Fine-Grain OpenMP Runtime Support with Explicit Communication Hardware Primitives

Pranav Tendulkar, Vassilis Papaefstathiou, George Nikiforos,
Stamatis Kavadias, Dimitrios S. Nikolopoulos and Manolis Katevenis
Institute of Computer Science, FORTH, Heraklion, Crete, Greece – member of HiPEAC
Email: {pranav,papaef,nikiforg,kavadias,dsn,kateveni}@ics.forth.gr

***Abstract*— We present a runtime system that uses the explicit on-chip communication mechanisms of the SARC multi-core architecture, to implement efficiently the OpenMP programming model and enable the exploitation of fine-grain parallelism in OpenMP programs. We explore the design space of implementation of OpenMP directives and runtime intrinsics, using a family of hardware primitives; remote stores, remote DMAs, hardware counters and hardware event queues with automatic responses, to support static and dynamic scheduling and data transfers in local memories. Using an FPGA prototype with four cores, we achieve OpenMP task creation latencies of 30-35 processor clock cycles, initiation of parallel contexts in 50 cycles and synchronization primitives in 65-210 cycles.**

I. Introduction

The increased demand for concurrency from many-core processors makes strong scaling of parallel applications a fundamental requirement of parallel programming models. Strong scaling in turn implies the extraction of fine-grain tasks and simple instruction streams [1]. Effective fine-grain parallelism exploitation remains a challenge for hardware and software due frequent communication and synchronization between cores. On multi-core processors with coherent caches, communication is consumer-initiated, thus requiring round-trip messages to transfer data between cores, while synchronization is implemented with atomic operations, which trigger sequences of invalidations and subsequent data transfers from remote caches or memory. Hardware support for explicit communication, such as hardware LIFO queues (Carbon [1]), asynchronous direct messages [2], RDMAs [3], and hardware queues with asynchronous event-responses [4] provide viable solutions to these problems.

This paper explores the design and implementation of OpenMP on a multi-core system that offers explicit communication primitives for fast on-chip data transfers. Our thesis is that explicit on-chip communication mechanisms support efficiently a wide range of primitives that are common in runtime systems for parallel programming. The OpenMP primitives include: (i) scheduling of parallel loops and asynchronous tasks [5], using either work-stealing [6] or work-sharing [7]; (ii) user-level synchronization including locks, barriers, and reductions and (iii) data privatization in local memories [8].

We present the design and implementation of an OpenMP runtime system for an FPGA prototype of the SARC architecture [9] which features explicitly managed on-chip local memories, explicit on-chip communication primitives including remote stores for producer-initiated short data transfers, RDMA operations for producer-initiated or consumer-initiated bulk data transfers, hardware event queues with automatically generated responses, and hardware counters [4].

Our OpenMP implementation on a four-core SARC multi-core FPGA prototype, achieves parallel task initiation in 30-35 processor clock cycles. Synchronization operations including barriers, locks, atomic regions, and reductions on integer variables cost 65-210 clock cycles. Our implementation achieves near-peak on-chip data transfer bandwidth in memory-bound computational kernels and good scaling with fine-grain tasks.

978-3-9810801-7-9/DATE11/©2011 EDAA

II. Hardware Platform

SARC features a configurable L2 cache memory which can be dynamically partitioned at run-time between a coherent cache and a software-managed local memory (scratchpad), which is visible in global address space. The FPGA prototype of SARC used in this work has four Microblaze cores, with a private L1 cache and the configurable L2 for each core. The current prototype does not support cache coherence.

SARC also defines primitive mechanisms for virtualized inter-core communication such as direct remote loads and stores to the scratchpad of other cores, remote DMA reads and writes for efficient bulk transfers between scratchpads, and between off-chip DRAM and scratchpads. The architecture offers three *event-response-type* hardware mechanisms to support communication and synchronization between cores: *(i) Command-Buffers*: used to initiate multi-word communication (messages and DMA), *(ii) Counters*: used to implement barrier synchronization and completion notification for unordered sequences of operations, and *(iii) Hardware Queues*: a generic mechanism to implement atomic multi-party synchronization.

Counters implement atomic add-on-store and allow up-to four configurable notification addresses to receive a pre-configured word when the counter reaches zero; these addresses may reside in remote nodes.

