Hardware Scheduler Performance on the Plural Many-Core Architecture

Itai Avron and Ran Ginosar
Electrical Engineering Dept., Technion—Israel Institute of Technology, Haifa 32000, Israel

***Abstract* – The Plural many-core architecture combines hundreds of simple cores, lock-free shared memory, hardware scheduler and a task-based programming model. The hardware scheduler enables fast scheduling and allocation of fine grain tasks to all cores. Scheduler performance is evaluated based on an architectural simulator and on multiple benchmarks representing a wide variety of inherent parallelism. Several architectural alternatives and scheduler configurations are simulated. It is shown that a scheduler with capacity to schedule and terminate 10 task-instances per cycle, along with a task queue of as little as two slots near each core, is sufficient to utilize 256 cores.**

***Keywords* – hardware scheduler; many-core; performance; task queues; task graph**

1 Introduction

Many-core architectures come in different flavors: a two-dimensional array of cores arranged around a mesh NoC, GPUs with clusters of cores, and rings. This paper discusses the Plural architecture [1][14][15][16][17], in which many cores are interconnected to a many-port shared memory rather than to each other.

Many cores also differ on their programming models, including PRAM-like shared memory and CSP-like message-passing. Memory access and message passing also relate to data dependencies and synchronization—locks, bulk-synchronous patterns and rendezvous. The Plural architecture employs a strict shared memory programming model.

The last defining issue relates to task scheduling—allocating tasks to cores and handling task dependencies. Scheduling methods include static (compile time) scheduling, dynamic software scheduling, architecture-specific scheduling (e.g., for NoC), and hardware schedulers, as in the Plural architecture, in which data dependencies are replaced by task dependencies in order to achieve better performance, better efficiency and easier programming.

This paper addresses the performance of hardware scheduling on the Plural architecture and investigates potential scheduling acceleration techniques, including task queues at the cores, reducing scheduling latency, increasing scheduler capacity (issue and commit width) and using expected execution-time to affect scheduling [18]. The study is based on cycle-accurate simulation of the entire many-core system executing complete applications.

The rest of this paper is organized as follows: Section 2 discusses related work. In Section 3 we present the Plural architecture. Proposed scheduling acceleration techniques are given in Section 4. In Section 5 we describe our simulation environment and benchmarks. Analysis of simulation results is presented in Section 6, and we conclude in Section 7.

MES '15, June 13 - 14, 2015, Portland, OR, USA

DOI: http://dx.doi.org/10.1145/2768177.2768184

2 Related Work

Tilera [2] employed static compile time scheduling [3], or a dynamic scheduler [4]. A more general treatment is given in [5]. Static scheduling in a many-core during compile time [3] cannot adapt to varying run-time circumstances. Software scheduling (work-stealing [6] or lazy-scheduling [7]) may not scale well because of very high rates of task allocations and terminations (commits) . Hardware schedulers should enable dynamic, flexible, adaptive execution of fine-grain tasks on many-cores [8][9][10][11][12]. Hardware scheduling by prefix-sum logic was employed in XMT [13], which was limited to executing multiple instances of only one task at a time.

3 Plural Architecture

This section presents the Plural architecture (Figure 1), the programming model, and the hardware scheduler.

3.1 *Architecture*

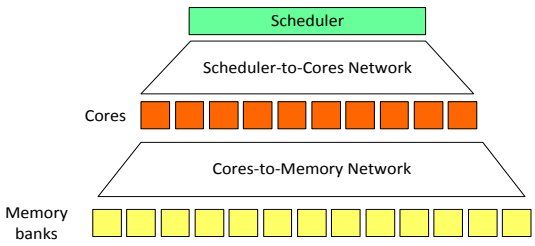

Figure 1: Plural many-core architecture

The Plural architecture is a shared-memory single-chip many-core system [1][14][15][18]. Emerging implementations include the MacSpace EC-FP7 project [16] and RC64 [17]. The Plural many-core consists of a hardware synchronization and scheduling unit, tens to hundreds of simple cores, and a shared on-chip memory accessible through a high-performance logarithmic

interconnection network. The cores may contain instruction and data caches; the latter is flushed and invalidated by the end of each task execution, guaranteeing consistency of the shared memory. In addition, the cores may contain private 'scratchpad' or 'tightly coupled' memories. The cores are designed for low power operation using 'slow clock' (typically slower than 500 MHz). Performance is achieved by high level of parallelism rather than by sheer speed, and access to the on-chip shared memory across the chip takes only a small number of cycles.

The on-chip shared memory is organized in a large number of banks, to enable many ports that can be accessed in parallel by the many cores, via the network. To reduce collisions, addresses are interleaved over the banks. The cores are connected to the memory banks by a multi-stage many-to-many interconnection network. The network detects access conflicts contending on the same memory bank, proceeds serving one of the requests and notifies the other cores to retry their access. The cores immediately retry a failed access. Two or more concurrent read requests from the same address are served by a single read operation and a multicast of the same value to all requesting cores.

