ReMAP: A Reconfigurable Architecture for Chip Multiprocessors

ReMAP is a reconfigurable architecture for accelerating and parallelizing applications within a heterogeneous chip multiprocessor (CMP). Clusters of cores share a common reconfigurable fabric adaptable for individual thread computation or fine-grained communication with integrated computation. ReMAP demonstrates significantly higher performance and energy efficiency than hard-wired communication-only mechanisms, and over allocating the fabric area to additional or more powerful cores.

Matthew A. Watkins
Harvey Mudd College

David H. Albonesi
Cornell University

General-purpose processors are intended to run a wide array of applications. As such, it's difficult to determine at design time what hardware to integrate on the chip to provide maximum performance for all applications. One possible solution is to incorporate some amount of reconfigurability so that the chip can dynamically adapt its hardware at runtime to meet the current application's requirements. Past reconfigurable computing proposals have traditionally involved attaching a reconfigurable fabric to a single processor core to accelerate sequential applications. The recent shift to chip multiprocessors (CMPs) and the prospect of large-scale CMPs with tens to hundreds of cores on a die, however, calls for a reexamination of reconfigurable computing from the perspective of multiple cores and multithreaded applications. To this end, we present Reconfigurable Multicore Acceleration and Parallelization (ReMAP), an architecture that both accelerates computation and aids communication among multiple threads in a heterogeneous CMP.

Past reconfigurable architectures have significantly outperformed general-purpose architectures for certain application classes, but at high area and power costs relative to the overall performance achieved across a broad set of applications, some of which 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 logic could 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

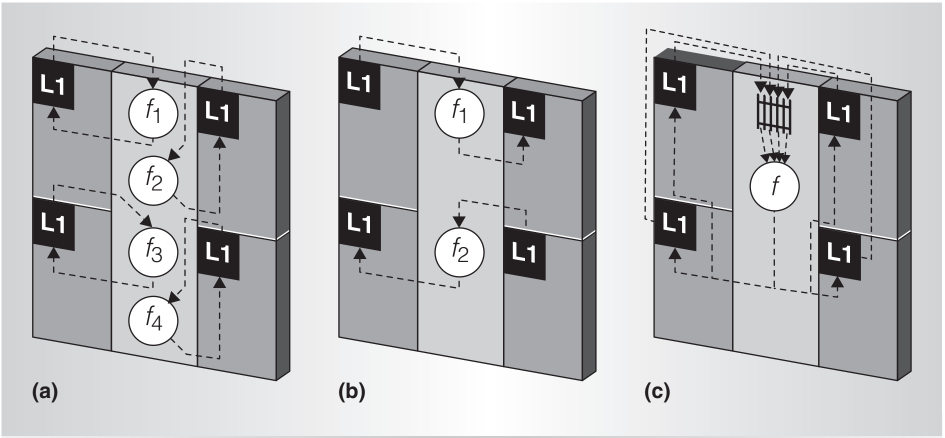


Figure 1. Shared reconfigurable fabric being used for individual computation (a), producer-consumer communication with computation (b), and barrier synchronization with computation (c).

incorporating a reconfigurable fabric decreases, as does the proportion of applications needed to justify the fabric's inclusion. Second, several threads could amortize the fabric's area and power costs by sharing the fabric among multiple cores, thereby forming a cluster of cores plus fabric. With intelligent fabric management, such sharing can increase fabric use 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.[1,2]

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 application optimization that reconfigurable architectures have focused on traditionally. Figure 1 depicts a portion of a heterogeneous CMP and shows the three ways that the ReMAP architecture is dynamically organized to accelerate multiple threads. Figure 1a depicts four threads, each independently performing a function (each function f_i can be unique or identical) within the fabric without communication. In Figure 1b, the fabric is being used for two instances of fine-grained 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 can be queued using any remaining fabric resources if necessary—then passes to the consumer thread. Finally, Figure 1c depicts four threads synchronizing at a barrier within the fabric with a global function (such as a global minimum) computed in the fabric after the synchronization point.

Unlike previous proposals, ReMAP supports multiple communication models and also allows customized computation on communicated data. The latter provides performance improvements beyond what's possible with previous communication-only options and traditional fixed computation alternatives.

ReMAP architecture

ReMAP pairs a specialized programmable logic (SPL) fabric with multiple cores of a CMP. Figure 2a shows an example ReMAP heterogeneous CMP with integrated SPL. The figure shows a 16-core ReMAP CMP with two SPL clusters on the left. Each cluster consists of four single-issue out-of-order processor cores sharing an SPL fabric, which Figure 2b shows at a high level. The cores in the same cluster temporally share the fabric in a round-robin fashion, and the SPL controller can spatially partition the fabric as needed to reduce interthread contention. ReMAP further reduces contention by limiting the degree of fabric