Queues offer atomic enqueue and dequeue operations that can be used for inter-core synchronization and for efficient task dispatching and scheduling. The SARC prototype offers two types of hardware queues: *(i) Single Reader Queues (SRQ)* and *(ii) Multiple Reader Queues (MRQ)* both hosted inside scratchpad memory and following FIFO order. Any core can atomically enqueue elements in both SRQs and MRQs with

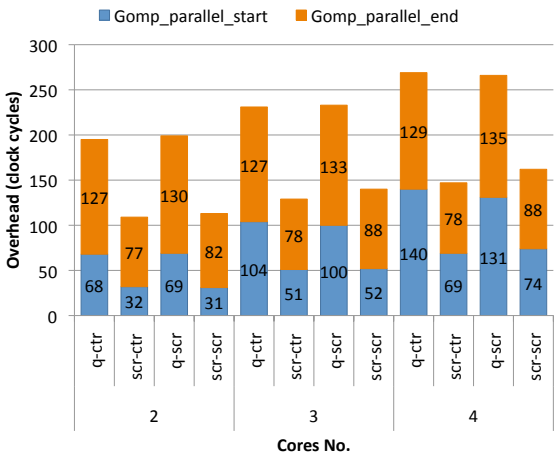

Fig. 1: **`omp parallel`** overhead

write-messages of size up to a cache-line. SRQs allow dequeue operations only from the core that hosts the queue locally, while any core may dequeue from MRQs; dequeue operations are read-messages. The unique feature of MRQs is that they match enqueue with dequeue operations, i.e. a dequeue from a empty queue does not fail but instead waits for the next enqueue in order to be matched; upon matching a new message containing the enqueued element is sent to the reader.

SARC FPGA prototype description and latency analysis of all these hardware primitives is available in [4], [9].

III. OpenMP directives implementation

The OpenMP v3.0 software stack on SARC comprises application code compiled using OpenMP C compiler Rose [10], a re-targeted runtime GOMP API [11] library and a low-level communication library mapping all hardware features like DMA, scratchpad management etc. In the following subsections we discuss the implementation of OpenMP directives on the SARC prototype, and measure the overhead of each directive using the EPCC benchmarks [12], Table I.

A. Initiating parallel execution

The **`omp parallel`** directive specifies a parallel region which will be cloned and executed in all cores. Spawning a parallel region requires transferring a *region descriptor* (function pointer and arguments) from the master core to slave cores. The runtime system can pass the *region descriptor*, either in a hardware SRQ (labeled q-, Table I) or in pre-allocated scratchpad area (labeled scr-), where slave cores wait for new descriptors. For region completion notification from the slaves we have two implementation options: (i) notify the master in a hardware up-counter incremented by slave cores (labeled ctr-) and (ii) use a pre-allocated scratchpad area (labeled scr-) per slave core for posing completion flag.

Figure 1 presents the overhead breakdown for the parallel directive. **`GOMP_parallel_start`** is the time required to start parallel regions on all slaves, whereas **`GOMP_parallel_end`** is the time spent by the master to wait for region completions. Posting to scratchpad space is implemented with fast remote stores – plain store instructions – for short descriptors (up-to 32 bytes) and with remote DMA writes for longer descriptors. In contrast, posting a descriptor to a remote hardware queue takes 20 cycles and it has to be copied from the hardware queue to software buffers at the receiver. Using a counter for region completion notification is faster than using remote stores to scratchpad addresses, since in the counter-based implementation the master waits on one scratchpad location, instead of many locations, one per slave, when distinct scratchpad notification addresses are used.

Initiating OpenMP parallel regions and processing their completion in the SARC prototype cost between 109 and 148 cycles per region, depending on NoC contention.

B. Statically scheduled for loops

The **`omp for`** directive parallelizes for loops. It incurs overhead for statically computing the range of loop iterations to execute on each core and executes an implicit barrier at the end of the loop, unless explicit **`nowait`** clause. Computation of the loop bounds costs 15 cycles, however the total overhead is dominated by the barrier where the counters clearly outperform the hardware mutex by factors two to five (Table I).

C. Barriers

We implement two alternatives for **`omp barrier`**. The first implementation is a centralized sense-reversing barrier which uses a Xilinx hardware mutex IP (hwmtx, Table I) that offers *test-and-set (TAS) like* operations in a "fast" non-cacheable address space (SRAM). The second implementation uses our hardware counters (ctr, Table I): the counter is initialized to minus the number of participating cores. Each core that reaches the barrier, posts a message to increment the counter by 1. When the counter becomes zero, then it automatically triggers a response that sends a barrier release flag to all participating cores in their scratchpad.