3.2 *Programming Model*

The Plural PRAM-like programming model is based on non-preemptive execution of multiple sequential tasks. The programmer defines the tasks, as well as their dependencies and priorities which are specified by a (directed) *task graph*. Tasks are executed by cores and the task graph is 'executed' by the scheduler.

Concurrent tasks (namely, tasks independent of each other that may execute together or at any order) may be prioritized by the programmer. Expected task durations may optionally be indicated in the task graph.

Some tasks (typically amenable to data parallelism) may be *duplicable*, accompanied by a *quota* that determines the number of instances that should be executed (declared parallelism [7]). All instances of the same duplicable task are mutually independent and concurrent, and hence they may be executed in parallel or in any arbitrary order. These instances are distinguishable from each other merely by their *instance number*. Concurrent instances do not write-share data. Ideally, their execution time is short (fine granularity). Concurrent instances can be scheduled for execution at any (arbitrary) order, and no priority is associated with instances.

Each task progresses through at most four states. Tasks that depend on predecessor tasks start in the *pending* state. Once all predecessors to a task have completed, the task becomes *ready* and the scheduler may schedule its instances for execution and allocate (dispatch) the instances to cores. Tasks without predecessors (enabled at the beginning of program execution) start in the *ready* state. Once all instances of a task have been allocated, the task is *completely allocated.* And once all its instances have terminated, the task moves into the *finished* state (possibly enabling successor tasks to become *ready*).

The task graph may include iterations, repeating certain portions of the graph until some conditions are met. No DAG assumption is made. Figure 2 shows the task graphs that are studied in this paper. Squares represent tasks (named A,B,C,…) and show the number of required duplications and the (average or assumed) number of cycles it takes one instance to complete. Arrows represent task dependencies and rhombi represent conditions. In the "Shared Variable" benchmark (a), for example, the condition controls looping: The scheduler goes back to task A (for another invocation) four times, and then proceeds to task F.

Data dependencies are expressed (by the programmer) as task dependencies. For instance, if a variable is written by task t_w and must later be read, then reading must occur in tasks $\{t_r\}$ and $t_w \rightarrow \{t_r\}$. The synchronization action of completion of t_w prior to any execution of tasks $\{t_r\}$ provides the needed barrier.

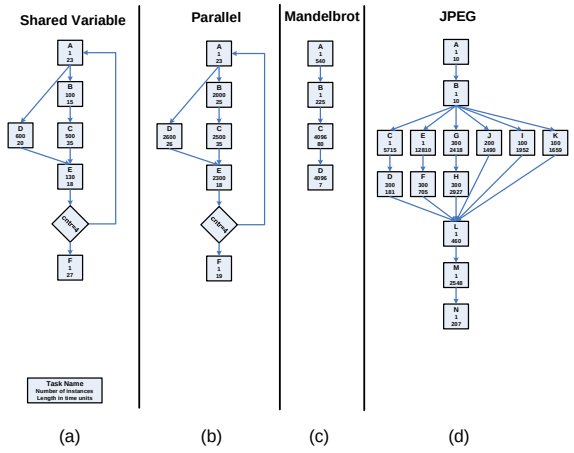

Figure 2 : Benchmark task graphs
(a) Shared Variable, (b) Parallel, (c) Mandelbrot, (d) JPEG

3.3 *Scheduler*

The hardware scheduler assigns tasks to cores for execution. A core which completes its task sends a termination message to the scheduler. The scheduler then allocates a new task to the core according to the task graph. Thus, the two possible states of each core, as managed by the hardware scheduler, are *Idle* and *Busy*. The scheduler communicates with the cores over the Scheduler-to-Cores Network (Figure 1).

The *scheduler capacity*, namely the number of simultaneous tasks which the scheduler is able to allocate or terminate during each cycle, is limited. Any additional task allocations and task termination messages beyond scheduler capacity await the following cycles in order to be processed. A core remains idle from the time it issues a termination message until the next task allocation arrives. That idle time comprises not only the delay at the scheduler (wait and

processing times) but also any transmission latency of the termination and allocation messages over the scheduler-to-cores network.

4 Scheduler Modifications

We investigate by simulations the following modifications:

1. *Enhancing scheduler capacity:* A variable specifies capacity for terminations (commit width) and allocations (issue width).
2. *Reducing scheduling latency:* Overall latency between core termination and next allocation.
3. *Adding task queues to each core:* A task queue near each core may eliminate idle waiting between termination and next allocation.
4. *Execution time aware scheduling:* A binary normal/long hint of task length in the task graph enables the scheduler to prioritize long tasks.

