DOI:10.1145/3139461

Automatically Accelerating Non-Numerical Programs by Architecture-Compiler Co-Design

By Simone Campanoni, Kevin Brownell, Svilen Kanev, Timothy M. Jones, Gu-Yeon Wei, and David Brooks

Abstract

Because of the high cost of communication between processors, compilers that parallelize loops automatically have been forced to skip a large class of loops that are both critical to performance and rich in latent parallelism. HELIX-RC is a compiler/microprocessor co-design that opens those loops to parallelization by decoupling communication from thread execution in conventional multicore architecures. Simulations of HELIX-RC, applied to a processor with 16 Intel Atom-like cores, show an average of 6.85× performance speedup for six SPEC CINT2000 benchmarks.

1. INTRODUCTION

On a multicore processor, the performance of a program depends largely on how well it exploits parallel threads. Some computing problems are solved by numerical programs that are either inherently parallel or easy to parallelize. Historically, successful parallelization tools have been able to transform the sequential loops of such programs into parallel form, boosting performance significantly. Most software, however, is still sequentially designed and largely non-numerical, with irregular control and data flow. Because manual parallelization of such software is error-prone and time-consuming, automatic parallelization of non-numerical programs remains an important open problem.

The last decade has seen impressive steps toward a solution, but when targeting commodity processors, existing parallelizers still leave much of the latent parallelism in loops unrealized.[5] The larger loops in a program can be so hard to analyze accurately that apparent dependences often flood communication channels between cores. Smaller loops are more amenable to accurate analysis, and our work shows that there is a lot of parallelism between the iterations of small loops in non-numerical programs represented by SPECint2000 benchmarks.[4] But even after intense optimization, small loops typically include loop-carried dependences, so their iterations cannot be entirely independent—they must communicate. Because the iterations of a small loop are short (25 clock cycles on average for SPECint2000), their communications are frequent.

On commodity processors, communication relies on the memory system and is reactive, triggered only when one core asks for data from another. The resulting delay is longer than the average duration of an iteration, and it is hard to overlap with computation, especially when the variance of durations is high, as in non-numerical workloads. The benefits of automatic loop parallelization therefore saturate at small numbers of cores for commodity processors.

Lowering the latency of inter-core communication would help, but it can only go so far, if communication remains reactive. We therefore propose a proactive solution, in which the compiler and an architectural extension called *ring cache* cooperate to overlap communication with computation and lower communication latency. The compiler identifies data that must be shared between cores, and the ring cache circulates that data as soon as it is generated.

To demonstrate this idea, we have developed HELIX-RC, a co-design incorporating a parallelizing compiler and a simulated chip multiprocessor extended with ring cache. The HELIX-RC compiler builds on the original HELIX code parallelizer for commodity multicore processors.[5] Because it relies on invariants of the code produced by the compiler, ring cache is a lightweight, non-invasive extension of conventional multicore architecture. Because it facilitates proactive, low-latency inter-core communication, ring cache allows HELIX-RC to outperform HELIX by a factor of 3×.

2. OPPORTUNITIES AND CHALLENGES OF SMALL LOOPS

2.1. Opportunities

Prior loop parallelization techniques have avoided selecting loops with small bodies because communication would slow down execution on conventional processors.[5, 20] On average, such techniques yield only about 60% coverage by parallelized loops for non-numerical programs. Excluding small loops limits overall speedup of such programs to less than 3 times no matter how many cores are available, because by Amdahl's law, coverage dictates the overall speedup of a program through parallelization.

Because the intricacy of control and data flow scales down with code size, small loops are easier than larger ones for a compiler to analyze, which reduces the proportion of data dependences that must be accommodated at run time because of conservative assumptions about possible pointer aliases. As a result, the optimized bodies of small loops yield relatively independent iteration threads.[5] So there could be

The original paper, "HELIX-RC: An Architecture-Compiler Co-Design for Automatic Parallelization of Irregular Programs," was published in *Proceedings of the International Symposium on Computer Architecture*, June 14–18, 2014, 217–228.

a significant increase in core utilization, and concomitant overall speedup, if the compiler were able to freely select small hot loops for parallelization. Realizing that potential requires understanding the characteristics of such loops and optimizing for them.

2.2. Low latency challenge

To illustrate the need for low-latency communication, Figure 1a plots a cumulative distribution of average iteration execution times on a single Atom-like core (described in our work[4]) for the set of hot loops from SPECint2000 chosen for parallelization by HELIX-RC. The shaded portion of the plot shows that more than half of the loop iterations complete within 25 clock cycles. The figure also shows the measured core-to-core round trip communication latencies for 3 modern multicore processors. Even the shortest of these latencies, 75 cycles for Ivy Bridge, is too heavy a communication penalty for the majority of these short loops.

2.3. Broadcast challenge

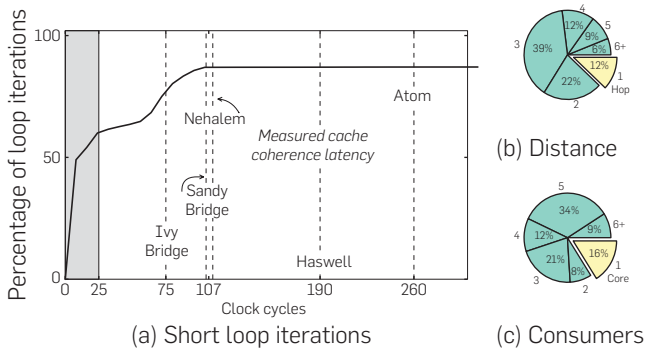
Loops within non-numerical programs generate values that are consumed by later iterations, but the compiler cannot know which iterations will use which values. So when the compiler distributes the iterations of a loop to separate cores, shared values that result from loop-carried dependences need to be accessible to any of those cores soon after being generated.

For loops parallelized by HELIX-RC, most communication of shared values is not between cores executing successive loop iterations, which HELIX-RC assigns to adjacent cores. Figure 1b charts the distribution of value communication distances (defined as the undirected distance between the core that produces a value and the first one that consumes it) on a platform with 16 cores organized in a ring. Only 15% of those transfers are between adjacent cores.

Moreover, Figure 1c shows that most (86%) of the loop-carried values in parallelized loops are consumed by multiple cores. Since consumers of shared values are not known at compile time, especially for non-numerical workloads, broadcasting is the most appropriate communication protocol.