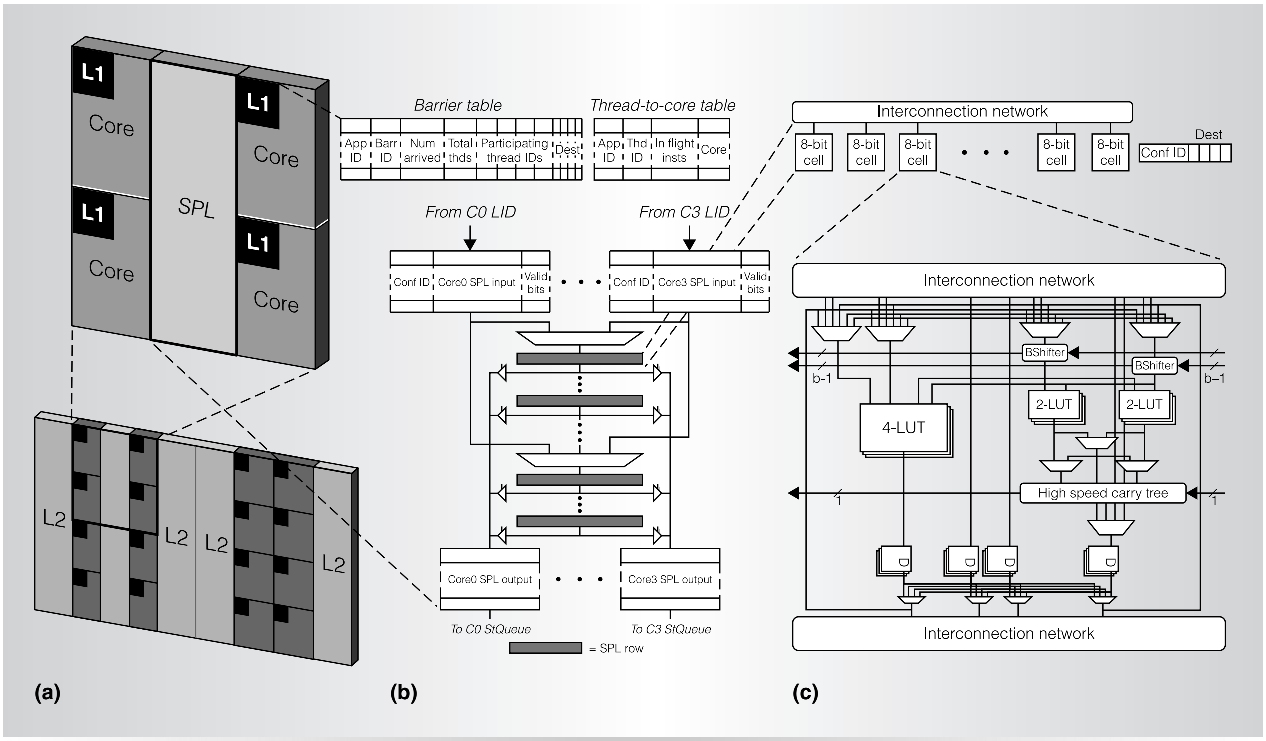


Figure 2. ReMAP heterogeneous chip microprocessor (CMP). The overall ReMAP chip, with two specialized programmable logic (SPL) clusters and one conventional cluster, and a blow-up of one SPL cluster (a); a four-way shared SPL including tables required for communication (b); and design of an SPL row and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

sharing. In this particular example, the proportion of applications benefiting 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 could be several SPL clusters as well as many other different cluster types, such as the traditional multicore cluster shown at the right of Figure 2a, on the die. Applications execute on an SPL cluster during periods of the program that use the fabric and execute on other clusters during other periods to obtain the best overall performance.

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

	SPL* rows	Total area	Peak dynamic power	Total leakage power
Four cores	N/A	1.00	1.00	1.00
Four-way shared SPL	24	0.51	0.14	0.67

*SPL: specialized programmable logic.

SPL organization

ReMAP's computational substrate is a highly pipelined, row-based SPL.[2] As Figure 2c shows, each row comprises 16 cells, and each cell computes 8 bits of data. Each cell in a row can perform a different operation on its set of inputs, and 24 of these rows group together to form the overall SPL fabric. The SPL is clocked at a fixed 500 MHz, one-quarter of the 2-GHz core frequency. Table 1 shows the relative area and power consumption of the SPL and associated single-issue cores, derived using methodology from Watkins et al.[2]

The fabric's row-based nature allows hardware requirements to be indicated by the number of rows needed to implement a function. If the number of rows a function requires exceeds the physical number on chip, the fabric controller can virtualize the function over the fabric. 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 execute, even if fewer rows are available than originally anticipated.

The SPL serves as a reconfigurable complex functional unit that can dynamically change its operation to meet the currently running application's needs. Special load and store instructions move data to and from the fabric's input and output queues (see Figure 2b), respectively. Figure 2b also shows the barrier and thread-to-core tables, input queue valid bits, and row destination IDs that support interthread and barrier communication.

Support for fine-grained communication plus computation

Most parallel applications use some form of communication to coordinate their activity. At a high level, communication requires threads to exchange information, be it a notification that a thread is arriving at a barrier or a producing thread passing results to a consuming thread. ReMAP facilitates fine-grained 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.

Fine-grained interthread communication plus computation. Hardware-accelerated interthread communication lets threads communicate with each other much more frequently than would be possible using the traditional memory system. Such fine-grained communication typically targets pipelined applications where one thread produces data that is sent to a consumer for further processing. 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. The queue helps decouple the two threads such that, unless the queue is full or empty, the two threads can continue to produce or consume data without concern for the other thread's progress.

Because multiple cores share the fabric, sending data to a different core simply requires sending the fabric output to the consuming core's output queue. The input and output queues provide queueing slots and the pipelined fabric both performs computation and provides 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 3a). Once all the necessary data is loaded, the producer issues an SPL instruction (Figure 3b). The data progresses through the SPL to perform the programmed computation. Once any computation is complete, the results are bypassed to the consuming core's output queue (Figure 3c). Finally, the consuming core stores the data from the output queue to memory (Figure 3d).

Because the actual assignment of threads to cores is determined at runtime, a small table (the thread-to-core table in Figure 2b) maintains a mapping of threads to cores so that the appropriate destination core receives the queued data. 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.

Barrier synchronization plus computation. A barrier is a common synchronization operation that ensures that all participating threads have reached a certain point in the code before any can continue execution. The typical memory-based implementation, however, has significant performance overhead that prevents a barrier's use at fine granularities. Specialized hardware implementations[3,4] can reduce this overhead, allowing parallelization of applications that wouldn't be possible otherwise. In addition to providing such specialized barrier support, ReMAP can further synthesize computation associated with a barrier—such as a reduction—into the fabric and send the output to all participating threads.

