all cases. ReMAP communication+computation always improves performance and reduces energy consumption compared to *OOO2+Comm* and is the only option to provide better ED than the the single threaded baseline in all cases.

C. Fine-Grain Barrier Synchronization

We evaluate the performance of software (SW) versus ReMAP barriers for our four barrier applications when executing 2, 4, 8, and 16 threads. Figure 12 shows the performance for SW and ReMAP barriers (with and without computation where appropriate) for the 8 and 16 threaded cases. Results for 2 and 4 threads show similar trends.

Similar to other fine-grained synchronization techniques [2], [25], [27], performing barriers via ReMAP significantly improves performance over SW barriers. For the Livermore Loops, the ReMAP versions start outperforming the sequential code for much smaller vector lengths. Fine-grained synchronization also makes larger thread counts useful for smaller problem sizes. In *dijkstra*, ReMAP barriers not only outperform software barriers with the same number of threads but also outperform software barriers with two or four times the number of threads in some cases.

1) Fine-Grain Barrier Synchronization with SPL Computation: Certain parallel benchmarks also benefit from the computational capabilities of the SPL. This computation is either performed as part of the barrier operation, as in *dijkstra*, or in a separate SPL function that only performs computation, as in *LL3*. The execution time and performance improvements of barriers+computation relative to barriers alone for the two benchmarks are shown in Figures 12(c-d) and 13, respectively.

For *dijkstra*, where the computation is integrated with the barrier, the benefits of adding computation are most pronounced with more threads and at finer synchronization granularities. This occurs because thread synchronization, which is the portion of code accelerated by the SPL, consumes more time with smaller problem sizes and more threads. In the 16 thread case, adding computation provides up to a 9% improvement versus hardware barriers alone.

In *LL3*, where the computation is a separate function, the greatest benefit occurs with fewer threads and coarser synchronization granularities. In either of these cases each thread has more work to do between barriers, resulting in the computation section, the part accelerated by the SPL, increasing as a percentage of the overall execution time. 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 13(a) for small problem sizes and large thread counts. In each case each thread has only 2 or 4 iterations to perform and so little

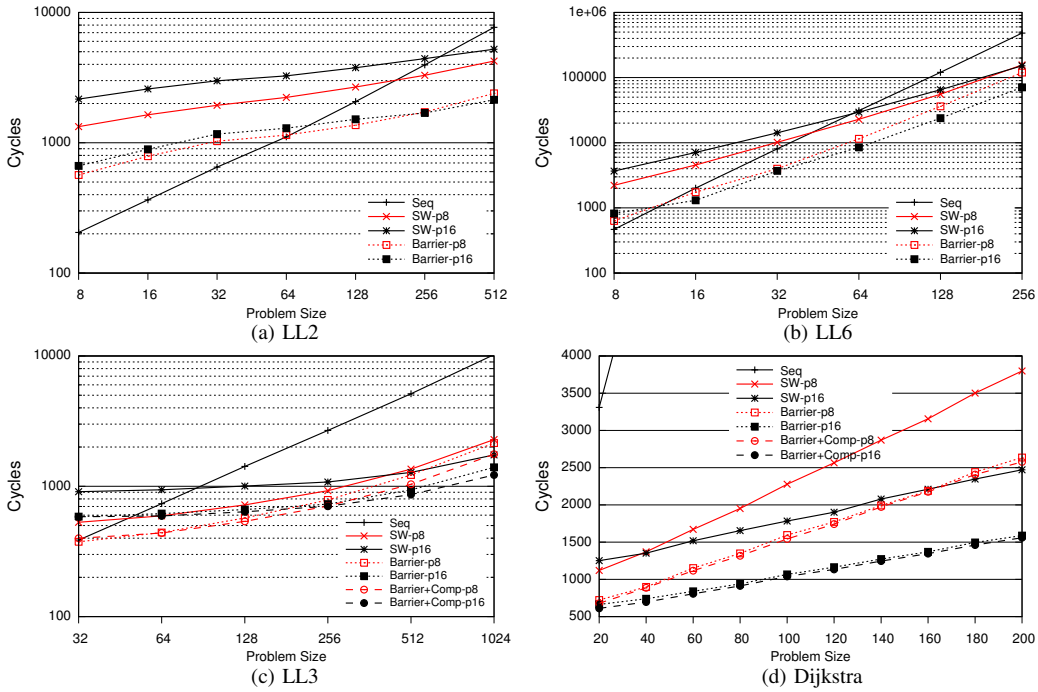


Figure 12. Per iteration execution time for Livermore loops (a) 2, (b) 6, and (c) 3 and (d) Dijkstra's Algorithm.