It is well known that implementing low-latency broadcast is challenging for a large set of cores. HELIX-RC uses a hardware mechanism that achieves proactive delayed broadcast of data and signals to all cores in the ring for a loop. Such proactive communication is the cornerstone of the HELIX-RC approach, which allows the communication needed for sharing data between cores to overlap with computation that the cores carry out in parallel.

Figure 1. Small hot loops have short iterations that send data over multiple hops and to multiple cores.



3. THE HELIX-RC SOLUTION

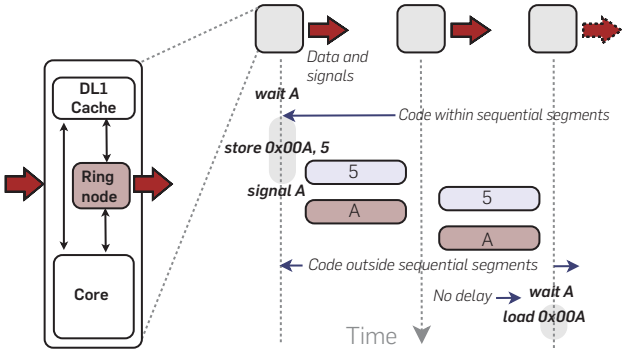
To run the iterations of small hot loops efficiently in parallel, HELIX-RC replaces communication-on-demand with proactive communication. It decouples value forwarding between threads from value consumption by the receiving thread. It also decouples transmission of synchronizing signals from the code that enforces sequential semantics. Extensions of conventional microprocessor architecture make this decoupling possible. Reliance on compiler-guaranteed machine code properties keeps those architectural extensions simple and efficient.

3.1. Approach

HELIX-RC is a co-design that binds its compiler (*HCCv3*) to a processor architecture enhancement called *ring cache*. When the compiler generates a set of parallel threads to run on separate cores, they are rarely completely independent. While most of each thread's code can run concurrently with other threads, there are segments of that code that must execute in strict sequence across the thread set. We call these *sequential segments*. The main role of the ring cache is to accelerate the communication of values and synchronizing signals needed to implement sequential segments correctly.

The ring cache is a ring network linking *ring nodes*, each of which is attached to a core in the processor. During sequential segments, this ring serves as a distributed first-level cache preceding the private L1 cache (Figure 2). HCCv3 marks the entry and exit points of sequential segments using 2 instructions that extend the instruction set. As a result, each core knows whether or not it is currently executing the sequential segment of a parallel thread, and it accesses the cache hierarchy accordingly.

Figure 2. The ring cache is a ring network that connects *ring nodes* attached to each core. It operates during sequential segments as a distributed first-level cache that precedes the private L1 cache (left side). Ring nodes propagate newly-generated values without involving their attached cores (right side). In this example, data generated by the rightmost core is available at the leftmost core when needed, so `wait A` incurs no delay.



Compiler. HCCv3 automatically generates parallel threads from sequential programs by distributing successive loop iterations across adjacent cores organized as a unidirectional ring within a single multicore processor. HCCv3 parallelizes loops that are most likely to speed up performance when their iterations execute in parallel. Only 1 loop runs in parallel at a time.

To preserve the sequential semantics of the original loop, the code that implements a loop-carried data dependence, that is, one spanning loop iterations, must run in a sequential segment whose instances in parallel threads execute in iteration order. Variables and other data structures involved in such dependences—even those normally allocated to registers in sequential code—are mapped to specially-allocated memory locations shared between cores. HCCv3 guarantees that accesses to those shared memory locations always occur within sequential segments.

ISA. We introduce a pair of instructions—`wait` and `signal`—that mark the beginning and end of a sequential segment. Each has an integer operand that identifies the particular sequential segment. A `wait 3` instruction, for example, blocks execution of the core that issues it until all other cores running earlier iterations have finished executing the sequential segment labeled `3`, which they signify by executing `signal 3`. Figure 2 shows a sequential segment with label `A` being executed by the core attached to the leftmost ring node. Between `wait A` and `signal A`, a `store` instruction sends the new value `5` for the shared location at address `0x00A` to the ring node for caching and circulation to its successor nodes. The `signal A` instruction that ends the segment also signals subsequent nodes that the value generated by segment `A` is ready.

A core forwards all memory accesses within sequential segments to its local ring node. All other memory accesses (not within a sequential segment) go through the private L1 cache.

Memory. Each ring node has a cache array that satisfies both loads and stores received from its attached core during a sequential segment. HELIX-RC does not require other changes to the existing memory hierarchy because the ring cache orchestrates interactions with it. To avoid any changes to conventional cache coherence protocols, the ring cache permanently maps each memory address to a unique ring node. All accesses from the distributed ring cache to the next cache level (L1) go through the associated node for a corresponding address.

3.2. Overlapping communication with computation

Because shared values produced by a sequential segment and the signal that marks its end are propagated through the ring node as soon as they are generated, this communication between iterations is decoupled from computation taking place on the cores.

Shared data communication. Once a ring node receives a store, it records the new value and proactively forwards its address and value to an adjacent node in the ring cache, all without interrupting the execution of the attached core. The value then propagates from node to node through the rest of the ring without interrupting the computation of any core.

Synchronization. Given the difficulty of determining which iteration depends on which in non-numerical programs, compilers typically make the conservative assumption that an iteration depends on all of its predecessor iterations. Therefore, a core cannot execute sequential code until it is unblocked by its predecessor.[5, 20] Moreover, an iteration unblocks its successor only if both it and its predecessors have executed this sequential segment or if they are not going to. This execution model leads to a chain of signal propagation across loop iterations that includes unnecessary synchronization: even if an iteration is not going to execute sequential code, it still needs to synchronize with its predecessor before unblocking its successor.