To implement barriers in ReMAP, SPL barrier instructions 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

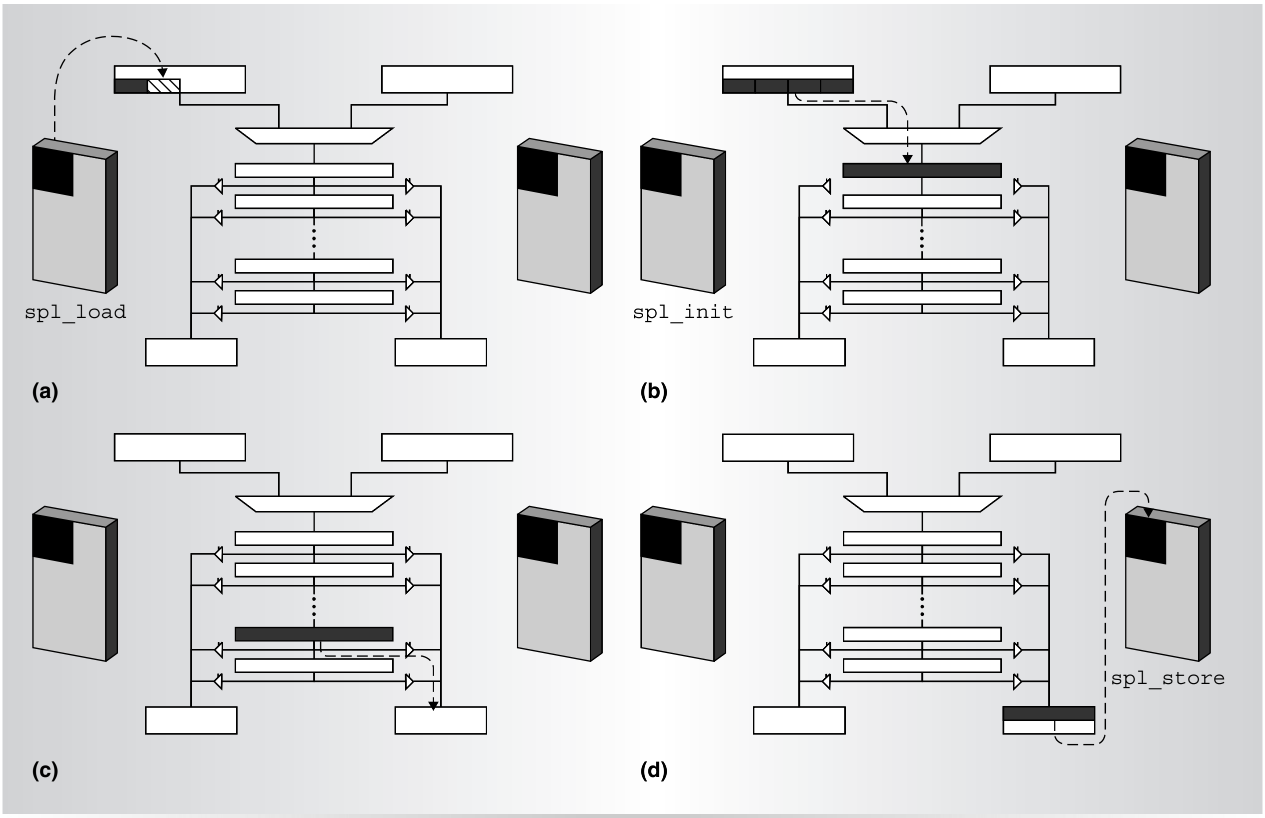


Figure 3. A walkthrough of intercore communication with integrated computation: the producing thread loads input data (a); the producer issues an SPL instruction (b); the SPL performs the computation and outputs the results (c); and the consuming core stores the data to memory (d).

loads some value(s) into its SPL input queue (part 1 in Figure 4a). Once all of the cores' loaded values have reached the head of their respective input queues and all threads have indicated arrival at the barrier (Figure 4b), the SPL controller issues an instruction to the fabric, and all loaded values pass into the fabric (Figure 4c). 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 each participating processor's output queue (Figure 4d), and the processor stores the data appropriately. 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 2b) with information related to each active barrier. The table tracks the total number of threads, the number of arrived threads, and the cores participating in the barrier. The SPL updates the number of arrived threads and participating cores each time a thread arrives (parts 2 through 5 in Figures 4a and 4b), and it compares the total and arrived thread counts to determine when to issue an instruction (part 6 in Figure 4b).

In a system with multiple SPL clusters, a dedicated bus communicates barrier updates among clusters. The bus is small, communicating only the barrier and associated application ID when a new thread arrives at a barrier.

Communication examples

We propose using the SPL to perform both fine-grained interthread communication and fine-grained barrier synchronization. This section shows example applications