A counter-based barrier takes 100 cycles on 4 cores and is up to $5.4\times$ faster than the Xilinx hardware mutex. The latter implementation generates superfluous traffic in the memory bus and increases contention and latency. The counter-based barrier requires 10 clock cycles to initiate the "increment" message, while the remaining time is NoC latency for the counter updates to arrive and trigger notifications that are delivered to all participating cores.

D. Critical and atomic sections

The **`omp critical`** directive specifies a critical section in an OpenMP program. Additionally, the **`omp_lock_t`** datatype and the **`omp atomic`** directive are used to perform atomic updates on shared variables. We implement all these operations using two alternatives: (i) using the hardware mutex and (ii) using MRQs to implement a lock [4]. The MRQ initially contains one token that all cores try to read. A dequeue from the MRQ implements a lock acquisition and the core that successfully acquires the token (dequeue) enters the critical section. Upon exit from the critical section, the token is placed back in the MRQ (enqueue).

TABLE I: OpenMP directive overheads (clock cycles)

# cores	`parallel`				`for`		`barrier`		`critical`		`atomic`		`reduction`	
	q-ctr	scr-ctr	q-scr	scr-scr	hwmtx	ctr	hwmtx	ctr	mtx	q	mtx	q	q	scr
1					127	102	108	79	180	93	180	94	125	50
2	195	109	199	113	220	110	201	87	66	67	200	116	308	152
3	231	137	233	140	359	119	337	94	76	65	304	128	355	178
4	269	148	266	162	561	126	539	100	91	65	447	134	426	210

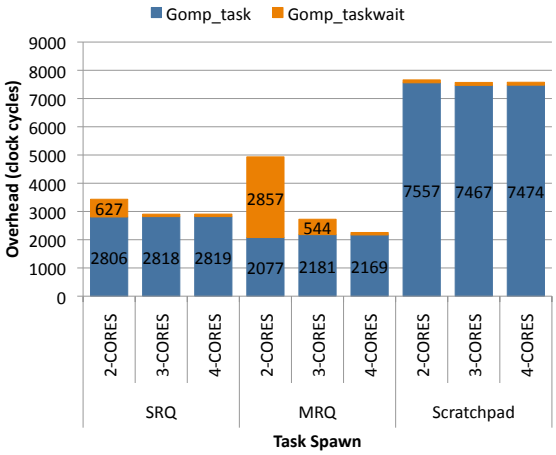

Fig. 2: Task directive overhead

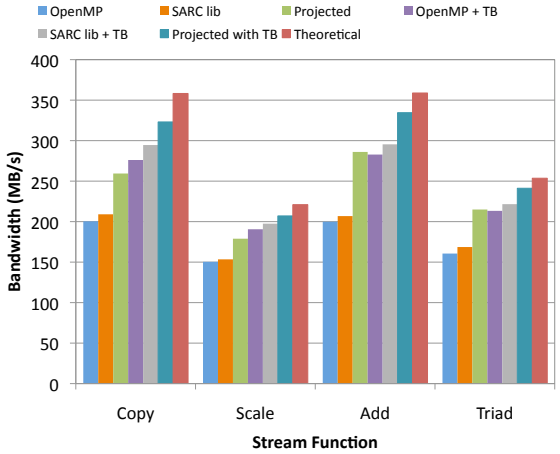

Fig. 3: STREAM bandwidth on the SARC prototype

The overhead of a lock-unlock pair on a single core is 93 clock cycles: locking costs 73 clock cycles and unlocking costs 20 clock cycles, Table I. The benchmark inserts a large "think time" inside the critical region and when multiple cores contend for the lock, then a portion of the lock overhead is overlapped; the lock requests are already waiting in the MRQ. The overlapped latency of these operations is 65 to 67 cycles. The overhead of the **`atomic`** directive is measured in a similar way, however the critical section in this case includes a single integer addition, therefore the overlap between the lock overhead and the critical section is negligible, Table I.

E. Reductions

We implement two alternatives for reductions: (i) using an SRQ or (ii) using per-core scratchpad locations, both located in the master core. All cores calculate their local portion of the reduction and send their values either in the SRQ with a message (i), or in their dedicated scratchpad location with remote stores (ii). The master gathers the partial results and computes the final value. Using dedicated scratchpad locations with remote stores is up-to 2× faster than using an SRQ, since the overhead of generating messages is eliminated, Table I.

