Performance of a Hardware Scheduler for Many-Core Architecture

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

Abstract – **A hardware scheduler for many-core architectures enables fast scheduling and allocation of fine granularity tasks to all cores. We present performance evaluation of a hardware scheduler for HyperCore, a many-core architecture. The evaluation is based on an architectural simulator, using multiple benchmarks representing a wide variety of inherent parallelism. Several architectural improvements are proposed, and various configurations of the scheduler are simulated. The results are analyzed, and are used to highlight the potential and the possible pitfalls of the architecture. It is shown that a scheduler with a capacity to schedule and terminate 10 instances per cycle, along with a task queue of as little as two slots near each core, is sufficient to utilize 256 cores. Other scheduling configurations are also analyzed.**

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

1 INTRODUCTION

Today, chips are produced with a small number of cores (multi-core). This enables handling them similarly to a uni-processor; memory access and scheduling of processes to cores are similar to what is done in a single core. But to keep up with the need for increased performance, the number of cores will have to grow. Several many-core architectures already exist. Such architectures include NVIDIA [*1*] and ATI GPUs and Tilera tiles [*2*], as well as proposals such as Plurality HyperCore [*3*], XMT [*4*], Intel Larrabee [*5*] and 2PARMA [*6*].

Unlike current multi-core solutions, many-core architecture requires new tools and techniques in order to be utilized efficiently. While many research works handle the subject of network, communication, memory [*7*] and cache [*8*] performance problems, the scheduling problem is somewhat less investigated. Unlike multi-core systems, scheduling in a many-core system requires great efficiency in order to utilize hundreds or even thousands of cores. This need prohibits the use of a software scheduler, since it imposes a large overhead on the scheduling process. While solutions that incorporate static scheduling during compile time exist [*9*] [*10*] [*11*], they cannot dynamically schedule tasks according to run-time workload. Hardware schedulers, on the other hand, enable such advantages and hold more promise for efficient many-cores [*4*] [*12*] [*13*] [*14*] [*15*] [*16*].

In this research, we present performance analysis of a hardware scheduler. We study the HyperCore architecture, and find the rules useful for developing a hardware scheduler for many-core architecture [*17*]. By analyzing the phenomena that occur in multiple benchmarks, we are able to highlight the pitfalls that a many-core designer might stumble upon. Several architectural improvements are proposed, including a queuing hierarchy. Scheduler parameters such as the scheduling latency, the scheduler capacity and task queue depth are simulated, and insights about their effect on overall system performance are presented.

The rest of this paper is organized as follows: Section 2 discusses related work. In Section 3 we present the HyperCore architecture. Proposed solutions for scheduler limitations are given in Section 4. In Section 5 we describe our simulation environment and benchmarks, as well as explain our format of presetting simulation data. Analysis of simulation results is presented in Section 6, and finally we conclude and describe future work in Section 7.

2 RELATED WORK

Several many-core architectures have been investigated. Tilera [*2*] offers a tiled architecture following RAW [*18*]. Multiple identical, programmable tiles having local caches are connected by multiple mesh NoCs. Extensions are proposed in [*19*] [*20*], and a use case is presented in [*21*]. It employs static compile time scheduling [*9*] that orchestrates parallelism within a basic block across tiles and also handles control flow across basic blocks. Processes are allocated to rectangular-shaped tile groups. Compiler and allocation enhancements can be found in [*10*], [*22*]. Instead of static scheduling, this paper discusses a hardware scheduler that enables dynamic load-balancing. A dynamic scheduler for Tilera is presented in [*23*], where system threads execute tasks from a task queue. Still, scheduling is software based, incurring longer overhead than a hardware scheduler. A more general treatment of NoC-based scheduling is given in [*24*]. In Rigel [*25*], another many core architecture with task-queue software scheduling, the cores are organized hierarchically in clusters and a bulk-synchronous programming model is employed.

NVIDIA Fermi [*1*] GPU comprises multiple SIMD processors. A hardware scheduler switches threads in order to hide memory latency. GPUs are efficient when processing many threads with same control flows. A task-based dynamic load-balancing solution is proposed in [*26*], using a persistent kernel which executes tasks from a task queue. A method for load-balancing across the CPU as well is given in [*27*]. Intel Larrabee [*5*], another GPU, combines many cores instead of many threads. The similar architecture Intel MIC is described in [*28*], demonstrating a distributed task-stealing software scheduler. In contrast, this paper investigates a hardware scheduler.

XMT [*4*] [*29*] uses a programming model based on PRAM with arbitrary CRCW (concurrent read concurrent write) SPMD and incorporates hardware prefix-sum logic to schedule same-code threads. It shows good performance for fine-grained tasks and irregular applications. Unlike our architecture, XMT threads explicitly call the hardware scheduler (using a PS instruction). Further, XMT can execute the multiple instances of only one task at a time. Grid Processor Architectures (GPA) [*30*] consists of a two-dimentional array of ALUs, each with limited control, connected by a thin operand network. Compilation and scheduling of instructions to ALUs is static, whereas execution is dynamic in dataflow order.

Hardware schedulers for a small number of cores are described in [*12*] and [*13*]. In the latter, hardware steering logic allocates strands (chains of dependent instructions) to cores based on inter-strand dependencies. Hardware scheduling for Godson-T is described in [*31*] [*14*], comparing also fine grain to barrier synchronization and managing instance dependencies in addition to coarser task dependencies. Carbon [*32*] and ADM [*33*] use hardware task queues to support scheduling. In the Data-Driven Multithreading (DDM) [*34*], [*16*] design, a hardware mechanism provides data-driven thread synchronization for multi-threaded architectures that uses control flow processors. Scheduling follows a task map created by the programmer, using a producer-consumer programming model. Task Superscalar [*15*] generalizes the concept of instruction-level out-of-order execution to tasks, detecting task-level parallelism in run time, but software scheduling incurs high overhead. Real-time scheduling manages not only starting tasks but also constraints on finish time of tasks. Hardware real-time schedulers are described in [*35*], [*36*]. Energy-efficient real-time scheduling is presented in [*37*]. Finally, the Tatung fine grain scheduler (TFGS) [*11*] operates at the machine instruction level, using a data/control dependency graph, a branch nest tree and a priority list to create a static scheduling prior to execution. At runtime, test bits synchronize the processors and notify of branch decisions.