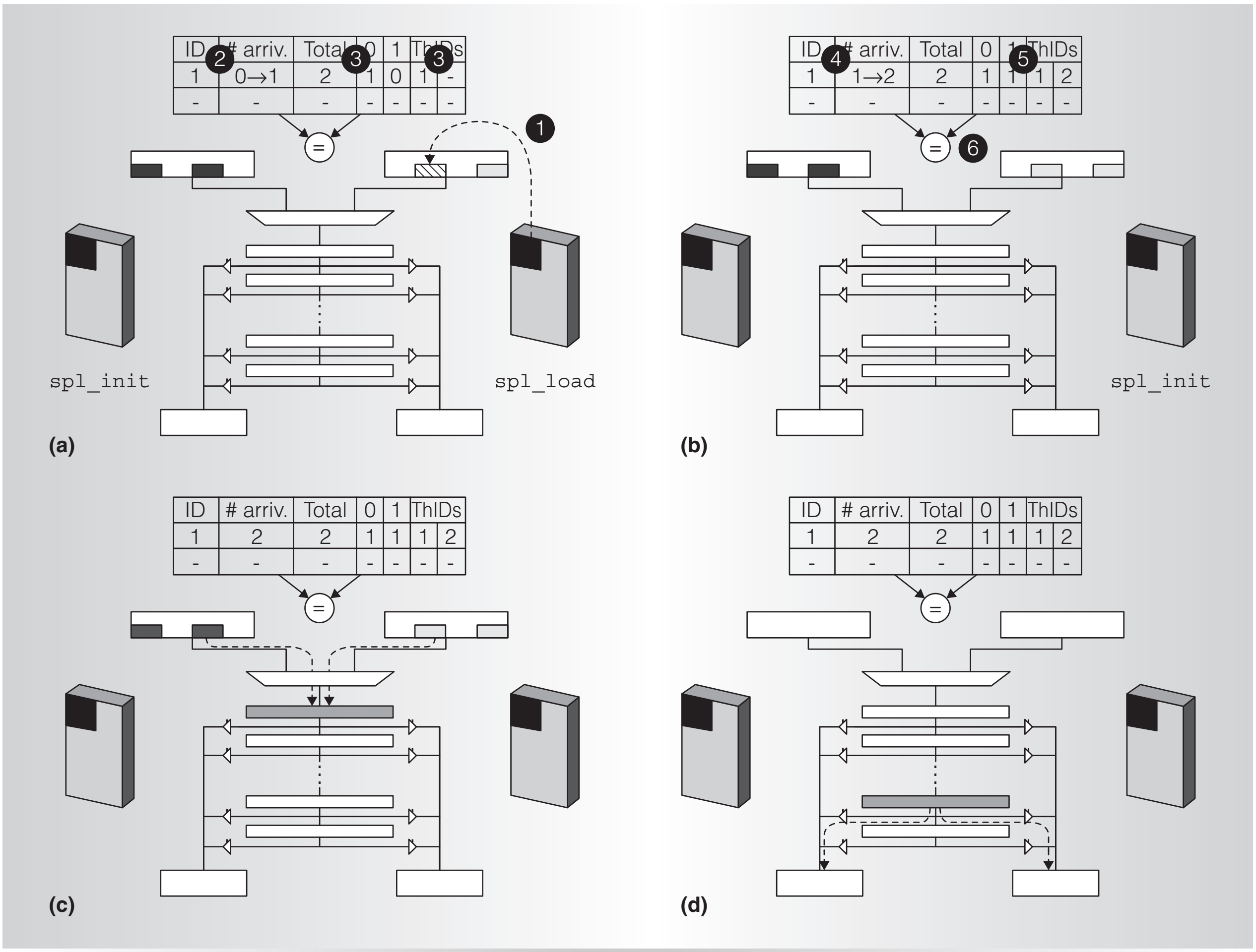


Figure 4. A walkthrough of barrier synchronization with integrated computation: core on left arrives at barrier while core on right continues to load data into its input queue (a); core on right arrives at barrier (b); SPL issues instruction to fabric (c); results are written to output queue of participating cores (d).

that benefit from the enhanced communication and from performing computation in the SPL during the communication.

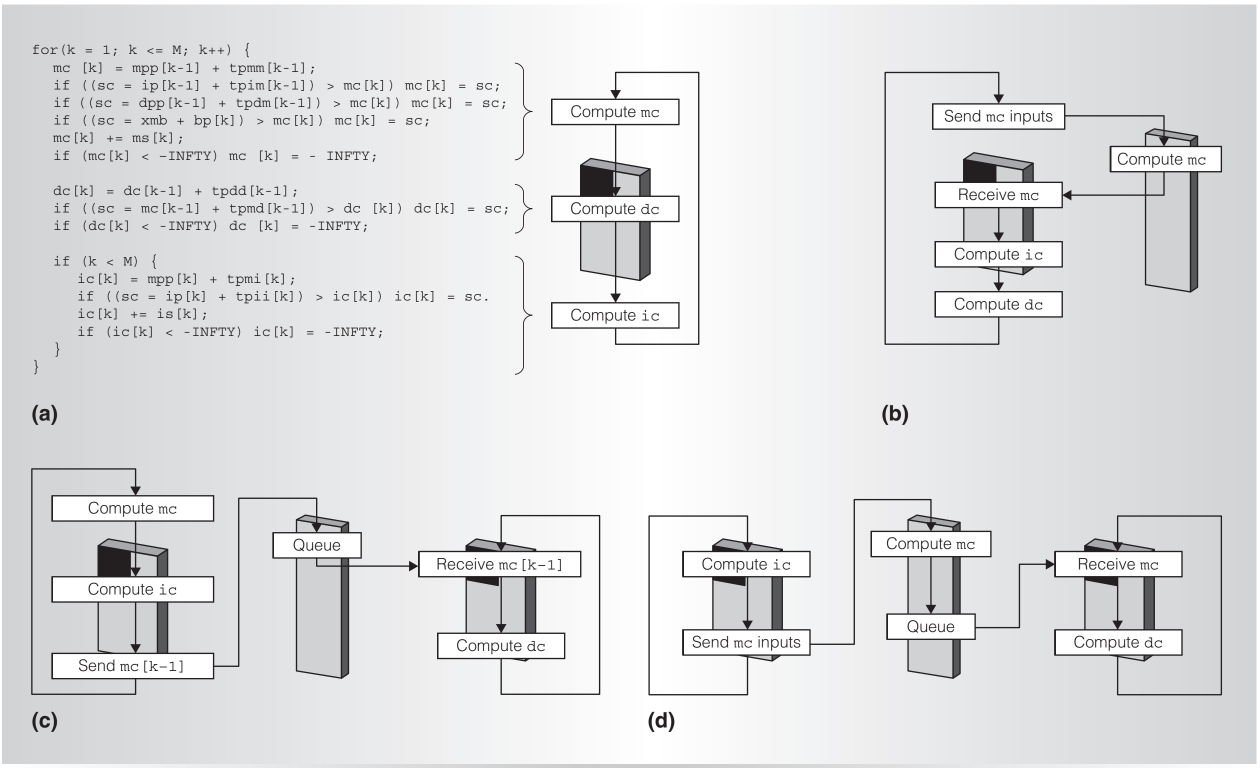
Interthread communication plus computation example

To illustrate interthread communication, we show an example parallelization of the SPEC 2006 application 456.hmmer. We optimize the `P7Viterbi` function's inner loop, which implements the dynamic programming Viterbi algorithm. Figure 5a shows the optimized section's original code, along with a flowchart summarizing the computation being performed.

