Unified Cache: A Case for Low-Latency Communication

Khalid Al-Hawaj Simone Campanoni Gu-Yeon Wei David Brooks

{hawajkm, xan, guyeon, dbrooks}@eecs.harvard.edu

Harvard University

1. INTRODUCTION

Increasing computational demand on mobile devices calls for energy-friendly solutions for accelerating single programs. In the multicore era, thread level parallelism (TLP) can accelerate single-threaded programs without requiring power-hungry cores. HELIX-RC, a recently proposed co-design between the HELIX parallelizing compiler and its target architecture, shows that substantial TLP can be extracted from loops with small bodies by optimizing core-to-core communication. Previously, the effectiveness of the HELIX-RC approach has been demonstrated through simulation. In this paper, we evaluate a HELIX-RC-like solution on a real platform.

We have developed a simplified version of the HELIX-RC architecture that we call *unified cache*, and we have implemented it on an FPGA board. Our design augments a multicore platform with a simplified ring cache—the architectural component of the HELIX-RC co-design. With the aid of microbenchmarks, our FPGA prototype confirms the HELIX-RC findings.

After describing both the ring cache and the parallel code generated by the HELIX compiler, we sketch the design of the unified cache and we evaluate its implementation on a Xilinx VC707 FPGA board.

2. BACKGROUND

The HELIX compiler automatically parallelizes sequential programs by distributing their loop iterations around a ring of cores. On conventional commodity platforms, dependences between loop iterations are satisfied through the memory hierarchy. The resulting high latency severely limits performance on such systems. In HELIX-RC, we proposed architectural support called *ring cache* to lower communication latency and enable more parallelization.

2.1 Helix Execution Model

HELIX [3, 4] accelerates programs by finding the most promising loops to parallelize. Successive iterations of each parallelized loop run on adjacent cores. For each data dependence between loop iterations, the compiler produces a *sequential segment* of code. The instances of a sequential segment running on separate cores must be executed in the same order as in the original sequential loop. So the compiler brackets each sequential segment with synchronization operations: *signal* ends the segment by signaling other cores that is has finished, and *wait* begins a segment by waiting for such a signal.

2.2 Conventional Inter-Core Communication

In commodity processors, cores communicate through memory. A core that needs data from another one uses the *cache coherence protocol* [5], which locates the data and moves it to the requesting core. Because coherence protocols are reactive, each request incurs a heavy overhead, on the order of a hundred cycles.

2.3 Ring Cache

To reduce communication latency between the cores of a conventional multicore processor, HELIX-RC [2] introduces *ring cache*: a light-weight architectural enhancement. HELIX-RC adds a *ring node* to each core and connects these nodes in a ring. When a ring node receives program data or synchronization signals, it passes them immediately to its successor node. This proactive communication around the ring of cores lowers latency and thereby enables loop parallelization that would otherwise be infeasible.

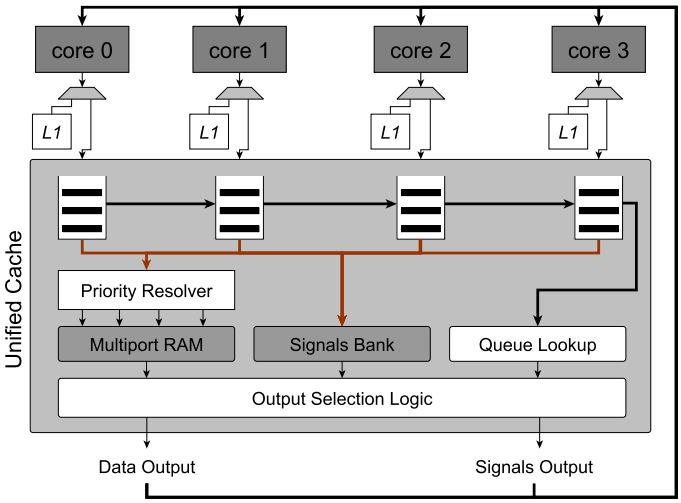

Figure 1: General overview of the unified cache design.

3. UNIFIED CACHE

To quantify the advantages of adding ring cache to a real platform, we implemented a simplification of it called *unified cache*.