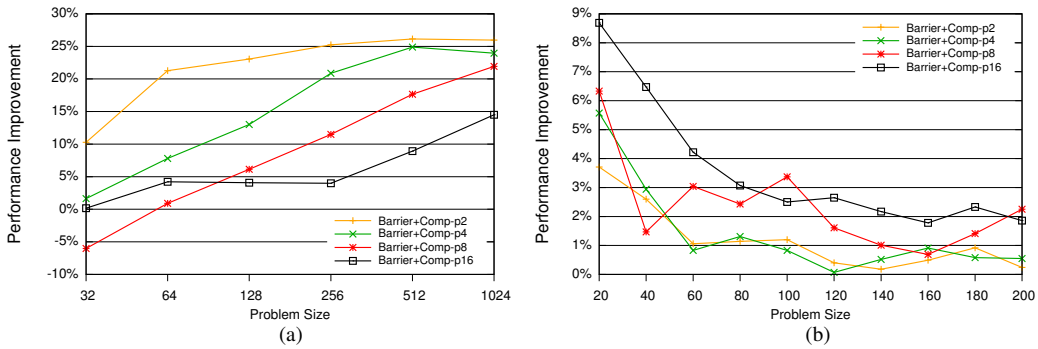


Figure 13. Performance improvement of barriers+computation over barriers alone for (a) *LL3* and (b) *dijkstra*.

pipelining occurs. For the larger problem sizes, however, the performance improvement is significant, ranging from 15-26%.

2) Energy Efficiency Results: Figure 14 shows energy×delay (ED) results for the four 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 ReMAP 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). For 16 threaded *LL2* and *LL6*, SW barriers *never* break even for the problem sizes we investigate. ReMAP barriers always achieve better ED than their SW counterparts, despite the additional energy consumed by the SPL.

We also evaluate the performance achieved by replacing the SPL with additional cores and dedicated fine-grain barrier support [2], [27]. Since the SPL consumes as much area as two single-issue cores, we simulate a system where each SPL is replaced by two additional cores and the cores are connected with a dedicated barrier network that incurs no hardware cost. We find that, compared to such a homogeneous cluster, ReMAP barriers+computation achieves up to 25.9% and 62.5% lower ED for *dijkstra* and *LL3*, respectively, demonstrating the benefits of ReMAP custom computation with fine-grain barrier synchronization.

VI. Related Work

A number of research efforts [6], [9], [11] have investigated the high level integration of a reconfigurable fabric on-