We first look at how the SPL can accelerate a portion of the computation, specifically the calculation of `mc`. As Figure 5b shows, the core loads the input values needed to compute `mc` into the fabric, the SPL computes `mc`'s value, and the core receives the result. After receiving `mc`, the core computes the values of `dc` and `ic` and repeats the loop.

The next implementation creates a producer-consumer thread pair that uses the SPL solely for communication (Figure 5c). The producer thread calculates the values of `mc` and `ic` and sends `mc`'s value from the previous iteration to the consumer through the SPL. The consumer receives this value and uses it to compute `dc`.

Finally, Figure 5d shows how the SPL can combine computation and communication.



```
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;
  }
}
```

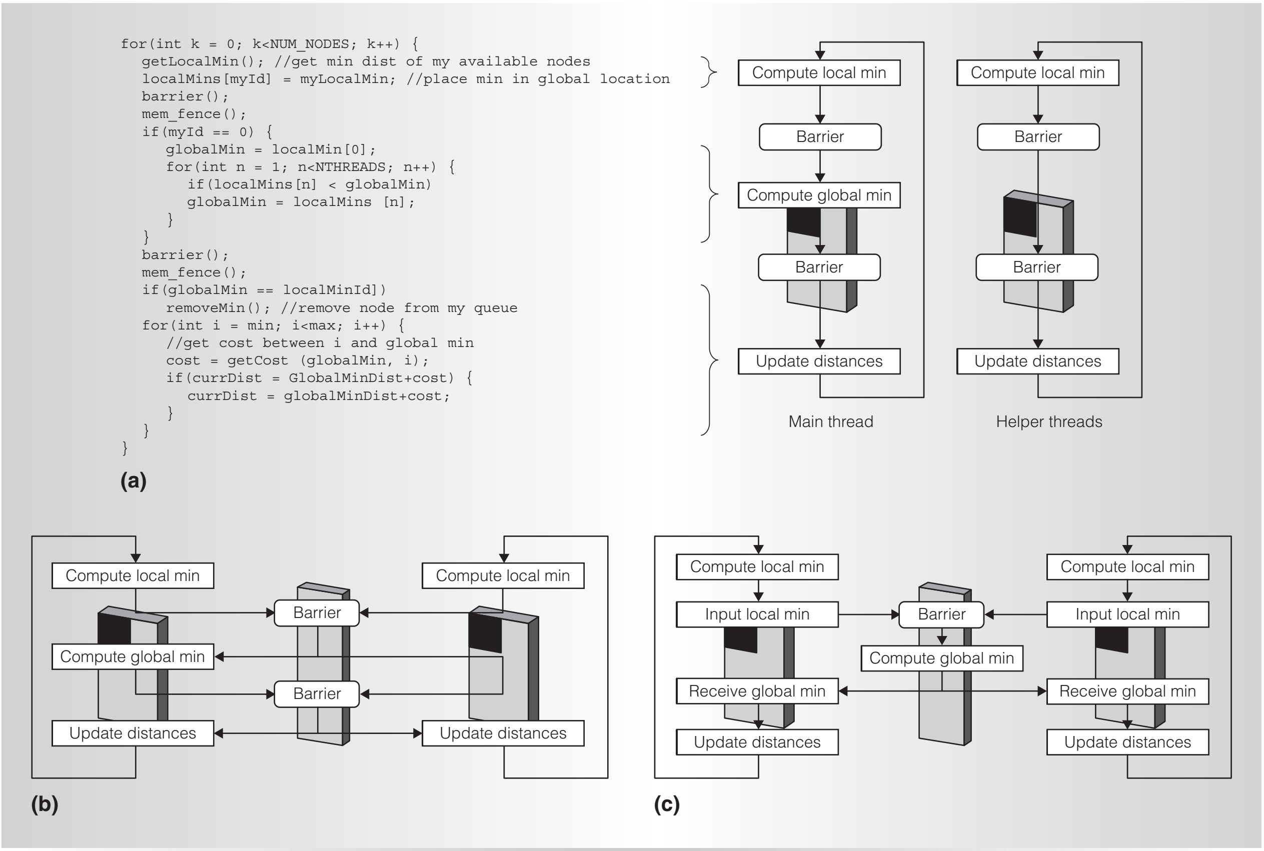
Figure 5. Parallelization of SPEC 2006 456.hmmer `P7Viterbi`. Original sequential version of code and associated flowchart (a) and flowcharts using the SPL for computation only (b), for communication only (c), and for communication plus computation (d).

The producer thread computes `ic` and loads the inputs needed for `mc`. The SPL computes `mc`'s value 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 producer's work load, which better balances the threads and improves the performance of the parallelization.

Barrier synchronization plus computation example

To show how ReMAP barrier synchronization operates, we consider a parallel version of Dijkstra's algorithm. Generically, at each iteration, Dijkstra's algorithm selects the smallest unvisited node in the graph and updates the distances to all other unvisited nodes. In the parallel version, each thread receives a portion of the entire graph to maintain. Figure 6a shows the basic parallel algorithm and the main and helper threads' high-level flow. 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. In the final section, 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 (see Figure 6b). As with previous dedicated barrier techniques,[3,4] 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 SPL's computational power to compute the global minimum within the fabric. Figure 6c shows this optimization for the case where all threads share a single SPL cluster. Each thread computes its local minimum as before and loads this value into the SPL. While performing the barrier,



```
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 == localMinId])
    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;
    }
  }
}
```

Figure 6. Parallelization of Dijkstra's algorithm. Original parallel code with software barriers and associated flowchart (a), flowchart for SPL barriers only (b), and flowchart for SPL barriers with integrated computation (c).

the SPL computes the minimum input value. Each participating core receives the global minimum from the SPL and updates the distances for its nodes. The SPL outputs the global minimum directly, thereby eliminating one barrier.