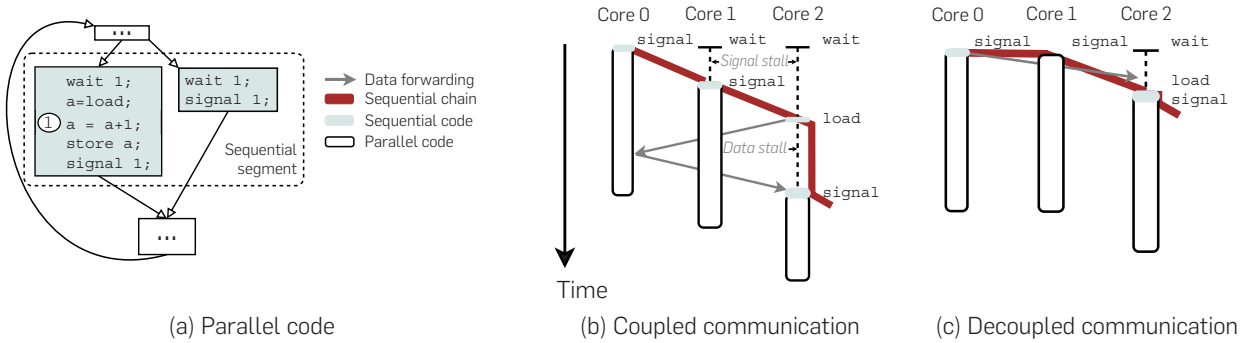
HELIX-RC removes this synchronization overhead by enabling an iteration to detect the readiness of all predecessor iterations, not just one. Therefore, once an iteration forgoes executing the sequential segment, it immediately notifies its successor without waiting for its predecessor. Unfortunately, while HELIX-RC removes unnecessary synchronization, it increases the number of signals that can be in flight simultaneously.

HELIX-RC relies on the `signal` instruction to handle synchronization signals efficiently. Synchronization between a producer core and a consumer includes generation of a signal by the producer, a request for that signal by the consumer, and transmission of the signal between the two. On a conventional multicore processor that relies on a demand-driven memory system for communication, signal transmission is inherently lazy, and signal request and transmission become serialized. With HELIX-RC, on the other hand, a `signal` instructs the ring cache to proactively forward a signal to all other nodes in the ring without interrupting any of the cores, thereby decoupling signal transmission from synchronization.

Code example. The code in Figure 3(a), abstracted for clarity, represents a small hot loop from 175.vpr of SPEC CINT2000. It is responsible for 55% of the total execution time of that program. The loop body has 2 execution paths. The left path entails a loop-carried data dependence because during a typical loop iteration, instruction `1` uses the value of variable `a` produced by a previous iteration. The right path does not depend on prior data. Owing to complex control flow, the compiler cannot predict the execution path taken during a particular iteration, so it must assume that instruction `1` may depend on the previous iteration.

In a conventional implementation coupling communication with computation, the compiler would add `wait 1` and `signal 1` instructions to the right path, as shown in Figure 3(a), to synchronize each iteration with its predecessor and successor iterations. If shared values and signals were communicated on demand, the resulting sequential signal chain would look like that highlighted in red shown in Figure 3(b). If we assume that only iterations 0 and 2, running on cores 0 and 2, respectively, take the left path and execute instruction `1`, then the sequential signal chain shown in Figure 3(b) is unnecessarily long, because iteration 1 only executes parallel code, so the `wait` instruction is unnecessary in that iteration. It results in a signal stall. Iterations 0 and 2, in order to update `a`, must load its

Figure 3. Example illustrating benefits of decoupling communication from computation.



(a) Parallel code (b) Coupled communication (c) Decoupled communication

previous value first, using a regular load. So lazy forwarding of this shared data leads to data stalls, because the transfer only begins when demanded by a load, rather than when generated by a store.

In HELIX-RC, however, a `wait A` unblocks when *all* predecessor iterations have signaled that segment `A` is finished. That allows HCCv3 to omit the `wait 1` on the right path through the loop body. That optimization, combined with HELIX-RC's proactive communication between cores, leads to the more efficient scenario shown in Figure 3(c). The sequential chain in red now only includes the delay required to satisfy the dependence—communication updating a shared value.

4. COMPILER

The decoupled execution model of HELIX-RC described so far is possible given the tight co-design of the compiler and architecture. In this section, we focus on compiler-guaranteed code properties that enable a lightweight ring cache design, and follow up with code optimizations that make use of the ring cache.

4.1. Code properties

- Only 1 loop can run in parallel at a time. Apart from a dedicated core responsible for executing code outside parallel loops, each core is either executing an iteration of the current loop or waiting for the start of the next one.
- Successive loop iterations are distributed to threads in a round-robin manner. Since each thread is pinned to a predefined core, and cores are organized in a unidirectional ring, successive iterations form a logical ring.
- Communication between cores executing a parallelized loop occurs only within sequential segments.
- Different sequential segments always access different shared data. HCCv3 only generates multiple sequential segments when there is no intersection of shared data. Consequently, instances of distinct sequential segments may run in parallel.
- At most 2 signals per sequential segment emitted by a given core can be in flight at any time. Hence, only 2 signals per segment need to be tracked by the ring cache.

This last property allows the elimination of unnecessary `wait` instructions while keeping the architectural enhancement simple. Eliminating `waits` allows a core to execute a later loop iteration than its successor (significantly boosting parallelism). Future iterations, however, produce signals that must be buffered. The last code property prevents a core from getting more than one "lap" ahead of its successor. So when buffering signals, each ring cache node only needs to recognize 2 types—those from the past and those from the future.

4.2. Code optimizations

In addition to conventional optimizations specifically tuned to extract Thread Level Parallelism (TLP) (e.g., code scheduling, method inlining, loop unrolling), HCCv3 includes ones that are essential for best performance of non-numerical programs on a ring-cache-enhanced architecture: aggressive splitting of sequential segments into smaller code blocks; identification and selection of small hot loops; and elimination of unnecessary `wait` instructions.

Sizing sequential segments poses a tradeoff. Additional segments created by splitting run in parallel with others, but extra segments entail extra synchronization, which adds communication overhead. Thanks to decoupling, HCCv3 can split aggressively to efficiently extract TLP. Note that segments cannot be split indefinitely—each shared location must be accessed by only 1 segment.

To identify small hot loops that are most likely to speed up when their iterations run in parallel, HCCv3 profiles the program being compiled using representative inputs. Instrumentation code emulates execution with the ring cache during profiling, which produces an estimate of time saved by parallelization. Finally, HCCv3 uses a loop nesting graph, annotated with the profiling results, to choose the most promising loops.

5. ARCHITECTURE ENHANCEMENTS

Adding a ring cache to a multicore architecture enables the proactive circulation of data and signals that boost parallelization. This section describes the design of the ring cache and its constituent ring nodes. The design is guided by the following objectives:

Low-latency communication. HELIX-RC relies on fast communication between cores in a multicore processor for synchronization and for data sharing between loop

iterations. Since low-latency communication is possible between physically adjacent cores in modern processors, the ring cache implements a simple unidirectional ring network.

Caching shared values. A compiler cannot easily guarantee whether and when shared data generated by a loop iteration will be consumed by other cores running subsequent iterations. Hence, the ring cache must cache shared data. Keeping shared data on local ring nodes provides quick access for the associated cores. As with data, it is also important to buffer signals in each ring node for immediate consumption.