The scheduler can allocate and receive termination messages of a configurable number of tasks instances. The allocation and termination algorithms are shown in Figure 3.

1. Find all *Ready* tasks.
2. Choose one of the *Ready* tasks (according to priority or by task index).
3. While there is still enough scheduler capacity
   a. Find core queue with fewest instances (choose lowest index queue)
   b. Allocate an instance to that queue
   c. Increase counter of instances in that queue
   d. Increase counter of allocated task instances
   e. If a task is *Completely Allocated,* continue to next task

1. Choose lowest index core which has sent a termination message
2. While there is still enough scheduler capacity
   a. Process termination message
   b. Decrease counter of instances in queue
   c. Increase counter of finished task instances
   d. If task *Finished*, find new tasks eligible to run, change them to *Ready*
   e. Continue to next core

Figure 3 **: Allocation (top) and termination (bottom) algorithms**

5 Simulation Environment

We used an in-house cycle-accurate architectural simulator of 256 cores and 256 memory banks. Four benchmark programs were simulated on 24 different configurations of the architecture: task queue depth of 0,1,2,10, scheduler capacity of 5,10,∞, and latency of 0,20 cycles from scheduler to cores. Other values of these parameters turned out insignificant. In individual benchmark analysis below, only cycle latency of 20 is considered.

Four programs were tested (Figure 2). The first two share the same task graph. In *Shared variable*, all instances of one duplicable task write into the same memory bank, causing many collisions, whereas in *Parallel* there is no write sharing. The remaining benchmarks are a *Mandelbrot* set and *JPEG* image compression (160×160 image). Benchmarks were designed specifically for this study, to take full advantage of the parallelism offered by the Plural architecture and to expose scheduling issues.

6 Analysis of Simulation Results

Figure 5 shows activity per core for the parallel benchmark, for queue size of 0,1 and capacity 5,10. Latency is incurred by both allocation and termination messages. Total run time drops as we add a one slot queue (left). Clearly, the queue helps hiding scheduling latency. The capacity 5 scheduler is unable to utilize all cores, not taking advantage of declared parallelism. Increasing capacity to 10 (bottom) results in lower idle time (yellow).

Notice the imbalanced work distribution in the top right chart. Lower index cores, which receive the first tasks, also receive a second task into their queues before they finish their jobs. In higher index cores, the second task arrives only after they already finished their work. Thus, queues help hide latency only if scheduler capacity is sufficiently high.

In the shared variable benchmark, all cores write to the same memory bank, resulting in many collisions. Interestingly, higher scheduler capacity (bottom, Figure 6) actually degrades performance (red, collisions). The low capacity scheduler spreads access times to shared bank, so collision rate is reduced. The high capacity scheduler enables more simultaneous accesses, increasing collisions.

In the JPEG benchmark, employing a task queue for the cores significantly degrades system performance (Figure 7). The scheduler capacity itself has no effect, due to task long run times, which enable the scheduler to reach the high index cores before any low index core finishes its work.

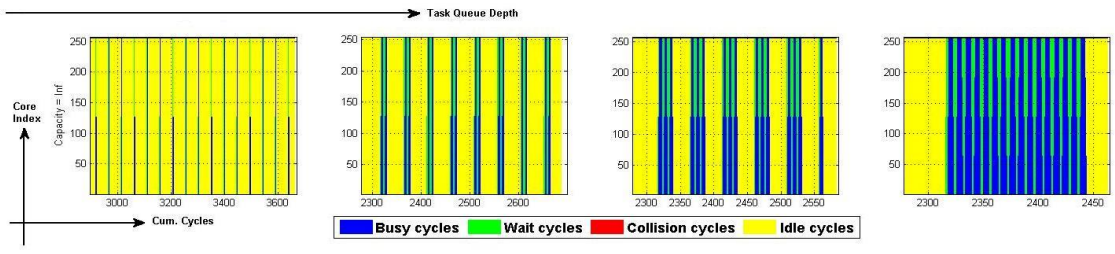

Figure 4 : Activity per cycle, Mandelbrot. Queue=0,1,2,10. Zoom-in on task D execution, infinite capacity