3.1 Overview

Unified Cache is a customized shared cache in a chip multiprocessor. It has an input port from each core. During each sequential segment of a parallelized loop, a core uses this shared cache instead of its private first-level data cache (DL1). Code at the start of a parallelized loop primes the unified cache for the loop's sequential segments. Termination of the loop triggers a *flush* operation that writes unified cache data back to DL1.

3.2 Implementation

The unified cache handles core-to-core communication both for sharing data and synchronizing parallel execution.

Data. Data coming into the unified cache from cores enters queues that buffer and order the data by its core of origin. The queues are needed because the unified cache has a multiport RAM block that can perform multiple data reads, but only one write at a time. As the queues hold in-flight data, they are searched for every read request, which minimizes the delay between a *store* and a subsequent *load*.

Synchronization. Signals are connected to a bank called the *signals bank*, which performs n reads and n writes in a single cycle, where n is the number of input ports. This enables the

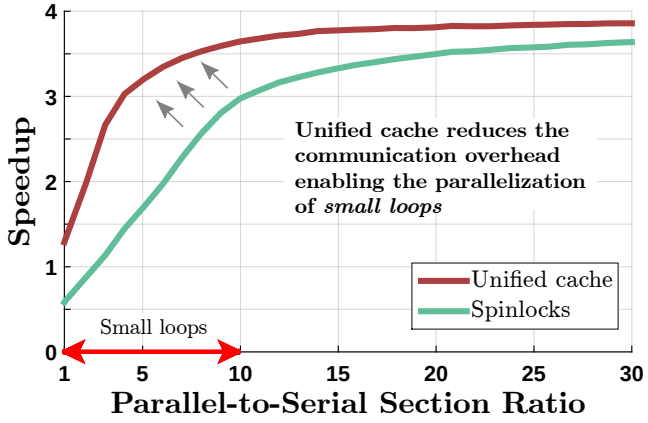

Figure 2: `LinkedList` *with different loop body sizes.*

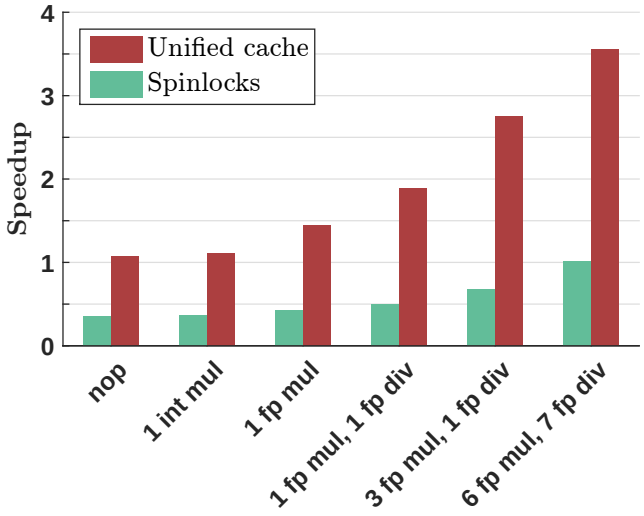

Figure 3: `Scan` *with different loop bodies*

unified cache to handle intensive synchronization requests from cores without stalls.

4. EXPERIMENT AND RESULTS

We implemented the unified cache on a LEON3 platform. Our results reinforce those of the HELIX-RC simulations: Lowering core-to-core communication latency to fewer than 10 clock cycles enables parallelization of small loops that would otherwise be infeasible.

4.1 Platform

We synthesized our LEON3 [1] platform with the parameters shown in Table 1. Each core in the LEON3 CMP has its own DL1 cache. Because there is no last-level cache, cache coherence is maintained through the DDR3 RAM. When unified cache is not in use, cores are interconnected through a shared AMBA bus, where they snoop memory requests to maintain coherence.

4.2 Benchmarks

To quantify the effects of adding unified cache to the system, we created two benchmarks: `LinkedList` and `SCAN`. Each is implemented in three ways: as a sequential program and as parallelized in two ways. The first parallelization relies on traditional spin locks, based on *load* and *store* instructions. The second parallelization relies on unified cache. Performance of each parallelized version is normalized to the respective sequential version.