F. Tasks

Spawning tasks is similar to spawning parallel regions and we evaluate three alternative implementations: (i) spawning tasks in per core SRQs, (ii) spawning tasks in a central MRQ on the master core, and (iii) spawning tasks in lock (fastest implementation) protected scratchpad areas per slave core. Task completion is signaled in hardware counters.

Figure 2 presents the overheads for spawning 64 empty tasks for the three alternative implementations. Task wait time decreases as we increase the number of cores because tasks are processed with a higher throughput. The average task spawning time is 45 clock cycles in (i), 35 clock cycles in (ii), and 118 clock cycles in (iii). In case (iii), the lock overhead dominates the task spawning latency and thus it cannot be amortized by very small tasks. On the other hand, spawning tasks in MRQ is 25% faster than spawning in SRQs, since tasks are generated locally.

IV. Application Benchmarks

We have parallelized two OpenMP application benchmarks, using the Rose compiler framework to automatically analyze the shared and private data accesses and generate the necessary data transfers to and from scratchpad memories.

A. Streaming kernels

The OpenMP version of the Stream [13] benchmark measures memory bandwidth using computation kernels (copy, add, scale, triad) with a high ratio of bytes per operation. We evaluate four implementations of Stream where 3 cores transfer data from the scratchpad of an idle core. The first implementation uses the SARC OpenMP runtime library and the second uses the low-level SARC prototype communication library [4] with the purpose of minimizing control overhead. The other implementations use same back-end and apply triple buffering to overlap communication with computation.

Figure 3 presents measurements of bandwidth for Stream with 3 cores transferring data from the scratchpad memory of an idle core. In addition we plot the maximum theoretical and projected bandwidth (model with basic queuing theory and simple assumptions). The difference between measured and projected bandwidth is in the range of 5% to 20%, due to

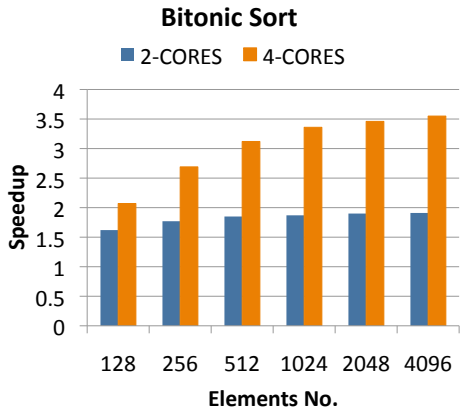

Fig. 4: Bitonic sort speedup on the SARC prototype

approximated NoC contention in our model. The additional overhead of OpenMP in all cases amounts to around 300 clock cycles. Overall, the SARC OpenMP does not impose heavy runtime overhead using near optimal of the available bandwidth of the architecture.

B. Bitonic Sort

We implemented an OpenMP version of bitonic sort on the SARC prototype and present the speedup achieved relative to the sequential version when using 2 and 4 cores. The input sizes range from 128 to 4096 integer elements and produce extremely fine-grain parallel tasks. We use remote DMAs for the butterfly data exchange during bitonic merge. The speedup of the SARC OpenMP implementation is satisfactory (though not perfect due to the overhead of task spawning, barriers and fine-grain data transfers) and amortizes the runtime implementation and saturates relatively fast, Figure 4.

V. Related Work

The Cell processor uses remote DMAs, mailboxes and signal registers, which enable exchange of control messages between the PowerPC (PPE) and the SPEs [3]. Our work explores a more extensive family of on-chip communication primitives (remote stores, hardware event queues with automatic responses, hardware counters) and their use in the runtime systems of high-level parallel programming models.

OpenMP compiler technology for the Cell processor [8] and compiler and runtime technology for architectures with explicitly managed local memories [14] implement techniques based on array reference analysis to automate data transfers from DRAM to local memories and to improve temporal locality by avoiding off-chip data transfers. Our work is directed towards using on-chip communication mechanisms to enable fine-grain parallelism in OpenMP programs and complements compiler technology for generating and optimizing off-chip data transfers. We leverage compiler analysis of array references to automatically generate and overlap data transfers in our OpenMP implementation.