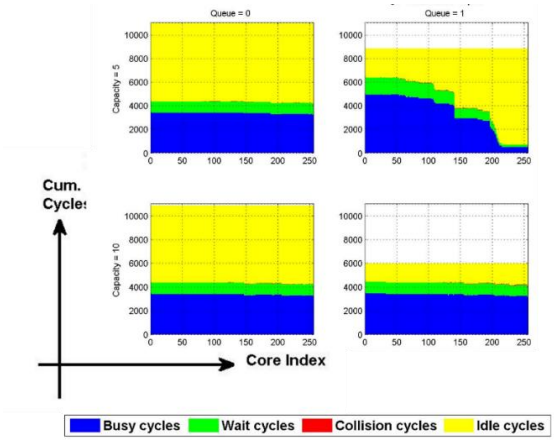

Figure 5 : Activity per core in Parallel benchmark

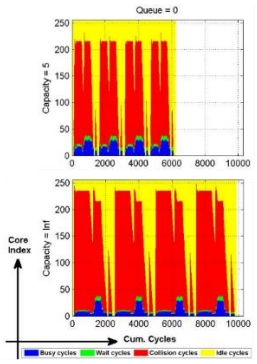

Figure 6 : Activity per cycle in Shared variable benchmark

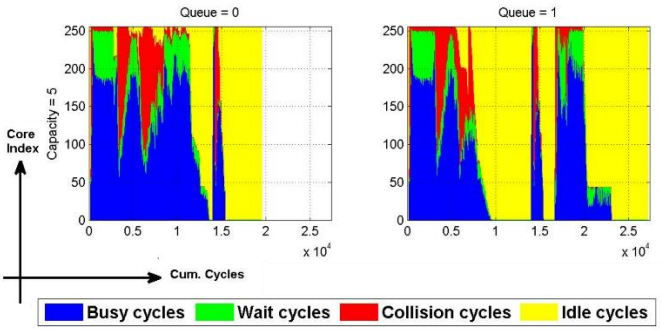

Figure 7 : Activity per cycle in JPEG benchmark

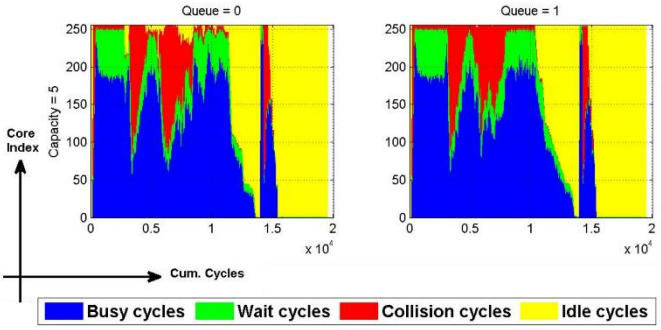

Figure 8 : Activity per cycle, JPEG, E & C flagged long

When the scheduler is aware of expected length of execution of each task, it avoids queueing tasks to a core where a long task is allocated. In Figure 8, task E is flagged long and total execution is shorter even with a queue.

In Mandelbrot benchmark (Figure 4), while task D (4096 instances of a 7 cycles task) executes, there are pauses. With no queue the pauses are incurred by task allocation latency. But even a two slots queue does not help to hide that latency, and only the 10 slots queue does. While fine granularity is nice to have, it requires deep queues and a powerful scheduler of very high capacity to assign instances fast enough to hide latencies.

7 Conclusions and Future Work

We have analyzed how the hardware scheduler affects performance of the Plural many-core architecture. We introduced task queues at the cores and considered varying scheduler capacity. A cycle-accurate simulator enabled analyzing different benchmarks and architectural modifications. We have shown that task assignment latency can degrade performance. To hide that latency, we added task queues at the cores. At times, these queues may degrade performance. Execution-time aware scheduling was shown effective in such cases. Additional studies are discussed in [18]. Future research may address a blocking cores-to-memories network, other scheduler distribution networks such as tree and mesh, implications of scheduling on power and profiling for scheduling optimization.

References

[1] Bayer and Ginosar, US Patent 5,202,987, 1993.
[2] Wentzlaff *et al.*, *IEEE micro* 5:15-31, 2007.
[3] Lee *et al.*, ASPLOS-8, 1998.
[4] Waddington *et al.*, SFMA 2011.
[5] Zydek and Selvaraj, ITNG 2009.
[6] Blumofe and Leiserson, *J. ACM*, 46.5:720-748, 1999.
[7] Tzannes *et al.*, *ACM TOPLAS* 36(3), 2014.
[8] Crummey *et al.*, CONTROL'94, 1098-1103, 1994.
[9] Kim and Smith, ISCA-29, 2002.
[10] Yu *et al.*, SNPD 2009.
[11] Etsion *et al.*, MICRO-43, 2010.
[12] Trancoso *et al.*, *Int. J. Parallel Programming*, 34(3):213-235, 2006.
[13] Wen and Vishkin, 5th Conf. Computing Frontiers, 2008.
[14] Bayer and Ginosar, "Tightly Coupled Multiprocessing: The Super Processor Architecture," Springer Japan, 2002.
[15] Bayer and Aviely, US patent 8,099,561, 2012.
[16] http://www.macspace.eu/
[17] http://www.ramon-chips.com/RC64brief.Feb2015.pdf
[18] Avron and Ginosar, HPCC-ICESS 2012.