Easy integration. The ring cache is a minimally-invasive extension to existing multicore systems, easy to adopt and integrate. It does not require modifications to the existing memory hierarchy or to cache coherence protocols.

With these objectives in mind, we now describe the internals of the ring cache and its interaction with the rest of the architecture.

5.1. Ring cache architecture

The ring cache architecture relies on properties of compiled code, which imply that the data involved in timing-critical dependences that potentially limit overall performance are both produced and consumed in the same order as loop iterations. Furthermore, a ring network topology captures this data flow, as sketched in Figure 4. The following paragraphs describe the structure and purpose of each ring cache component.

Ring node structure. The internal structure of a per-core ring node is shown in the right half of Figure 4. Parts of this structure resemble a simple network router. Unidirectional links connect a node to its two neighbors to form the ring backbone. Bidirectional connections to the core and private L1 cache allow injection of data into and extraction of data from the ring. There are 3 separate sets of data links and buffers. A primary set forwards data and signals between cores. Two other sets manage infrequent traffic for integration with the rest of the memory hierarchy (see Section 5.2). Separating these 3 traffic types simplifies the design and avoids deadlock. Finally, signals move in lockstep with forwarded data to ensure that a shared memory location is not accessed before the data arrives.

In addition to these router-like elements, a ring node also contains structures more common to caches. A set associative *cache array* stores all data values (and their tags) received by the ring node, whether from a predecessor node or from its associated core. The line size of this cache array is kept at one machine word. While the small line is contrary to typical cache designs, it ensures there will be no false data sharing by independent values from the same line.

The final structural component of the ring node is the *signal buffer*, which stores signals until they are consumed.

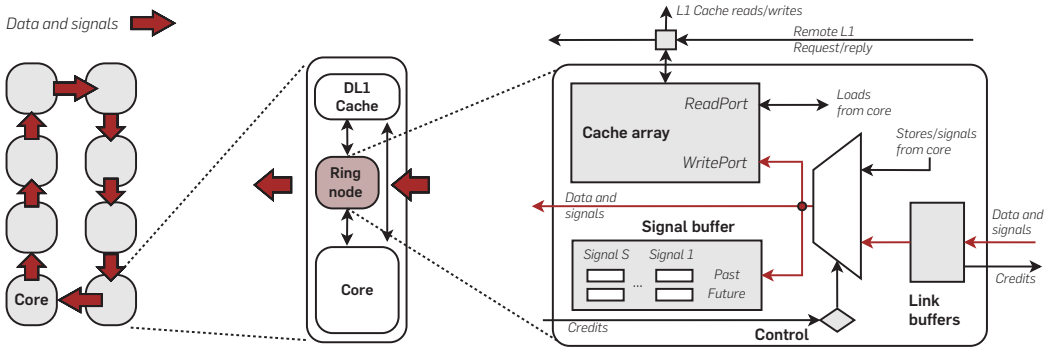
Node-to-node connection. The main purpose of the ring cache is to proactively provide many-to-many core communication in a scalable and low-latency manner. In the unidirectional ring formed by the ring nodes, data propagates by *value circulation*. Once a ring node receives an (address, value) pair, either from its predecessor, or from its associated core, it stores a local copy in its cache array and propagates the same pair to its successor node. The pair eventually propagates through the entire ring (stopping after a full cycle) so that any core can consume the data value from its local ring node, as needed.

This value circulation mechanism allows the ring cache to communicate between cores faster than reactive systems (like most coherent cache hierarchies). In a reactive system, data transfer begins once the receiver requests the shared data, which adds transfer latency to an already latency-critical code path. In contrast, a proactive scheme overlaps transfer latencies with computation to lower the receiver's perceived latency.

The ring cache prioritizes the common case, where data generated within sequential segments must propagate to all other nodes as quickly as possible. Assuming no contention over the network and single-cycle node-to-node latency, the design shown in Figure 4 allows us to bound the latency for a full trip around the ring to N clock cycles, where N is the number of cores. Each ring node prioritizes data received from the ring and stalls injection from its local core.

To eliminate delays to forward data between ring nodes, the number of write ports in each node's cache array must match the link bandwidth between two nodes. While this may seem like an onerous design constraint for the cache array, Section 6 shows that just one write port is sufficient to reap more than 99% of the ideal-case benefits.

Figure 4. Ring cache architecture overview. From left to right: overall system; single core slice; ring node internal structure.



To ensure correctness under network contention, the ring cache is sometimes forced to stall all messages (data and signals) traveling along the ring. The only events that can cause contention and stalls are ring cache misses and evictions, which may then need to fetch data from a remote L1 cache. While these ring stalls are necessary to guarantee correctness, they are infrequent.

The ring cache relies on credit-based flow control[9] and is deadlock free. Each ring node has at least two buffers attached to the incoming links to guarantee forward progress. The network maintains the invariant that there is always at least one empty buffer per set of links somewhere in the ring. That is why a node only injects new data from its associated core into the ring when there is no data from a predecessor node to forward.

Node-core integration. Ring nodes are connected to their respective cores as the closest level in the cache hierarchy (Figure 4). The core's interface to the ring cache is through regular loads and stores for memory accesses in sequential segments.

As previously discussed, `wait` and `signal` instructions delineate code within a sequential segment. A thread that needs to enter a sequential segment first executes a `wait`, which only returns from the associated ring node when matching signals have been received from all other cores executing prior loop iterations. The signal buffer within the ring node enforces this. Specialized core logic detects the start of the sequential segment and routes memory operations to the ring cache. Finally, executing the corresponding `signal` marks the end of the sequential segment.

The `wait` and `signal` instructions require special treatment in out-of-order cores. Since they may have system-wide side effects, these instructions must issue non-speculatively from the core's store queue and regular loads and stores cannot be reordered around them. Our implementation reuses logic from load-store queues for memory disambiguation and holds a lightweight local fence in the load queue until the `wait` returns to the senior store queue. This is not a concern for in-order cores.

5.2. Memory hierarchy integration[a]

The ring cache is a level within the cache hierarchy and as such must not break any consistency guarantees that the hierarchy normally provides. Consistency between the ring cache and the conventional memory hierarchy results from the following invariants: (i) shared memory can only be accessed within sequential segments through the ring cache (compiler-enforced); (ii) only a uniquely assigned *owner* node can read or write a particular shared memory location through the L1 cache on a ring cache miss (ring cache-enforced); and (iii) the cache coherence protocol preserves the order of stores to a memory location through a particular L1 cache.