If the threads spread across multiple clusters, the operation occurs over multiple stages. The first stage computes regional minimum values (minimum values of all cores in a single cluster). All threads resynchronize at a second barrier, and the fabric computes the global minimum at a final barrier. Despite the extra barrier, performance still improves over using ReMAP for communication only.

Evaluation methodology

We evaluated ReMAP with a cycle-accurate, execution-driven simulator. We assumed processors implemented in 65-nm technology running at 2 GHz.[5] We used benchmarks from SPEC 2000, SPEC 2006, MediaBench, MiBench, and Livermore loops suites along with the Unix utility wc to analyze the three usage modes shown in Figure 1. To evaluate barrier synchronization, we created parallel versions of Dijkstra's algorithm and Livermore loops 2, 3, and 6. 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 occurs during the synchronization operation (as in Figure 1c). Livermore loop 3 uses two ReMAP operation modes: performing computation on the data within the loop (Figure 1a) and using the SPL to accelerate synchronization between iterations (Figure 1c).

Results

We first evaluated ReMAP in the context of a heterogeneous CMP executing entire

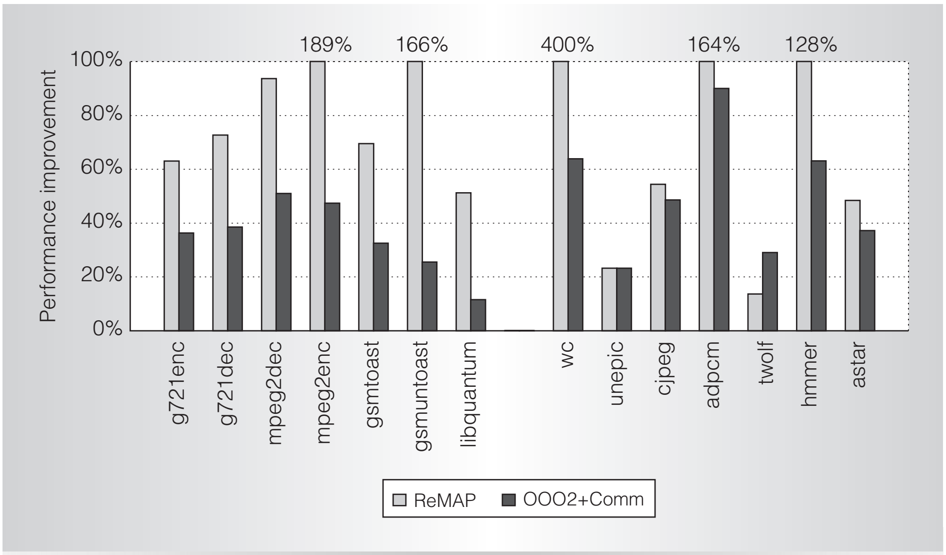


Figure 7. Performance improvement of ReMAP and OOO2+Comm relative to the single-threaded baseline.

programs and then evaluated the performance of the optimized regions to show the sources of the improvements.

ReMAP in a heterogeneous CMP

We compared a ReMAP system composed of clusters of four single-issue out-of-order cores (OOO1) plus a 24-row SPL, coupled with clusters of dual-issue out-of-order cores (OOO2) as shown in Figure 2a to an alternative composed of clusters of four OOO2 cores with a dedicated hardware communication network, similar to previous proposals[6,7] (OOO2+Comm). 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 use the SPL run on the SPL cluster while other regions run on an OOO2 core. We compare these two schemes using workloads that employ the SPL for computation alone and those that utilize the SPL for computation plus communication (we discuss results for barrier synchronization separately).

Figure 7 shows the two configurations' performance improvement relative to executing the original sequential code on an OOO1 core. ReMAP performs as well as or better than OOO2+Comm in all but one case and performs 45 percent better on average. In the one exception, twolf, the sequential regions' duration is so short that the time lost in moving the thread to the faster OOO2 core outweighs the benefits received from executing the sequential regions on the faster core.

As a measure of ReMAP's energy efficiency, Figure 8 shows energy×delay for the two configurations relative to the single-threaded baseline. ReMAP provides the best energy efficiency in all but one case. The one exception is again twolf, where both communication alternatives achieve worse energy×delay than the baseline, indicating that twolf should be run as a single thread on a simple core if energy is a significant concern. With the exception of twolf, ReMAP provides 45 percent better performance and 35 percent lower energy consumption on average than the more powerful cores and dedicated communication network.

Analysis of optimized regions

To see the sources of these improvements, we analyzed just the code regions optimized for ReMAP. We compared the performance of using the SPL for computation only and (where appropriate) using the SPL for communication only and using the SPL for

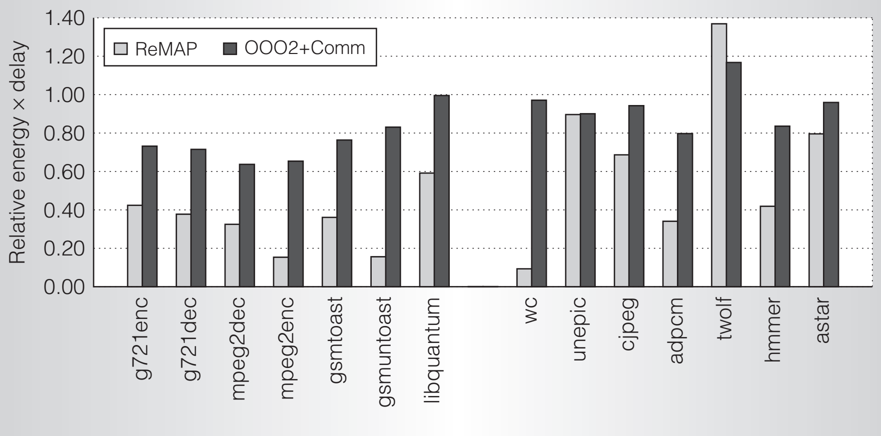


Figure 8. Energy×delay of ReMAP and OOO2+Comm relative to the single-threaded baseline.

computation plus communication to OOO2 cores with idealized communication hardware (OOO2+Comm). It's only with the combination of computation and communication that ReMAP outperforms the OOO2+Comm alternative in all cases (by 79 percent on average), showing the clear benefit of integrating computation and communication in the SPL.