3 HYPERCORE ARCHITECTURE

This section presents the HyperCore architecture, the programming model, and the hardware scheduler.

3.1 *Architecture*

The HyperCore architecture (Figure 1) is a shared-memory many-core system [*3*] [*38*] [*39*] [*40*]. It has a hardware synchronization and scheduling unit, 16-256 RISC cores, and a shared on chip memory that is accessed through a high-performance interconnection network.

The cores themselves are simple general-purpose 32-bit RISC processors. The cores do not have any data cache, and thus no coherency logic is needed. A small instruction cache is used to enable efficient access to code. The cores use blocking data load and store with no out-of-order execution, and run each task instance to completion (no task switching is allowed).

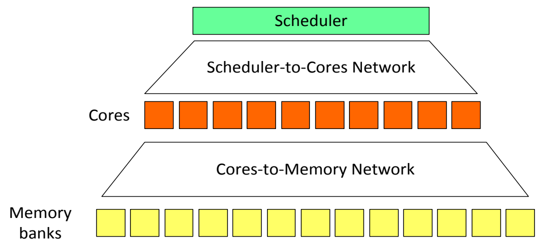

Figure 1: HyperCore architecture

The shared memory is organized in a large number of banks, to enable many ports that can be accessed in parallel by the many cores. To reduce collisions, addresses are interleaved over the banks. The cores are connected to the memory banks by a many-to-many interconnection network that can serve simultaneous accesses from all cores. 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.

All memory accesses from each core to each memory bank take a constant time of two cycles if there are no conflicts. Since the HyperCore is designed as a single clock system, the clock cycle time is limited by the longest wire delay between any core and any memory bank.

The possible states of each core are shown in Figure 2. A core starts in *Idle* state. Once allocated a task for execution, it becomes *Busy*. When it encounters a memory access operation, it is either *Waiting* (if the memory access will succeed) or *Colliding* (if it is about to collide). Once completing a task, it moves back to *Idle.*

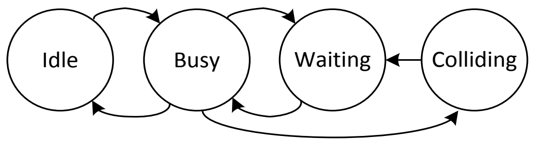

Figure 2 : Core state transition graph

3.2 *Programming Model*

The programming model of the HyperCore is based on multiple sequential tasks and their inter-dependencies. The programmer defines the tasks, as well as the list of dependencies, formulated as a (directed graph) *task map*. The tasks are executed by the cores, while the task map is executed by the scheduler. Some tasks may be *duplicable*, accompanied by a *quota* that determines the number of instances that should be executed; all instances are mutually independent and may be executed in parallel or in any arbitrary order. The instances are distinguishable from each other merely by their *instance number*. Ideally the instances do not share data, and their execution time is short (fine granularity). The scheduler distributes the tasks that are eligible for execution among the available cores at that moment.

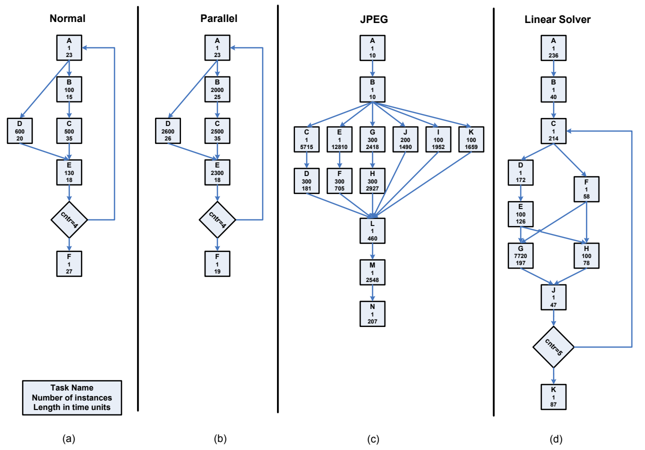

Figure 3 : Demo benchmarks task maps: (a) Normal, (b) Parallel. Benchmarks task maps : (c) JPEG, (d) Linear Solver

Figure 3 shows a few task maps. Squares represent tasks (named A, B, C, …) and show the number of required duplications and the number of cycles it takes for one instance to complete. Arrows represent task dependencies. A task is eligible to run only when all its predecessors have completed. The rhombus represents a condition and is executed only by the scheduler; there is no real code associated with condition tasks. In the "Normal" benchmark (a) for example, the condition controls task looping: The scheduler goes back to task A (for another invocation) for 4 times, and then proceeds to task F.

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 using the task map. The Scheduler communicates with the cores over the Scheduler Network (SN), as in Figure 1.

Each task progresses through four states, as in Figure 4. It starts as *pending,* when it waits for its predecessors to finish. It then becomes *ready* and the scheduler may schedule its instances for execution and allocate them to cores. Once all its instances have been scheduled, it is c*ompletely allocated.* And once all its instances have terminated it moves into the f*inished* state.