Sequential consistency. To preserve the semantics of a parallelized single-threaded program, memory operations on shared values require sequential consistency. The ring cache meets this requirement by leveraging the unidirectional data flow guaranteed by the compiler. Sequential consistency must be preserved when ring cache values reach lower-level caches, but the consistency model provided by conventional memory hierarchies is weaker. We resolve this difference by introducing a single serialization point per memory location, namely a unique *owner* node responsible for all interactions with the rest of the memory hierarchy. When a shared value is moved between the ring cache and L1 caches (owing to occasional ring cache load misses and evictions), only its owner node can perform the required L1 cache accesses. This solution preserves existing consistency models with minimal impact on performance.

Cache flush. Finally, to guarantee coherence between parallelized loops and serial code between loop invocations, each ring node flushes the dirty values of memory locations it owns to L1 once a parallel loop has finished execution. This is equivalent to executing a distributed fence at the end of loops. In a multiprogram scenario, signal buffers must also be flushed/restored at program context switches.

6. EVALUATION[b]

By co-designing the compiler along with the architecture, HELIX-RC more than triples the performance of parallelized code when compared to a compiler-only solution. This section investigates HELIX-RC's performance benefits and their sensitivity to ring cache parameters. We confirm that the majority of speedups come from decoupling all types of communication and synchronization. We conclude by analyzing the remaining overheads of the execution model.

6.1. Experimental setup

We ran experiments on 2 sets of architectures. The first relies on a conventional memory hierarchy to share data among the cores. The second relies on the ring cache.

Simulated conventional hardware. We simulate a multicore in-order ×86 processor by adding multiple-core support to the XIOSim simulator. We also simulate out-of-order cores modeled after Intel Nehalem.

The simulated cache hierarchy has 2 levels: a per-core 32 KB, 8-way associative L1 cache and a shared 8 MB 16-bank L2 cache. We vary the core count from 1 to 16, but do not vary the amount of L2 cache with the number of cores, keeping it at 8 MB for all configurations. Also scaling cache size would make it difficult to distinguish the benefits of parallelizing a workload from the benefits of fitting its working set into the larger cache, causing misleading results. Finally, we use DRAMSim2 for cycle-accurate simulation of memory controllers and DRAM.

We extended XIOSim with a cache coherence protocol assuming an optimistic cache-to-cache latency of 10 clock cycles. This 10-cycle latency is optimistically low even compared to research prototypes of low-latency coherence.[11] We only use this low-latency model to simulate conventional hardware, and later (Section 6.2) shows that low latency alone is not enough to compensate for the lazy nature of its

[a] This feature may add one multiplexer delay to the critical delay path from the core to L1.

[b] Most cache coherence protocols (including Intel, AMD, and ARM implementations) provide this minimum guarantee.

coherence protocol.

Simulated ring cache. We extended XIOSim to simulate the ring cache as described in Section 5. We used the following configuration: a 1KB 8-way associative array size, one-word data bandwidth, five-signal bandwidth, single-cycle adjacent core latency, and two cycles of core-to-ring-node injection latency to minimally impact the already delay-critical path from the core to the L1 cache. A sensitiviy analysis of these parameters as well as the evaluation of the ring cache in out-of-order cores can be found in.[4] We use a simple bit mask as the hash function to distribute memory addresses to their owner nodes. To avoid triggering the cache coherence protocol, all words of a cache line have the same owner. Lastly, XIOSim simulates changes made to the core to route memory accesses either to the attached ring node or to the private L1.

Benchmarks. We use 10 out of the 15 C benchmarks from the SPEC CPU2000 suite: 4 floating point (CFP2000) and 6 integer benchmarks (CINT2000). For engineering reasons, the data dependence analysis that HCCv3 relies on[4] requires either too much memory or too much time to handle the rest. This limitation is orthogonal to the results described in this paper.

Compiler. We extended the Intermediate Language Distributed Just-In-Time (ILDJIT) compilation framework,[3] version 1.1, to use LLVM 3.0 for backend machine code generation. We generated both single- and multi-threaded versions of the benchmarks. The single-threaded programs are the unmodified versions of benchmarks, optimized (O3) and generated by LLVM. This code outperforms GCC 4.8.1 by 8% on average and under-performs ICC 14.0.0 by 1.9%. The multi-threaded programs were generated by HCCv3 and the HELIX compiler (i.e., compiler-only solution) to run on ring-cache-enhanced and conventional architectures, respectively. Both compilers produce code automatically and do not require any human intervention. During compilation, they use SPEC training inputs to select the loops to parallelize.

Measuring performance. We compute speedups relative to sequential simulation. Both single- and multi-threaded runs use reference inputs. To make simulation feasible, we simulate multiple phases of 100 M instructions as identified by SimPoint.

6.2. Speedup analysis[c]

In our 16-core processor evaluation system, HELIX-RC boosts the performance of sequentially-designed programs (CINT2000), assumed not to be amenable to parallelization. Figure 5 shows that HELIX-RC raises the geometric mean of speedups for these benchmarks from 2.2× for a compiler-only solution to 6.85×.

HELIX-RC not only maintains the performance of a compiler-only solution on numerical programs (SPEC CFP2000), but also increases the geometric mean of speedups for CFP2000 benchmarks from 11.4× to almost 12×.

We now turn our attention to understanding where the speedups come from.

Communication. Speedups obtained by HELIX-RC come from decoupling both synchronization and data communication from computation in loop iterations, which significantly reduces communication overhead, allows the compiler to split sequential segments into smaller blocks, and cuts down the critical path of the generated parallel code. Figure 6 compares the speedups gained by multiple combinations of decoupling synchronization, register-, and memory-based communication. As expected, fast register transfers alone do not provide much speedup since most in-register dependences can be satisfied by re-computing the shared variables involved.[4] Instead, most of the speedups come from decoupling communication for both synchronization and memory-carried actual dependences. To the best of our knowledge, HELIX-RC is the only solution that accelerates all 3 types of transfers for actual dependences.

Figure 5. HELIX-RC triples the speedup obtained by a compiler-only solution for SPEC INT benchmarks. Speedups are relative to sequential program execution.