Hardware support for fine-grain task-level parallelism, such as hardware task queues and message queues [1], [2], is advocated for the exploitation of fine-grain parallelism in many-core processors. In the same context, related work proposes application-specific hardware task schedulers [15]. Our work explores general-purpose hardware primitives, with several potential uses, and compares them in alternative implementations of a software stack for high-level parallel programming.

VI. Conclusions

The exploration of the implementation design space of OpenMP directives on the SARC prototype, demonstrates OpenMP task creation and synchronization latencies in the order of a few tens of processor clock cycles. We further demonstrate through application benchmarks the value of using separate hardware primitives for fine-grain communication and bulk communication, as well as the importance of low-latency communication for the exploitation of fine-grain parallelism. Overall, our OpenMP implementation on SARC approaches closely the theoretical peak performance for bandwidth-bound code and parallelizes effectively codes with task granularity of tens of clock cycles.

References

[1] S. Kumar, C. J. Hughes, and A. Nguyen, "Carbon: Architectural Support for Fine-grained Parallelism on Chip Multiprocessors," in *ISCA '07: Proc. of Intl. Symposium on Computer Architecture*, 2007, pp. 162–173.

[2] D. Sanchez, R. M. Yoo, and C. Kozyrakis, "Flexible Architectural Support for Fine-grain Scheduling," in *ASPLOS '10: Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2010, pp. 311–322.

[3] O. Takahashi *et al.*, "The Circuit Design of the Synergistic Processor Element of a CELL Processor," in *ICCAD '05: Proc. of the IEEE/ACM Intl. Conference on Computer-Aided Design*, 2005, pp. 111–117.

[4] S. Kavadias, M. G. Katevenis, M. Zampetakis, and D. S. Nikolopoulos, "On-chip Communication and Synchronization with Cache-Integrated Network Interfaces," in *Proc. of the 2010 ACM International Conference on Computing Frontiers (CF)*, FORTH-ICS, Heraklion, Greece, 2010.

[5] E. Ayguadé *et al.*, "The Design of OpenMP Tasks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 20, no. 3, pp. 404–418, 2009.

[6] C. E. Leiserson, "The Cilk++ Concurrency Platform," in *DAC '09: Proc. of the 46th Design Automation Conference*, 2009, pp. 522–527.

[7] A. Duran, M. Gonzàlez, and J. Corbalán, "Automatic Thread Distribution for Nested Parallelism in OpenMP," in *ICS '05: Proc. of the 19th International Conference on Supercomputing*, 2005, pp. 121–130.

[8] K. O'Brien, K. O'Brien, Z. Sura, T. Chen, and T. Zhang, "Supporting OpenMP on Cell," *International Journal of Parallel Programming*, vol. 36, no. 3, pp. 289–311, 2008.

[9] G. Kalokairinos, V. Papaefstathiou, G. Nikiforos, S. Kavadias, M. Katevenis, D. Pnevmatikatos, and X. Yang, "FPGA Implementation of a Configurable Cache/Scratchpad Memory with Virtualized User-Level RDMA Capability," in *IC-SAMOS09: Proc. of the IEEE International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*, Samos, Greece, Jul. 2009.

[10] T. P. Chunhua Liao, Daniel J. Quinlan and B. de Supinski, "A ROSE-based OpenMP 3.0 Research Compiler Supporting Multiple Runtime Libraries," in *Intl. Workshop on OpenMP (IWOMP)*, 2010.

[11] "GCC OpenMP," http://gcc.gnu.org/wiki/openmp.

[12] "EPCC OpenMP Microbenchmarks 2.0," http://www.epcc.ed.ac.uk/research/openmpbench.

[13] J. D. McCalpin, "Memory Bandwidth and Machine Balance in Current High Performance Computers," http://www.cs.virginia.edu/~mccalpin/papers/balance/, 1995.

[14] T. J. Knight, J. Y. Park, M. Ren, M. Houston, M. Erez, K. Fatahalian, A. Aiken, W. J. Dally, and P. Hanrahan, "Compilation for Explicitly Managed Memory Hierarchies," in *PPoPP '07: Proc. of the 12th Symp. on Principles and Practice of Parallel Programming*, 2007, pp. 226–236.

[15] G. Al-Kadi and A. S. Terechko, "A Hardware Task Scheduler for Embedded Video Processing," in *HiPEAC '09: Proceedings of the 4th International Conference on High Performance Embedded Architectures and Compilers*, Berlin, Heidelberg, 2009, pp. 140–152.