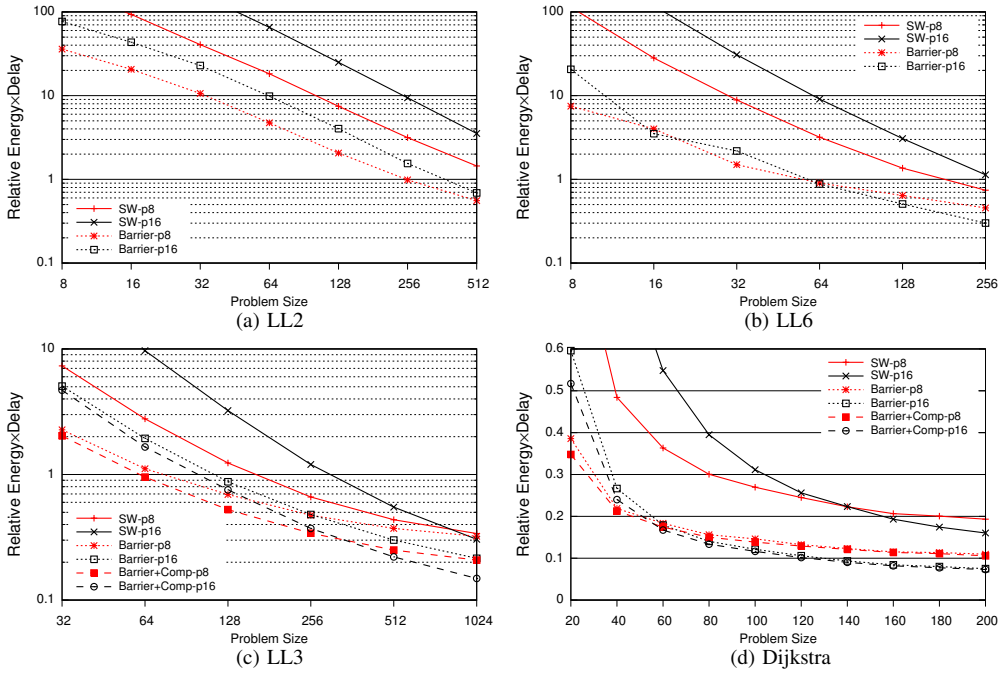


Figure 14. Energy×Delay for Livermore loops (a) 2, (b) 6, and (c) 3 and (d) Dijkstra's Algorithm relative to sequential execution.

chip. All of these, however, only investigate the integration with a single core, although Garcia and Compton [11] state that their technique could be extended to a multicore system.

In [12], configuration data for a reconfigurable coprocessor is shared among multiple cores all running the same application to improve fabric utilization. Chen et al. [8] investigate the benefits of including reconfigurable ISA support in a multicore processor and find that combining program parallelization with custom ISA support provides larger speedups than the sum of the two techniques applied in isolation.

StreamIt [15], [30] 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 [24], [23], automatic extraction of streaming threads [20], data parallelization of pipeline stages [22], and speculative DSWP [31]. 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.

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.

Beckmann and Polychronopoulos [2] and Shang and Hwang [27] both propose 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. [25] propose barrier filters to eliminate the dedicated interconnect required in most barrier synchronization proposals. The Multi-ALU Processor [17] provides an explicit barrier instruction in the ISA and supports register to register communication between clusters.

VII. Conclusions

We propose ReMAP, a shared reconfigurable architecture for accelerating and parallelizing applications in a heterogeneous CMP. In addition to accelerating computation like traditional reconfigurable fabrics, ReMAP can be configured to facilitate multiple forms of fine-grained communication. In contrast to previous fine-grain communication approaches, ReMAP enables custom computation to be integrated with communication. Combining these multiple modes improves performance by 45% over an area equivalent system with larger cores and dedicated hardware communication. We also show that ReMAP provides better energy efficiency than can be achieved by using the area consumed by the SPL for either additional or more powerful cores, providing a 44% reduction in ED. These results demonstrate the significant advantages of incorporating reconfigurability into future heterogeneous CMPs.

Acknowledgments

This research was supported by an NSF Graduate Research Fellowship; NSF grants CCF-0916821, CCF-

0811729, and CNS-0708788; and equipment grants from Intel.

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*, 1990, pp. 180–189.

[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*, Feb. 1999, pp. 195–205.

[4] C. Caşcaval, J. Castaños, L. Ceze *et al.*, "Evaluation of a Multithreaded Architecture for Cellular Computing," in *Proc. 8th IEEE Symposium on High Performance Computer Architecture*, 2002, pp. 311–321.

[5] T. Callahan, J. Hauser, and J. Wawrzynek, "The Garp Architecture and C Compiler," *Computer*, vol. 33, pp. 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*, 2001, pp. 141–150.

[7] E. Caspi, M. Chu, R. Huang *et al.*, "Stream Computations Organized for Reconfigurable Execution (SCORE)," in *Proceedings of the 10th Int'l Workshop on Field-Programmable Logic and Applications*, Aug. 2000, pp. 605–614.

[8] Z. Chen, R. N. Pittman, and A. Forin, "Combining Multicore and Reconfigurable Instruction Set Extensions," in *Proc. 18th ACM/SIGDA Int'l Symposium on Field Programmable Gate Arrays*, 2010, pp. 33–36.

[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*, 2003, pp. 980–985.

[10] G. Fursin, J. Cavazos, M. O'Boyle, and L. Temam, "MiDataSets: Creating the Conditions for a More Realistic Evaluation of Iterative Optimization," in *Proceedings of the 2nd Int'l Conference on High Performance Embedded Architectures and Compilers*, 2007, pp. 245–260.

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

[12] P. Garcia and K. Compton, "Kernel Sharing on Reconfigurable Multiprocessor Systems," in *Int'l Conference on Field Programmable Technology*, 2008, pp. 225–232.