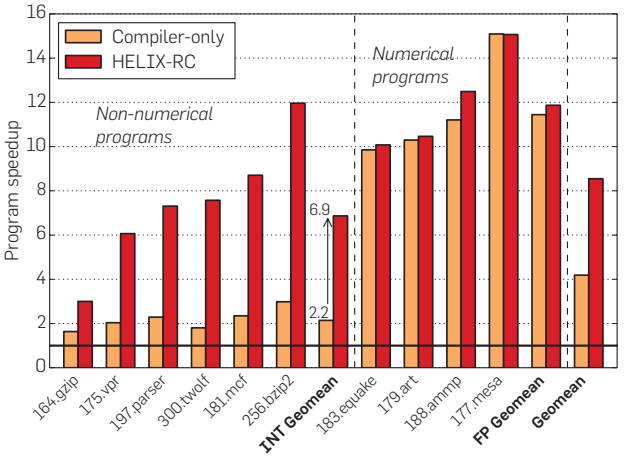
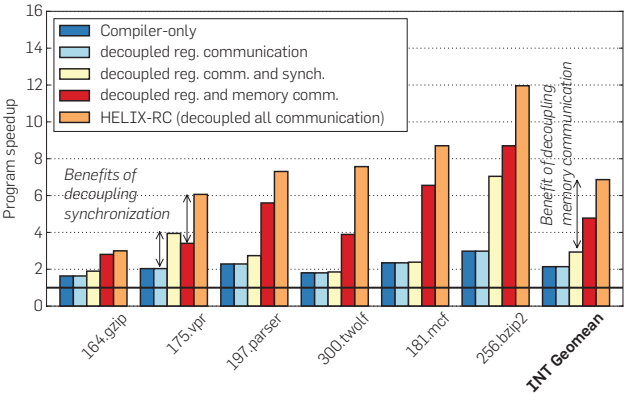


Figure 6. Breakdown of benefits of decoupling communication from computation.



[c] As an aside, automatic parallelization features of ICC led to a geomean slowdown of 2.6% across SPEC CINT2000 benchmarks, suggesting ICC cannot parallelize non-numerical programs.
These speedups are possible even with a cache coherence latency of conventional processors (e.g., 75 cycles).

Sequential segments. While more splitting offers higher TLP (more sequential segments can run in parallel), it also requires more synchronization at run time. Hence, the high synchronization cost for conventional multicores discourages aggressive splitting of sequential segments. In contrast, the ring cache enables aggressive splitting to maximize TLP.

To analyze the relationship between splitting and TLP, we computed the number of instructions that execute concurrently for the following 2 scenarios: (i) conservative splitting constrained by a contemporary multicore processor with high synchronization penalty (100 cycles) and (ii) aggressive splitting for HELIX-RC with low-latency communication (<10 cycles) provided by the ring cache. In order to compute TLP independent of both the communication overhead and core pipeline advantages, we used a simple abstracted model of a multicore system that has no communication cost and is able to execute 1 instruction at a time. Using the same set of loops chosen by HELIX-RC and used in Figure 5, TLP increased from 6.4 to 14.2 instructions with aggressive splitting. Moreover, the average number of instructions per sequential segment dropped from 8.5 to 3.2 instructions.

Coverage. Despite all the loop-level speedups possible via decoupling communication and aggressively splitting of sequential segments, Amdahl's law states that program coverage dictates the overall speedup of a program. Prior parallelization techniques have avoided selecting loops with small bodies because communication would slow down execution on conventional processors.[5, 20] Since HELIX-RC does not suffer from this problem, the compiler can freely select small hot loops to cover almost the entirety of the original program.

6.3. Analysis of overhead

To understand areas for improvement, we categorize every overhead cycle preventing ideal speedup. Figure 7 shows the results of this categorization for HELIX-RC, again implemented on a 16-core processor.

Most importantly, the small fraction of *communication* overheads suggests that HELIX-RC successfully eliminates the core-to-core latency for data transfer in most benchmarks. For several benchmarks, notably 175.vpr, 300.twolf, 256.bzip2, and 179.art, the major source of overhead is the low number of iterations per parallelized loop (*low trip count)*. While many hot loops are frequently invoked, low iteration count (ranging from 8 to 20) leads to idle cores. Other benchmarks such as 164.gzip, 197.parser, 181.mcf, and 188.ammp suffer from dependence waiting due to large sequential segments. Finally, HCCv3 must sometimes add a large number of `wait` and `signal` instructions (i.e., many sequential segments) to increase TLP, as seen for 164.gzip, 197.parser, 181.mcf, and 256.bzip2.

7. RELATED WORK

To compare HELIX-RC to a broad set of related work, Table 1 summarizes different parallelization schemes proposed for non-numerical programs organized with respect to the types of communication decoupling implemented (register vs. memory) and the types of dependences targeted (actual vs. false). HELIX-RC covers the entire design space and is the only one to decouple memory accesses from computation for actual dependences.

7.1. Multiscalar register file

Multiscalar processors[19] extract both Instruction Level Parallelism (ILP) and TLP from an ordinary application. While a ring cache's structure resembles a Multiscalar register file, there are fundamental differences. For the Multiscalar register file, there is a fixed and relatively small number of shared elements that must be known at compile time. Furthermore, the Multiscalar register file cannot handle memory updates by simply mapping memory to a fixed number registers without a replacement mechanism. In contrast, the ring cache does not require compile-time knowledge to handle an arbitrary number of elements shared between cores (i.e., memory locations allocated at runtime) and can readily handle register updates by deallocating a register to a memory location. In other words, HELIX-RC proposes to use a distributed cache to handle both register and memory updates.

7.2. Cache coherence protocols

The ring cache addresses an entirely different set of communication demands. Cache coherence protocols target relatively small amounts of data shared infrequently between cores. Hence, cores can communicate lazily, but the resulting communication almost always lies in the critical sequential chain. In contrast, the ring cache targets frequent and time-critical data sharing between cores.

7.3. On-chip networks

Figure 7. Breakdown of overheads that prevent achieving ideal speedup.