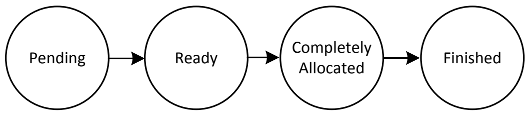

Figure 4 : Task states

The number of simultaneous tasks which the scheduler is able to terminate or allocate during each cycle is limited. Any additional termination message beyond the *scheduler capacity* awaits the following cycles in order to be processed. The same applies to any additional task allocations beyond the scheduler capacity. 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

As described above (Section 3.3), the scheduler (including the SN) may be limited in terms of capacities and latencies. In this research we investigate, by simulations, how these limitations may affect system performance, and explore possible solutions as follows:

1. *Enhancing scheduler capacity:* The scheduler can process only a limited number of termination and allocation messages each cycle. For fine granularity tasks, many tasks may need to be invoked or terminated simultaneously. Several ways can be used in order to achieve this increase, from enhancing the Scheduler to enhancing the SN. In this study we do not distinguish among these methods, and use the overall scheduler capacity as a single parameter. To simplify things further, we also assume a similar capacity for the termination and allocation processes. This assumption seems reasonable for a balanced Scheduler.
2. *Reducing scheduling latency:* Latency is incurred due to two factors: network delay between the scheduler and the cores and processing time in the scheduler. We study the overall latency between core termination time and next allocation time. Latency may be reduced by constructing a more powerful Scheduler and constraining the physical distance between the Scheduler and the cores.
3. *Adding task queues to each core:* A core is idle from the time it completes a task until the next task allocation arrives at the core. In order to cope with this problem, we suggest adding a task queue near each core. This queue will hold additional available tasks for the core. When the core finishes its current task, it already has another task available to start working on. Several queue depths are tested in this research in order to understand their effect on system behavior.
4. *Sharing queues:* After the addition of task queues, cases may occur when load imbalance degrades system performance. In order to implement a simple work-stealing algorithm, we propose sharing a task queue among several cores, thus creating core clusters. We show that this simple change in architecture restores balance to the system in most cases.

We study a modified HyperCore architecture comprising 256 RISC cores and 256 memory banks, and introduce a queue with variable depth near each core. The scheduler is able to allocate and receive termination messages of a configurable number of tasks instances. The allocation and termination algorithms are shown in Figure 5.

5 SIMULATION ENVIRONMENT

The study is based on an architectural simulator implemented in Matlab, developed in [7]. The simulator is cycle accurate and allows investigating parallel execution under different architectural variations. In this work, we have added the scheduler part to the above simulator, and implemented the modifications described in Section 4.

```
1. Find all Ready tasks.
2. Choose one of the Ready tasks.
3. While there is still enough scheduler capacity
   a. Find the core queue with fewest instances (if
      several such queues exist, choose the 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 the task is Finished, find new tasks that are
      eligible to run and change them to Ready state
   e. Continue to next core
```

Figure 5 : Allocation (top) and Termination (bottom) algorithms

Four benchmark programs (explained in Section 5.1) were simulated on 24 different configurations of the architecture, as follows:

- Four values of task queue depth: 0, 1, 2, 10
- Three values of scheduler capacity: 5, 10, infinite
- Two levels of latency between the scheduler and the cores: 0 and 20 cycles (in the following, only results related to 20 cycles latency are presented as they are more meaningful).

All configurations employed 256 cores and 256 memory banks. Preliminary simulations revealed that other values of task queue depth were insignificant: performance was quite similar for both 2 and 5 queue depths. Similarly, increasing that value beyond 10 did not affect the results much. Note that scheduling latency occurs both from the scheduler to the cores (for allocation messages) and from the cores to the scheduler (for termination messages).

5.1 *Benchmarks*

A benchmark program consists of tasks, some of which are *duplicable* and some are conditional. Running a program requires the program's task map and the code of each task. A task map includes the program task names, the task dependencies and the number of duplicated instances for each task.

Two demo programs were tested. These programs have the same task map with variations that are aimed to investigate a range of parallelism. The demo programs are:

- *Normal*: the program with a moderate number of duplications.
- *Parallel*: same as Normal, but with many more duplications for each task.

Figure 3 shows the task map of the demo programs and two additional benchmarks: JPEG image compression (image 160x160) and a Linear Solver benchmark. Task maps are explained above in Section 3.2.

5.2 *Simulation details*

For each of the above benchmarks, statistics have been gathered in order to explore the phenomena that can be encountered in real life programs. Computing only the total number of cycles each program took may mask the reasons for those results, preventing us from analyzing them properly. More detailed statistics were needed, to show what is done by each part of the architecture on each cycle of the program. The first statistics that were gathered show for each cycle how many cores are busy, idle, waiting or colliding. Through it, one can observe certain points within the program that are hard for the scheduler to handle. For example, points where the cores are idle though tasks that are ready to run are available. The second set of statistics tell the tale from the cores' point of view; for each core, the number of cycles the core was busy, idle, waiting or colliding is shown.

Results of the above statistics and analysis show that different programs behave differently. A detailed analysis of each benchmark is given in Section 6. Two kinds of graphs are presented, each for the appropriate statistics. The first one is "Activity per cycle" (e.g. Figure 9). In those graphs the X-axis shows cycles of program execution and the Y-axis shows cumulative core activity in each cycle. The second graph is "Activity per core" (e.g. Figure 6). In those graphs the X-axis is the core index, and the Y-axis shows the cumulative activity for each core.

For each of those graph, we show 12 charts related to the 12 configurations (task queue depth of 0, 1, 2 or 10 and scheduler capacity of 5, 10 or infinity). Queue depth increases towards the right-hand side of each graph and scheduler capacity increases when going down over the charts.

6 ANALYSIS OF SIMULATION RESULTS

We first describe simulation results for each of the four benchmarks, and then compare the results to each other.