[13] S. Goldstein, H. Schmit, M. Moe *et al.*, "PipeRench: A Coprocessor for Streaming Multimedia Acceleration," in *Proc. 26th IEEE/ACM Int'l Symposium on Computer Architecture*, May 1999, pp. 28–39.

[14] S. Goldstein, H. Schmit, M. Budiu *et al.*, "PipeRench: A Reconfigurable Architecture and Compiler," *Computer*, vol. 33, pp. 70–77, 2000.

[15] M. I. Gordon, W. Thies, M. Karczmarek *et al.*, "A Stream Compiler for Communication-Exposed Architectures," in *Proc. 10th ACM Symposium on Architectural Support for Programming Languages and Operating Systems*, Oct. 2002, pp. 291–303.

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

[17] S. W. Keckler, W. J. Dally, D. Maskit *et al.*, "Exploiting Fine-Grain Thread Level Parallelism on the MIT Multi-ALU Processor," in *Proc. 25th IEEE/ACM Int'l Symposium on Computer Architecture*, Jun. 1998, pp. 306–317.

[18] C. Lee, M. Potkonjak, and W. Mangione-Smith, "MediaBench: A Tool for Evaluation and Synthesizing Multimedia and Communications Systems," in *Proc. IEEE/ACM 30th Int'l Symposium on Microarchitecture*, 1997, pp. 330–335.

[19] "Livermore Loops Coded in C," http://www.netlib.org/benchmark/livermorec, 2009.

[20] 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*, Nov. 2005, pp. 105–118.

[21] E. Perelman, G. Hamerly, and B. Calder, "Picking Statistically Valid and Early Simulation Points," in *Proc. 12th IEEE/ACM Int'l Conference on Parallel Architectures and Compilation Techniques*, Sept 2003, pp. 244–255.

[22] E. Raman, G. Ottoni, A. Raman *et al.*, "Parallel-Stage Decoupled Software Pipelining," in *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, Apr. 2008, pp. 114–123.

[23] R. Rangan, N. Vachharajani, A. Stoler *et al.*, "Support for High-Frequency Streaming in CMPs," in *Proc. IEEE/ACM 39th Annual Int'l Symposium on Microarchitecture*, Dec. 2006, pp. 259–272.

[24] 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*, Oct. 2004, pp. 177–188.

[25] J. Sampson, R. Gonzalez, J.-F. Collard *et al.*, "Exploiting Fine-Grained Data Parallelism with Chip Multiprocessors and Fast Barriers," in *Proc. IEEE/ACM 39th Annual Int'l Symposium on Microarchitecture*, Dec. 2006, pp. 235–246.

[26] "SESC Architectural Simulator," http://sourceforge.net/projects/sesc, 2007.

[27] S. Shang and K. Hwang, "Distributed Hardwired Barrier Synchronization for Scalable Multiprocessor Clusters," *IEEE Trans. Parallel Distrib. Syst.*, vol. 6, no. 6, pp. 591–605, 1995.

[28] Standard Performance Evaluation Corporation, "SPEC CPU Benchmark Suite," http://www.specbench.org/cpu2000/, 2000.

[29] Standard Performance Evaluation Corporation, "SPEC CPU Benchmark Suite," http://www.specbench.org/cpu2006/, 2006.

[30] W. Thies, M. Karczmarek, and S. P. Amarasinghe, "StreamIt: A Language for Streaming Applications," in *Proceedings of the 11th International Conference on Compiler Construction*, 2002, pp. 179–196.

[31] N. Vachharajani, R. Rangan, E. Raman *et al.*, "Speculative Decoupled Software Pipelining," in *Proc. 16th IEEE/ACM Int'l Conference on Parallel Architectures and Compilation Techniques*, Sep. 2007, pp. 49–59.

[32] M. Watkins and D. Albonesi, "Dynamically Managed Multithreaded Reconfigurable Architectures for Chip Multiprocessors," in *Proc. 19th IEEE/ACM Int'l Conference on Parallel Architectures and Compilation Techniques*, Sep. 2010.

[33] M. Watkins, M. Cianchetti, and D. Albonesi, "Shared Reconfigurable Architectures for CMPs," in *Proc. of the 18th Int'l Conference on Field-Programmable Logic and Applications*, Sep. 2008, pp. 299–304.

[34] 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*, Feb. 2000, pp. 95–100.