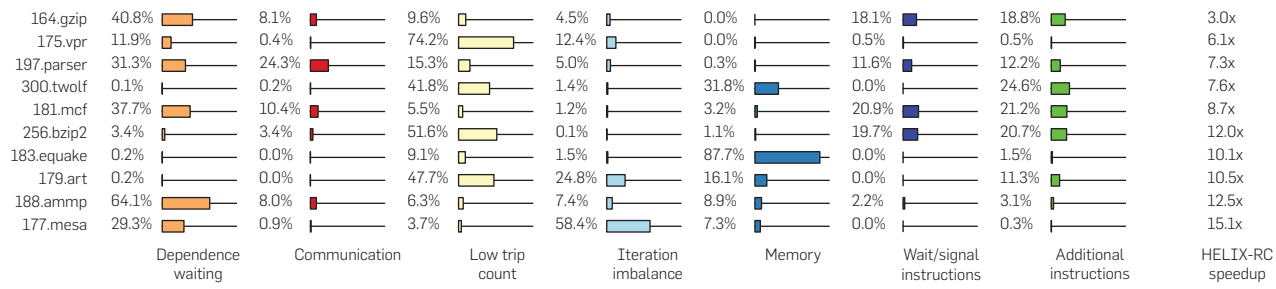


Table 1. Only HELIX-RC decouples communication for all types of dependences.

	Actual dependences	False dependences
Register	**HELIX-RC**, Multiscalar, TRIPS, T3	**HELIX-RC**, Multiscalar, TRIPS, T3
Memory	**HELIX-RC**	**HELIX-RC**, TLS-based approaches, Multiscalar, TRIPS, T3

While On-Chip-Networks (OCNs) can take several forms, they commonly implement reactive coherence protocols[18, 21, 24, 25] that do not fulfill the low-latency communication requirements of HELIX-RC. Scalar operand networks[22] somewhat resemble a ring cache to enable tight coupling between known producers and consumers of specific operands, but they suffer from the same limitations as the Multiscalar register file. Hence, HELIX-RC implements a relatively simple OCN, but supported by compiler guarantees and additional logic to implement automatic forwarding.

7.4. Off-chip networks

Networks that improve bandwidth between processors have been studied extensively.[17] While they work well for Cyclic Multithreading (CMT) parallelization techniques that require less frequent data sharing, there is less overall parallelism. Moreover, networks that target chip-to-chip communication do not meet the very different low-latency core-to-core communication demands of HELIX-RC.[9] Our results show HELIX-RC is much more sensitive to latency than to bandwidth.

7.5. Non-commodity processors

Multiscalar,[19] TRIPS,[16] and T3 (Ref. Robatmil *et al*.[15]) are polymorphous architectures that target parallelism at different granularities. They differ from HELIX-RC in that (i) they require a significantly larger design effort and (ii) they only decouple register-to-register communication and/or false memory dependence communication by speculating.

An iWarp system[2] implements special-purpose arrays that execute fine- and coarse-grained parallel numerical programs. However, without an efficient broadcast mechanism, iWarp's fast communication cannot reach the speedups offered by HELIX-RC.

7.6. Automatic parallelization of non-numerical programs

Several automatic methods to extract TLP have demonstrated respectable speedups on commodity multicore processors for non-numerical programs.[5, 8, 13, 23] All of these methods transform loops into parallel threads. Decoupled Software Pipelining (DSWP)[13] reduces sensitivity to communication latency by restructuring a loop to create a pipeline among the extracted threads with unidirectional communication between pipeline stages. Demonstrated both on simulators and on real systems, DSWP performance is largely insensitive to latency. However, significant restructuring of the loop makes speedups difficult to predict and generated code can sometimes be slower than the original. Moreover, DSWP faces the challenges of selecting appropriate loops to parallelize and keeping the pipeline balanced at runtime. While DSWP-based approaches focus more on restructuring loops to hide communication latency,[8, 13] HELIX-RC proposes an architecture-compiler co-design strategy that selects the most appropriate loops for parallelization.

Combining DSWP with HELIX-RC has the potential to yield significantly better performance than either alone. DSWP cannot easily scale beyond 4 cores[14] without being combined with approaches that exploit parallelism among loop iterations (e.g., DOALL).[8] While DSWP + DOALL can scale beyond several cores, DOALL parallelism is not easy to find in non-numerical code. Instead, DSWP + HELIX-RC presents an opportunity to parallelize a much broader set of loops.

Several TLS-based techniques,[7, 10, 20] including Stanford Hydra, POSH, and STAMPede, combine hardware-assisted Thread Level Speculation (TLS) with compiler optimizations to manage dependences between loop iterations executing in different threads. When the compiler identifies sources and destinations of frequent dependences, it synchronizes using `wait` and `signal` primitives; otherwise, it uses speculation. HELIX-RC, on the other hand, optimizes code assuming all dependences are actual. While we believe adding speculation may help HELIX-RC, Figure 5 shows decoupled communication already yields significant speedups without misspeculation overheads.

8. CONCLUSION

HELIX-RC shows how to *accelerate non-numerical programs* by exploiting parallelism between the iterations of their small loops. Successfully mapping the iterations of such loops onto multiple cores of a single chip requires a *low-latency*, broadcast interconnect between cores. This interconnect needs to be *proactive* (so that communication starts as soon as data is generated), and it must be able to update memory locations stored in each core's private cache.

8.1. Accelerating non-numerical programs to catch up with hardware evolution

Adding multiple cores to a single chip has been proposed, studied, and realized in products since the 90's (Ref. Olukotun *et al*.[12]), but the majority of these cores are still under-utilized even after more than 15 years' effort in both compiler and programming language research. Having reached the "ILP wall," industry now relies on these multiple cores to gain performance from each system. However, successful uses of multiple cores exist only when the goal is maximizing throughput combined with massive data parallelism or parallelism among multiple programs, as is available in Graphics Processing Unit (GPU) computing or within data centers. On the other hand, if single program performance is the target and there is little or no data parallelism available (e.g., non-numerical programs running on mobile phones or client computers), then only a few cores are actually used, leaving the majority of them under-utilized.[1] Our work shows how to actually take advantage of the cores that are available within a single chip when running non-numerical programs, highlighting the great potential of including

hardware support for a proactive, cache-based, low-latency core-to-core interconnect.

8.2. Transforming parallelism into performance requires low-latency communication

Our work demonstrates the fundamental value of having a low-latency interconnect to boost the performance of complex, non-numerical programs. The dependence between communication latency and performance of a program has already been observed in high-performance computing domains.[17] Moreover, prior work on on-chip networks has shown the value of a low-latency interconnect both for programs with regular control and data flows[22, 25] and for a novel research architecture.[6, 21] Our work is the first to demonstrate the value of a cache-based, low latency interconnect between cores of commodity processors for accelerating complex, non-numerical programs running on a chip.