6.1 *"Normal" Benchmark*

Figure 6 shows an "Activity per core" graph for the "Normal" benchmark in the 20 cycles latency scenario. Latency is incurred for both the allocation (Scheduler to cores) and termination (cores to Scheduler) messages.

Observe that there is a drop in the total run time (when the capacity is high enough) as we move from no task queue to only one slot per queue. The reason is that the queue acts as a buffer to hide some of the latency between the scheduler and the cores.

By examining the last line of charts (infinite capacity), it is possible to see that the overall run time increases from 1410 to 1470 cycles as the queues depth is increased. This result is unexpected, and may be explained as follows. Consider the scheduling balance among different cores, and observe that when the queue depth increases, the working load spreads unevenly among the cores. That is, lower index cores perform more work than higher index ones. Studying the statistics of that imbalance clearly shows the reason for the phenomenon. Each cycle, the imbalance in the scheduling is measured by counting the number of idle cores versus the number of ready tasks waiting to be executed by a busy core (namely, ready to run instances waiting in the queue of an already busy core). The results can be seen in Figure 7. As the queue gets deeper, more imbalance occurs. In the case of a no queue, no imbalance is possible (the scheduler assigns a new job to a core only after the last job has finished). In the case of low scheduler capacity, the scheduler does not work fast enough to fill more than one task per core (it schedules idle cores first), as already seen by the fact that it does not utilize the high index cores well. When unbalanced load does occur, it means that some cores are idle while there is a job to be done, and so time is wasted and the program total run time gets longer.

When we compare the results for the 20 cycles latency (Figure 6) against the no latency simulation (not shown), we find that the total run-time has increased (e.g. from about 950 cycles in the capacity=inf, queue depth=10 in the latency=0 case, to roughly 1470 cycles in the latency=20 case, Figure 6). Taking a second look into the task map helps answer this puzzle. The latency that is now inherent in the system means that every time the scheduler sends a task to the cores, 20 cycles pass before execution can start. Similarly, when a core completes the task and sends a termination message to the scheduler, again 20 cycles pass before the scheduler can send the next task. Therefore, in each synchronization point in the task map (where the scheduler waits for some tasks to finish in order to start scheduling new tasks), a 40 cycles latency is encountered. In this benchmark, for example (Figure 3), the scheduler needs to wait for both task C and task D to finish before task E can be assigned to the cores. This synchronization point's latency cannot be compensated for by the task queues, because only when the queues are all empty can the scheduler assign the new task. Counting all the synchronization points in the program and the fact that the task map is repeated four times, the difference of 500 cycles in the total run time can be accounted for.

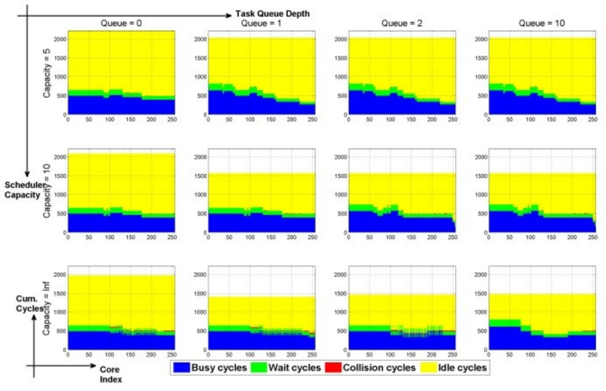

Figure 6 : Activity per core in Normal benchmark, Latency = 20 cycles

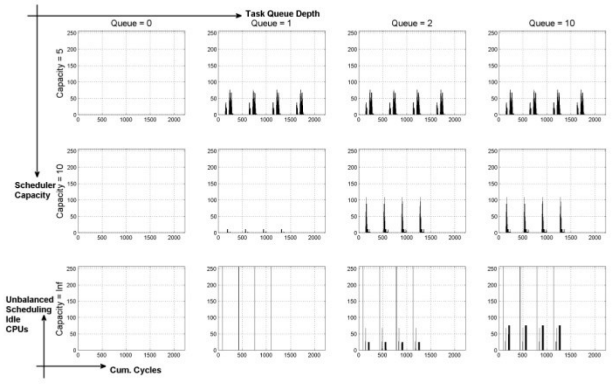

Figure 7 : Unbalanced scheduling per cycle in Normal benchmark, Latency = 20 cycles

6.2 *"Parallel" Benchmark*

The parallel benchmark (Figure 3(b)) is different from the normal benchmark (Figure 3(a)) in the number of instances each task has. Figure 8 shows the Activity per core for 20 cycles latency. In this case, the low capacity scheduler is still unable to utilize all the cores, thus not taking advantage of the vast parallelism. Increasing the capacity just by a little (to 10 instances per cycle, second row of Figure 8) enables the scheduler to reach its full potential as seen by the low idle time of the system. As for infinite capacity (last row of the figure), the number of tasks is so great that no unbalanced work distribution takes place (there is always work for everyone).

Here we witness again the effect of the 20 cycles latency on the program run time. When there is no task queue, the latency between the scheduler and the cores is just the same as extending the tasks run time. For instance, consider the capacity=inf, queue depth=0 case (bottom left chart in the figure) and assume perfect balancing. We have (2000+2500+2600+2300)×4 tasks (as per the task map), each suffering a 40 cycles latency. When distributing this latency over the 256 cores, we get an additional idle time of $(2000 + 2500 + 2600 + 2300) * 4[repeats] * \frac{40[cycles]}{256[cores]} \approx 5900[cycles]$. On the other hand, adding as little as two slots to each core's queue (from queue depth=0 to 2, third column in the figure) enables hiding most of the latency and obtaining results similar to those with no latency at all. The only difference is the synchronization points where the latency cannot be compensated for.