LinkedList. This benchmark traverses a linked list of elements. After accessing an element e in the list, the benchmark calls a function $f(e)$ that takes n cycles, where n is passed as a parameter to the benchmark. This benchmark has one sequential segment, which implements retrieval of the current list link from a previous iteration. Its parallel segment is the computation of $f(e)$. We sweep the ratio between the number of cycles required by the parallel segment and the number required by the sequential segment.

Parameter	Value
Platform	LEON3
ISA	SPARC-v8e
Number of Cores	4
Cache Configuration	16 kByte, 4 ways
TLB Entries	16 Entries
FPGA Platform	Xilinx VC707
Main Memory	1GByte DDR3-RAM
Operating System	GNU Linux 2.6.8
Number of Unified Cache Ports	4
Size of Unified Cache Queues	1 Entry
Unified Cache Signal Banks	64 Signals
Unified Cache Data Cache	4 kByte, 1 way

Table 1: LEON3 platform configuration.

Scan. The `Scan` benchmark [6] also traverses a linked list of elements e, accumulating $\sum_e f(e)$ and replacing each element in the list with the sum that its contribution produces. As a result, it has two *sequential segments*: retrieval of the current list link from a previous iteration, and reading and updating the accumulator.

4.3 Results

Characterization. Adding the unified cache to the platform requires the modification of two crucial paths in the processor: *memory operations* and *exception handling*. Therefore, adding the unified cache might degrade platform frequency and add area and routing overhead. After synthesizing the system, we found that the addition of the unified cache *did not* incur a heavy penalty. Moreover, unified cache lowers the communication latency between cores from 58 cycles to 3 cycles. (Three cycles is the ISA defined minimum for one load and one store [7].) Finally, as incoming data to the unified cache is being queued, we expect *store* instructions to take longer than the two cycles that the ISA specifies [7]. Our measurements show that even in the worst case, when the queues for every port are fully utilized, *store* latency increases to only 4 cycles.

Unified cache increases parallelism. To show the importance of low-latency communication between cores, we ran the `LinkedList` benchmark on our platform sweeping parallel-to-serial ratio from 1× to 30× (Figure 2). As shown in this figure, unified cache soon outstrips the traditional spin lock solution. Therefore, unified cache allows parallelization of small loops that tend to have small parallel-to-serial ratios.

Testing `Scan` leads to the same conclusion as for `LinkedList`. We compiled `Scan` with 6 different functions $f(e)$. Figure 3 shows that, compared to traditional spin locks, the unified cache helps parallelized `Scan` achieve higher performance with minimal $f(e)$. `Scan`'s spin lock version struggles to improve performance even when compared to the spin-lock version of `LinkedList`. Hence, even for `Scan`, proactive inter-core communication is essential for accelerating the performance of small loops through parallelization.

5. REFERENCES

[1] Aeroflex Gaisler, "GRLIB IP Library User's Manual – version grlib-gpl-1.3.7-b4144," http://www.gaisler.com/products/grlib/grlib.pdf, 2014.

[2] S. Campanoni, K. Brownell, S. Kanev, T. M. Jones, G.-Y. Wei, and D. Brooks, "HELIX-RC: An Architecture-Compiler Co-Design for Automatic Parallelization of Irregular Programs," in *ISCA*, 2014.

[3] S. Campanoni, G. Holloway, G.-Y. Wei, and D. Brooks, "Helix: Automatic parallelization of irregular programs for chip multiprocessing," in *CGO*, 2012.

[4] S. Campanoni, T. M. Jones, G. H. Holloway, G.-Y. Wei, and D. M. Brooks, "HELIX: Making the extraction of thread-level parallelism mainstream," *IEEE Micro*, 2012.

[5] L. Choi and P.-C. Yew, "Compiler and hardware support for cache coherence in large-scale multiprocessors: Design considerations and performance study," *SIGARCH Comput. Archit. News*, 1996.

[6] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter, ser. GPGPU workshop, 2010.

[7] SPARC International Inc., "The SPARC Architecture Manual Version 8," http://gaisler.com/doc/sparcv8.pdf.