8.3. From reactive hardware-driven to proactive software-driven cache communication

HELIX-RC has the potential to influence the adoption of proactive, cache-based, and one-to-many interconnects in commodity processors. To quantify the need for such solutions, we measured the communication latency between adjacent cores in several generations of Intel commodity processors. As highlighted in Figure 1a, conventional reactive solutions have latencies of around 100 cycles. The figure shows that, among the five generations of Intel processors we considered, adjacent core latency bounces around 100 cycles without a monotonic trend over time. This suggests that there is no reason to expect conventional solutions (reactive hardware-driven) to improve in the future.

HELIX-RC motivates shifting inter-core communication mechanisms towards alternative cache-based solutions, in which a compiler identifies for the hardware the code that will generate shared data. The architecture, for its part, will proactively communicate modified values to make them locally accessible by other cores. This allows a drastic cut in the latency of remote data access, which, therefore, allows a parallelizing compiler to take advantage of the substantial latent parallelism between the iterations of small loops. C

References

1. Blake, G., Dreslinski, R.G., Mudge, T., Flautner, K. Evolution of thread-level parallelism in desktop applications. In *ISCA* (2010).
2. Borkar, S., Cohn, R., Cox, G., Gleason, S., Gross, T., Kung, H.T., Lam, M., Moore, B., Peterson, C., Pieper, J., Rankin, L., Tseng, P.S., Sutton, J., Urbanski, J., Webb, J. iWarp: An integrated solution to high-speed parallel computing. *Supercomputing* (1988).
3. Campanoni, S., Agosta, G., Reghizzi, S.C., Biagio, A.D. A highly flexible, parallel virtual machine: Design and experience of ILDJIT. *Software: Practice and Experience* (2010).
4. Campanoni, S., Brownell, K., Kanev, S., Jones, T.M., Wei, G.-Y., Brooks, D. HELIX-RC: An architecture-compiler co-design for automatic parallelization of irregular programs. In *ISCA* (2014).
5. Campanoni, S., Jones, T., Holloway, G., Reddi, V.J., Wei, G.-Y., Brooks, D. HELIX: Automatic Parallelization of Irregular Programs for Chip Multiprocessing. In *CGO* (2012).
6. Ceze, L., Tuck, J., Torrellas, J., Cascaval, C. Bulk disambiguation of speculative threads in multiprocessors. In *ISCA* (2006).
7. Hammond, L., Hubbert, B.A., Siu, M., Prabhu, M.K., Chen, M.K., Olukotun, K. The Stanford Hydra CMP. *IEEE Micro* (2000).
8. Huang, J., Raman, A., Jablin, T.B., Zhang, Y., Hung, T.-H., August, D.I. Decoupled software pipelining creates parallelization opportunities. In *CGO* (2010).
9. Jerger N.E., Peh, L.-S. *On-Chip Networks*. Synthesis Lectures on Computer Architecture. Morgan & Claypool, 2009.
10. Liu, W., Tuck, J., Ceze, L., Ahn, W., Strauss, K., Renau, J., Torrellas, J. POSH: A TLS compiler that exploits program structure. In *PPoPP* (2006).
11. Martin, M.M.K. *Token Coherence*. PhD thesis, University of Wisconsin-Madison, 2003.
12. Olukotun, K., Nayfeh, B.A., Hammond, L., Wilson, K., Chang, K. The case for a single-chip multiprocessor. *ASPLOS* (1996).
13. Ottoni, G., Rangan, R., Stoler, A., August, D.I. Automatic thread extraction with decoupled software pipelining. In *MICRO* (2005).
14. Rangan, R., Vachharajani, N., Ottoni, G., August, D.I. Performance scalability of decoupled software pipelining. In *ACM TACO* (2008).
15. Robatmil, B., Li, D., Esmaeilzadeh, H., Govindan, S., Smith, A., Putnam, A., Burger, D., Keckler, S.W. How to Implement Effective Prediction and Forwarding for Fusable Dynamic Multicore Architectures. In *HPCA* (2013).
16. Sankaralingam, K., Nagarajan, R., Liu, H., Kim, C., Huh, J., Ranganathan, N., Burger, D., Keckler, S.W., McDonald, R.G., Moore, C.R. TRIPS: A polymorphous architecture for exploiting ILP, TLP, and DLP. In *ACM TACO* (2004).
17. Scott, S.L. Synchronization and Communication in the T3E Multiprocessor. In *ASPLOS* (1996).
18. Seiler, L., Carmean, D., Sprangle, E., Forsyth, T., Abrash, M., Dubey, P., Junkins, S., Lake, A., Sugerman, J., Cavin, R., Espasa, R., Grochowski, E., Juan, T., Hanrahan, P. Larrabee: a many-core x86 architecture for visual computing. *ACM Transactions on Graphics* (2008).
19. Sohi, G.S., Breach, S.E., Vijaykumar, T.N. Multiscalar processors. In *ISCA* (1995).
20. Steffan, J.G., Colohan, C., Zhai, A., Mowry, T.C. The STAMPede approach to thread-level speculation. *ACM Transactions on Computer Systems* (2005).
21. Taylor, M.B., Kim, J., Miller, J., Wentzlaff, D., Ghodrat, F., Greenwald, B., Hoffman, H., Johnson, P., Lee, J.-W., Lee, W., Ma, A., Saraf, A., Seneski, M., Shnidman, N., Strumpen, V., Frank, M., Amarasinghe, S., Agarwal, A. The RAW microprocessor: A computational fabric for software circuits and general-purpose programs. *IEEE Micro* (2002).
22. Taylor, M.B., Lee, W., Amarasinghe, S.P., Agarwal, A. Scalar operand networks. *IEEE Transactions on Parallel Distributed Systems* (2005).
23. Tournavitis, G., Wang, Z., Franke, B., O'Boyle, M.F.P. Towards a holistic approach to auto-parallelization. In *PLDI* (2009).
24. van der Wijngaart, R.F., Mattson, T.G., Haas, W. Light-weight communications on Intel's single-chip cloud computer processor. *SIGOPS Operating Systems Review* (2011).
25. Wentzlaff, D., Griffin, P., Hoffmann, H., Bao, L., Edwards, B., Ramey, C., Mattina, M., Miao, C.-C., Brown, J.F. III, Agarwal, A. On-chip interconnection architecture of the tile processor. *IEEE Micro* (2007).

Simone Campanoni, Northwestern University, Evanston, IL.
Kevin Brownell, Svilen Kanev, Gu-Yeon Wei, and David Brooks, Harvard University, Cambridge, MA.
Timothy M. Jones, University of Cambridge, England.

Copyright held by authors/owners. Publishing rights licensed to ACM. $15.00