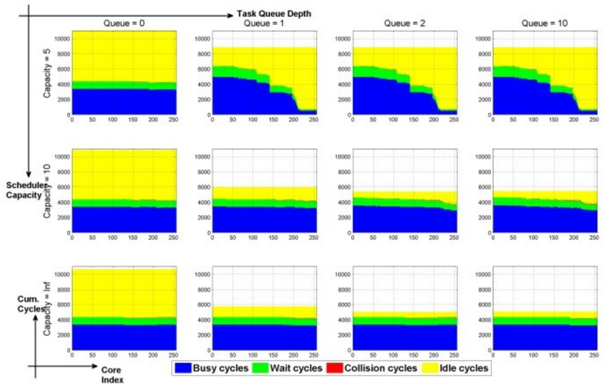

Figure 8 : Activity per core in Parallel benchmark, Latency = 20 cycles

Observing the first line of charts in Figure 8, we notice that work distribution is again unbalanced. The latency has effectively increased the tasks run time. In turn, this should have made the work more balanced, as now the scheduler has enough time to assign work to high index cores. However, this does not take into account the queues role. The lower index cores, which receive the first tasks, also get a second task into their queues before they finish their jobs. This second task does not suffer from any latency, since the cores had work to do until they got them. For the higher index cores, unfortunately, the second task comes only after they already finished their work. So, they suffer the latency penalty for both the report back to the scheduler and for the scheduling of the new task. This makes them work less than the other cores, and hence the results. We may conclude that queues help hide latency only if scheduler capacity is sufficiently high.

6.3 *JPEG Benchmark*

The JPEG benchmark leads to another strange phenomenon (Figure 9): employing a task queue for the cores significantly degrades system performance. 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. Consider scheduling imbalance; the charts in Figure 10 show the number of cores, each cycle, which remain idle although work ready for execution is waiting in some other queues. Notice that in addition to the peaks, there are also tails that indicate a small number of ready tasks that are stuck in some queues when there are also idle cores elsewhere.

Note that at one point during the program execution all the cores are waiting for one core to complete (e.g. the one busy core between 10,000 and 15,000 cycles in the three columns on the right in Figure 10). This period matches exactly the time in which almost all cores are idle (yellow sections in Figure 9). When considering the JPEG task map (Figure 3), we notice that task E is much longer than all other tasks. When there is no queue, this task is executed along the other tasks, and when it finishes only task F is left to run (the second block of work visible in the left column in Figure 9). With a non-zero queue, however, another instance is assigned to the core that works on task E. This instance comes from task G in this case. Now, when task E is finally over, task F is ready to run. But task G has not completed yet, and so task H is not assigned to any core. Only when the last instance of task G (which has waited for task E to finish) is over, can the scheduler start assigning task H (as can be seen in the third block of work being done on all the queued configurations). In conclusion, task queues may degrade system performance in some cases, and thus need to be treated with caution. Those cases occur when there are (very) coarse grained tasks in the task map.

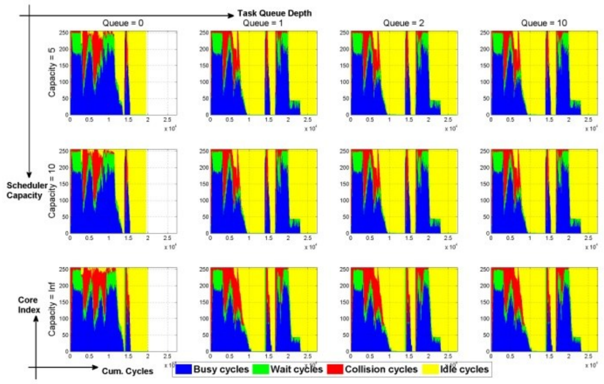

Figure 9 : Activity per cycle in JPEG benchmark, Latency = 20 cycles

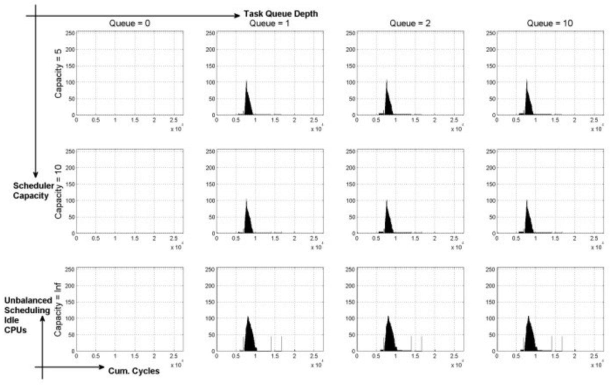

Figure 10 : Unbalanced scheduling per cycle in JPEG benchmark, Latency = 20

We can see in this example that even though only one instance suffers from unbalanced scheduling between the cores, the results can be quite devastating to overall system performance. In order to cope with this scenario, several solutions are suggested:

1. Queue sharing among multiple cores, requiring more complex hardware

2. Using fine granularity tasks
3. Scheduling awareness of long tasks [*41*], possibly requiring a more complex scheduler
4. Task migration among queues, possibly requiring more complex hardware and enhanced communication bandwidth, and incurring higher power and latency
5. Task map optimization [*41*]
6. Pipeline multiple instances of an algorithm (e.g. image compression applied to a sequence of image segments) so that parallel sections of one instance overlap the serial bottlenecks of other instances.

The first two solutions have been simulated and are presented in the following sections. The other four solutions were not simulated, and are offered for future research.

6.3.1 Shared Queues

In order to handle unbalanced scheduling (where ready tasks wait for a core to finish its work although there may be other cores that are idle), queue sharing among several cores can be implemented. In that architecture, the scheduler assigns work to a cluster of cores, such that a task pool is formed, and each core in the cluster can pick up a new job from the pool once it has finished its old job.