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

Splitting the application into a producer-consumer pair can place sections of code with unpredictable loads or branches in their own thread to reduce or eliminate their impact on performance. By placing just this code in one thread and the rest in the other, the first thread can start processing these unpredictable instructions earlier. This reduces the impact of these instructions' unpredictability and improves performance. Furthermore, splitting a thread into a producer-consumer pair makes each core responsible for approximately half of the SPL-related instructions (either the loads or the stores). This reduces the number of instructions that both threads need to process, which can reduce pressure on hardware resources and improve performance.

Performing computation in the SPL while data is 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, reducing the amount of time spent waiting on a full or empty SPL queue and improving performance. Removing instructions from one or both of the threads can also reduce pressure on hardware resources, again improving performance. Finally, moving computation inside the SPL can improve branch prediction in one or both threads by moving conditional operations into the SPL. The improved branch prediction improves processor efficiency, which again improves performance.

Fine-grained barrier synchronization

We compared ReMAP barrier performance (with and without computation where appropriate) to traditional software barriers when running two, four, eight, and 16 threads. Figure 9 shows the performance for the eight- and 16-threaded cases. Results for two and four threads show similar trends. Similar to other fine-grained synchronization techniques,[3,4] performing barriers via ReMAP significantly improves performance

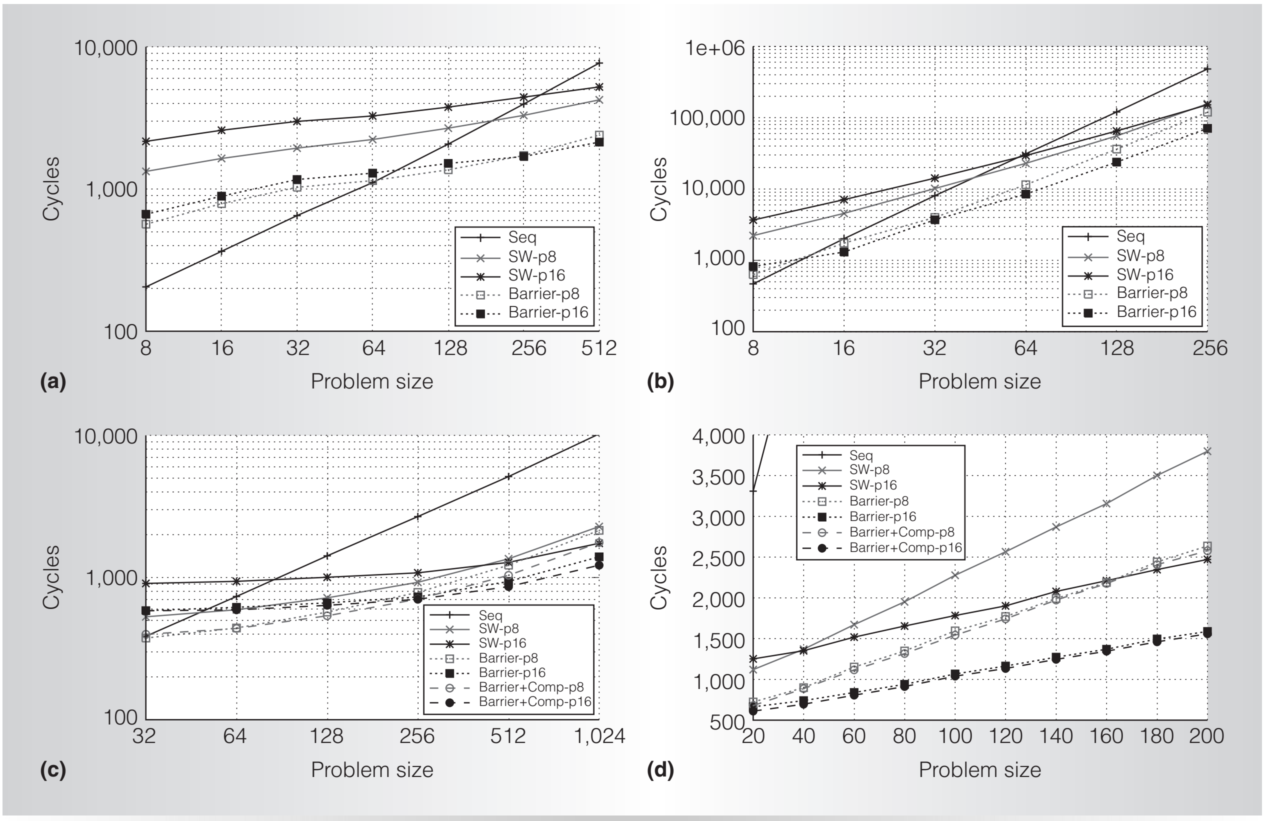


Figure 9. Per-iteration execution time for Livermore loops 2 (a), 6 (b), and 3 (c) and for Dijkstra's algorithm (d) of sequential code and parallel versions of code with software and ReMAP barriers and, where appropriate, ReMAP barrier with integrated computation.

over software barriers and makes larger thread counts useful for smaller problem sizes.