In our case, each cluster contains two adjacent cores (having two consecutive index values). That is, cores 2*i* and 2*i*+1 share a common queue (*i* = 0, 1, 2, …, 127). The scheduler assigns tasks to empty queues first (namely, one task to the queue of cores 0 and 1, the next task to the queue of cores 2 and 3, etc.). In each cluster, the core which is idle can take the next task from the queue. The queues holding termination messages are not shared. Notice that a task queue depth of 1 means one job per core, which are two jobs per queue.

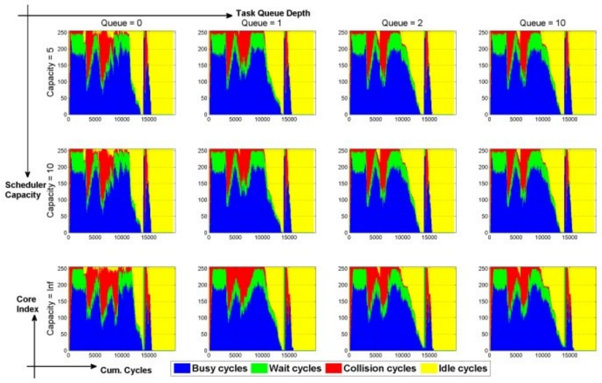

Figure 11 : Activity per cycle in JPEG benchmark with shared queues, Latency = 20 cycles

With shared queue (Figure 11), the waiting time for task E to finish is exploited working on other tasks, since task G does not get stuck behind it. This solves the problem in our case. Caution may be needed when employing this solution in the architecture, since scenarios still exist when a cluster is assigned several long tasks, creating scheduling bottlenecks as described above.

6.3.2 Fine Granularity Tasks

Observe that the only reason for the above phenomenon is the existence of long tasks. A trivial solution is to eliminate long tasks by breaking them into smaller ones. One way to achieve that is to restrict the programmer to writing only small tasks. Alternatively, we investigate decomposing long tasks into a series of short ones. In the JPEG example, long task E is replaced by 3 shorter tasks E1, E2, E3, as in Figure 12. The results are presented in Figure 13, showing improved performance (shorter total run time compared to Figure 9). Additional improvements may be achievable by decomposing task E further and by also decomposing task C.

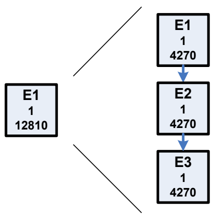

Figure 12 : Task E of the JPEG Benchmark decomposed into three equal parts.

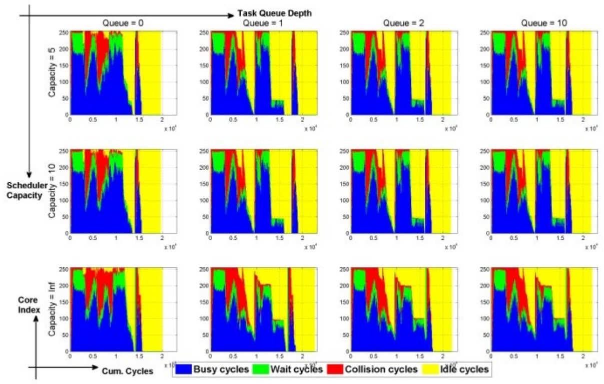

Figure 13 : Activity per cycle in the fine granularity JPEG benchmark, Latency = 20 cycles

6.4 *Linear Solver Benchmark*

This benchmark is highly parallel as seen by the great amount of instances of task G in the task map (Figure 3). In the simulation too (Figure 14) it is clear that most time all cores are busy, without much access to memory. One slot of task queue is sufficient to hide the latency and recover the performance of the system without latency. The scheduling capacity can remain low in this benchmark since the tasks are long enough to enable the scheduler to assign work to all cores.

6.5 *Benchmarks Analysis*

In this section, several comparisons of the different benchmarks are presented in order to illuminate the compromises a designer must make in a many-core architecture to support different programs and their unique attributes.

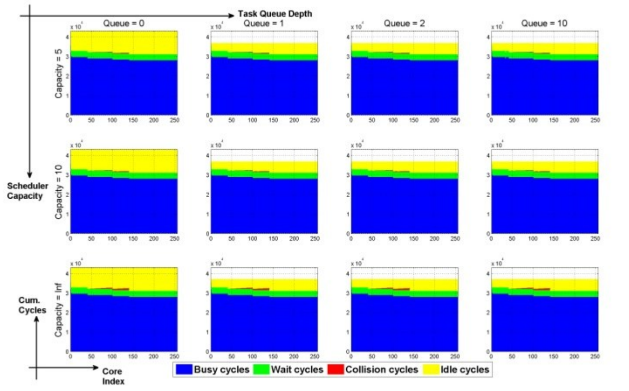

Figure 14 : Activity per core in Linear solver benchmark, Latency = 20 cycles

6.5.1 *Total Run Time*

Figure 15 shows the total run time of each benchmark for each configuration of queue depth and scheduler capacity. The red surface represents the no latency case, and the blue surface is for the 20 cycles latency scenario (latency from the Scheduler to the cores and vice versa). Notice variations along the Queue Depth axis: aside from the JPEG benchmark, where increasing the queues causes an increase in the total run time (explained by scheduling imbalance in Section 6.3), adding of as little as a two slots queue to each core can compensate for most of the latency between the scheduler and the cores. Where there is no latency, the queues obviously do not improve performance; fortunately, they do not degrade performance. Considering the Scheduler Capacity axis, observe that increasing scheduler capacity mostly helps performance. A capacity of 10 instances per cycle is sufficient to reach maximum utilization of the cores.

In summary, an architecture where each core has a 2 slot queue, and the scheduler has the capacity of scheduling 10 instances each cycle, suffices to utilize 256 cores in most cases even in the presence of long latency between the scheduler and the cores.

6.5.2 *Load Balancing*

Another important aspect of scheduling is balancing work among the different cores. Such balance can for example distribute the power and heat throughout the entire chip, thus enabling higher clock frequency and better overall performance.

Figure 16 presents standard deviation of the total busy times of all cores. As expected, removing the queues (0 Queue in the graphs) results in a lower STD, meaning the work is more balanced among the cores. Also, increasing scheduler capacity enables the scheduler to reach the high index cores, and distribute the work more evenly among the cores.

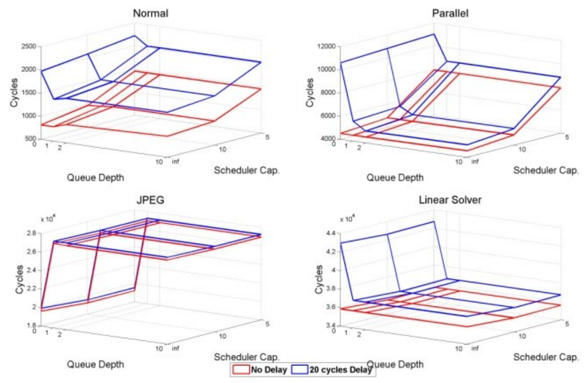

Figure 15 : Total run time vs. queue depth and scheduler capacity

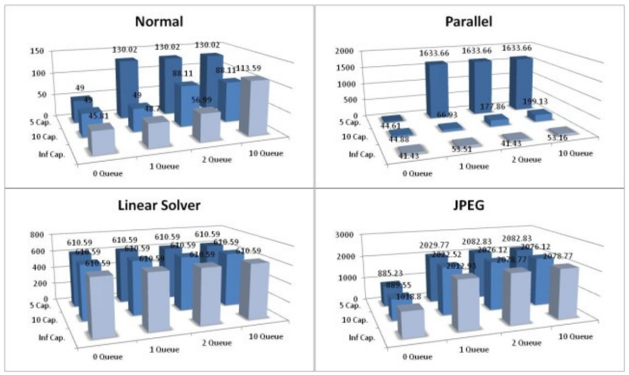

Figure 16 : STD of cores busy time

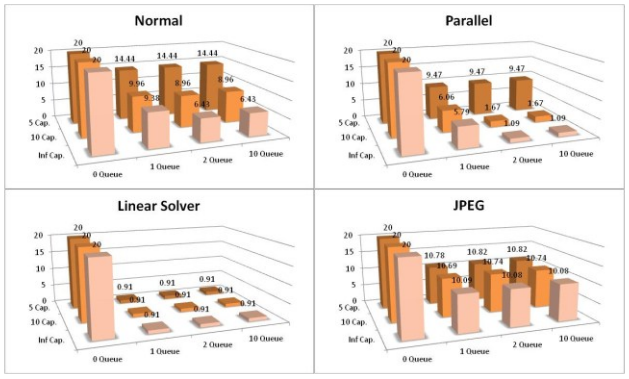

Figure 17 : Effective scheduling latency, Latency = 20 cycles

6.5.3 *Effective Allocation Latency*

In half the simulations, a 20 cycles latency was inserted between the scheduling of a task and the core receiving that task. Queues were added in order to compensate for this latency. Figure 17 shows how well the queues worked. For each instance that was scheduled by the scheduler, the time that was wasted until it arrived at the queue was counted. That is, if the core to which it was destined was idle, the entire 20 cycles latency was counted. If, on the other hand, the core was busy during the entire 20 cycles, then the latency was hidden and not counted. The effective latencies that each instance suffered were averaged over all the instances in the program, and are presented in the figure.

When there is no queue, each instance suffers the entire 20 cycles latency. Adding a one slot queue, however, is sufficient to hide much of the latency, even when scheduling capacity is low.

7 Conclusions and Future Work

In this paper we have analyzed how the performance of a many-core architecture depends on the hardware scheduler. The scheduler was simulated with various configurations. We conclude that an architecture where each core has a 2 slot queue, and the scheduler is capable of scheduling 10 instances each cycle is sufficient to efficiently utilize 256 cores in most cases even in the presence of long latency between the scheduler and the cores (Section 6.5.1).

We have extended the cycle accurate simulator of a many-core architecture developed in [*7*], by inserting the scheduling process. With this simulator, it is possible to analyze the behavior of different benchmarks, and explore new architectural modifications. Using the presented "Activity per cycle" and "Activity per core" graphs it is now easier to understand the different phenomena occurring in many-core architectures.

We have studied the effect of task assignment latency on system performance. It was shown that such latency can degrade performance, down to one half in the case of fine granularity. In order to hide this latency, we investigated adding task queues near each core. Such queues, even if very short, can hide most of the latency in many cases.

Adding a queue near each core may also reduce system performance as seen in Section 6.3. Several ideas of mitigating this issue were proposed, and some were implemented and analyzed. Two improvements seem promising: sharing queues among several cores and task map optimization and tuning.

Future research may address the following topics. Additional benchmarks may be analyzed in order to expose new phenomena. A blocking network may be investigated. Other scheduler distribution networks may be studied, such as tree and mesh. More solutions to the queues imbalance may be suggested and simulated. The implications of scheduling on power consumption may also be taken into account. Profiling for task map optimization and scheduling analysis may be examined to enable tuning.

References

[1] C.M. Wittenbrink, E. Kilgariff, and A. Prabhu, "Fermi GF100 GPU Architecture," *IEEE Micro*, vol. 31, no. 2, pp. 50-59, March-April 2011.

[2] [Online]. http://www.tilera.com

[3] [Online]. http://plurality.com

[4] X. Wen and U. Vishkin, "FPGA-based prototype of a PRAM-on-chip processor," in *Proceedings of the 5th conference on Computing frontiers*, 2008.