Fine-grained barrier synchronization with SPL computation. Certain parallel benchmarks benefit from the SPL's computational capabilities (Barrier+Comp in Figure 9). The SPL computation occurs either as part of the barrier operation, as in dijkstra, or in a separate SPL function that only performs computation, as in Livermore loop 3.

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

In Livermore loop 3, where the computation is a separate function, the greatest benefit occurs with fewer threads and larger problem sizes. In either of these cases, each thread has more work to do between barriers, resulting in the computation section (the SPL-accelerated part) consuming a larger percentage of the overall execution time. Improvement ranges from 15 percent to 26 percent for large problem sizes.

Energy efficiency. Despite the additional energy the SPL consumes, ReMAP barriers always achieve better energy×delay than their software counterparts. For both options, the energy×delay break-even point—the point at which the parallel code starts outperforming the sequential case—requires larger problem sizes than the performance break-even point. This occurs because, especially at very fine granularities, the performance improvement achieved by increasing the number of threads isn't ideal (that is, doubling the number of threads doesn't halve the runtime).

With the end of single processor performance scaling, the question arises of how to best use the additional transistors provided by Moore's Law. Currently, the additional transistors are used to incorporate multiple cores on a single die. While such homogeneous CMPs provide a partial solution, their single, generic design limits their power-performance efficiency when considering any individual application. Homogeneous CMPs' performance benefit is further impeded by the fact that not all applications can easily be parallelized to use the many cores, essentially providing zero performance growth for these applications.

An alternative use of these transistors is to incorporate specialized hardware accelerators tailored to different applications on chip. Incorporating multiple accelerators, however, makes the already challenging design and verification process even more difficult. Furthermore, an individual accelerator can perform only a limited set of computations and has limited ability to adapt as algorithms change.

A reconfigurable architecture like ReMAP can help address all these issues. Fine-grained communication lets us parallelize otherwise sequential regions of code, and its single design can accelerate a range of computations. Reconfigurable logic, however, isn't a global solution. While certain applications see significant benefits from reconfigurability, others see no benefit at all. For this reason, we propose a heterogeneous CMP that incorporates reconfigurable logic in only a portion of the chip.

Reconfigurable computing is gaining increased attention from industry.[8,9] Our work demonstrates that judiciously integrating a reconfigurable fabric on die can benefit a relatively broad range of applications. A key result is that ReMAP is useful for more than just straight computation acceleration. This additional functionality provides further incentive to incorporate reconfigurability on chip. Future work is needed to evaluate other organizations for reconfigurable logic in CMPs and to investigate other potential nonstandard uses of integrated reconfigurable fabrics. MICRO

References

1. M.A. Watkins and D.H. Albonesi, "Dynamically Managed Multithreaded Reconfigurable Architectures for Chip Multiprocessors," *Proc. 19th IEEE/ACM Int'l Conf. Parallel Architectures and Compilation Techniques,* ACM Press, 2010, pp. 41-52.
2. M.A. Watkins, M. Cianchetti, and D.H. Albonesi, "Shared Reconfigurable Architectures for CMPs," *Proc. 18th Int'l Conf. Field-Programmable Logic and Applications,* IEEE Press, 2008, pp. 299-304.
3. C.J. Beckmann and C.D. Polychronopoulos, "Fast Barrier Synchronization Hardware," *Proc. 1990 ACM/IEEE Conf. Supercomputing,* IEEE CS Press, 1990, pp. 180-189.
4. J. Sampson et al., "Exploiting Fine-Grained Data Parallelism with Chip Multiprocessors and Fast Barriers," *Proc. IEEE/ACM 39th Ann. Int'l Symp. Microarchitecture,* IEEE CS Press, 2006, pp. 235-246.
5. M.A. Watkins and D.H. Albonesi, "ReMAP: A Reconfigurable Heterogeneous Multicore Architecture," *Proc. 43rd Ann. IEEE/ACM Int'l Symp. Microarchitecture,* ACM Press, 2010, pp. 497-508.
6. E. Caspi et al., "Stream Computations Organized for Reconfigurable Execution," *Proc. Roadmap to Reconfigurable Computing, 10th Int'l Workshop Field-Programmable Logic and Applications,* Springer, 2000, pp. 605-614.
7. R. Rangan et al., "Decoupled Software Pipelining with the Synchronization Array," *Proc. 13th IEEE/ACM Int'l Conf. Parallel Architectures and Compilation Techniques,* IEEE CS Press, 2004, pp. 177-188.
8. J.S. Emer, "An Evolution of General Purpose Processing: Reconfigurable Logic Computing," *Proc. 7th Ann. IEEE/ACM Int'l Symp. Code Generation and Optimization,* IEEE CS Press, 2009, doi:10.1109/CGO.2009.38.
9. M. Feldman, "Reconfigurable Computing Prospects on the Rise," *HPCwire,* 3 Dec. 2008; www.hpcwire.com/features/Reconfigurable-Computing-Prospects-on-the-Rise_35498449.html.

Matthew A. Watkins is a visiting assistant professor in the Engineering Department at Harvey Mudd College. His research interests include reconfigurable architectures for chip multiprocessors and the design of large multicore systems. He received his PhD in electrical and computer engineering from Cornell University. He's a member of IEEE and the ACM.

David H. Albonesi is a professor in the School of Electrical and Computer Engineering and a member of the Computer Systems Laboratory at Cornell University. His research interests include adaptive and reconfigurable architectures, power- and reliability-aware computing, and high-performance interconnect architectures using silicon nanophotonics. He received his PhD in computer engineering from the University of Massachusetts, Amherst. He's a Fellow of IEEE.

Direct questions and comments to Matthew A. Watkins, Engineering Dept., Harvey Mudd College, 301 Platt Blvd., Claremont, CA 91711; mwatkins@csl.cornell.edu.