[5] L. Seiler et al., "LARRABEE: A Many-Core X86 Architecture for Visual Computing," *IEEE MICRO*, vol. 29, no. 1, pp. 10-21, Jan.-Feb. 2009.

[6] C. Silvano et al., "2PARMA: Parallel Paradigms and Run-time Management Techniques for Many-Core Architectures," in *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2010 , pp. 494 - 499.

[7] E. Friedman, D. Khoretz, and R. Ginosar, "HypercoreX: Non-Equidistant Memory Network in a Many-Core-Architecture," in *20th Euromicro Int. Conf. on Parallel, Distributed and Network-Based Computing (PDP), WIP session*, Feb. 2012.

[8] Z. Guz et al., "Many-Core vs. Many-Thread Machines: Stay Away From the Valley," *Computer Architecture Letters*, vol. 8, no. 1, pp. 25-28, JANUARY-JUNE 2009.

[9] W. Lee et al., "Space-Time Scheduling of Instruction-Level Parallelism on a Raw Machine," in *Proceedings of the 8th International Conference on Architectural Support for Programming Language and Operating Systems (ASPLOS-8)*, San Jose, Ca, 1998.

[10] W. Lee, D. Puppin, S. Swenson, and S. Amarasinghe, "Convergent Scheduling," in *Proceedings of the 35th Annual IEEE/ACM International Symposium on Microarchitecture*, 2002.

[11] J.-J. Shieh, Y.-C. Lee, and H.-R. Chen, "Fine grain scheduler for shared-memory multiprocessor systems," *IEE Proceedings - Computers and Digital Techniques*, vol. 142, no. 2, pp. 98 – 106, Mar 1995.

[12] T.P. Crummey, D.I. Jones, P.J. Fleming, and W.P. Mamane, "A Hardware Scheduler for Parallel Processing in Control Applications," in *CONTROL'94*, 1994, pp. 1098-1103.

[13] H. S. Kim and J. E. Smith, "An Instruction Set and Microarchitecture for Instruction Level Distributed Processing," in *Proceedings of the 29th Annual International Symposium on Computer Architecture*, 2002, pp. 71-81.

[14] L. Yu et al., "Study on Fine-grained Synchronization in Many-Core Architecture," in *10th ACIS International Conference on Software Engineering, Artificial Intelligences, Networking and Parallel/Distributed Computing (SNPD '09)*, 2009, pp. 524-529.

[15] Y. Etsion et al., "Task Superscalar : An Out-of-Order Task Pipeline," in *43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2010, pp. 89-100.

[16] P. Trancoso, P. Evripidou, K. Stavrou, and C. Kyriacou, "A Case for Chip Multiprocessors Based on the Data-Driven Multithreading Model," *International Journal of Parallel Programming*, vol. 34, no. 3, pp. 213-235, June 2006.

[17] N. Bayer and R. Ginosar, "High Flow-Rate Synchronizer/Scheduler Apparatus and Method for Multiprocessors," US Patent 5,202,987, April 13, 1993.

[18] M.B. Taylor et al., "The Raw microprocessor: a computational fabric for software circuits and general-purpose programs," *IEEE MICRO*, vol. 22, no. 2, pp. 25-35, Mar/Apr 2002.

[19] B. Beresini, S. Ricketts, and M.B. Taylor, "Unifying manycore and FPGA processing with the RUSH architecture," in *NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, 2011, pp. 22-28.

[20] X. Chen et al., "Multi-FPGA Implementation of a Network-on-Chip Based Many-core Architecture with Fast Barrier Synchronization Mechanism," in *NORCHIP*, 2010, pp. 1-4.

[21] Y.F. Hung, S.Y. Tseng, C.T. King, H.Y. Liu, and S.C. Huang, "Parallel Implementation and Performance Prediction of Object Detection in Videos on the Tilera Many-core Systems," in *10th International Symposium on Pervasive Systems, Algorithms, and Networks (ISPAN)*, 2009, pp. 563-567.

[22] T. C. Xu, L. Pasi, and H. Tenhunen, "Process Scheduling for Future Multicore Processors," in *INA-OCMC*, 2010.

[23] D. Waddington, C. Tian, and KC Sivaramakrishnan, "Scalable Lightweight Task Management for MIMD Processor," in *EuroSys workshop, Systems for Future Multicore Architectures (SFMA*

2011), Salzburg, 2011, pp. 1-6.

[24] D. Zydek and H. Selvaraj, "Processor Allocation Problem for NoC-based Chip Multiprocessors," in *Sixth International Conference on Information Technology: New Generations (ITNG '09)*, 2009, pp. 96-101.

[25] J. H. Kelm et al., "Rigel: an architecture and scalable programming interface for a 1000-core accelerator," in *Proceedings of the 36th annual international symposium on Computer architecture*, Austin, TX, USA, 2009.

[26] L. Chen, O. Villa, S. Krishnamoorthy, and G.R. Gao, "Dynamic Load Balancing on Single- and Multi-GPU Systems," in *IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, 2010, pp. 1-12.

[27] A.P.D. Binotto, C.E. Pereira, A. Kuijper, A. Stork, and D.W. Fellner, "An Effective Dynamic Scheduling Runtime and Tuning System for Heterogeneous Multi and Many-Core Desktop Platforms," in *IEEE 13th International Conference on High Performance Computing and Communications (HPCC)*, Sept. 2011, pp. 78-85.

[28] I. Wald, "Fast Construction of SAH BVHs on the Intel Many Integrated Core (MIC) Architecture," *IEEE Transactions on Visualization and Computer Graphics*, vol. 18, no. 1, pp. 47-57, January 2012.

[29] G. C. Caragea, F. Keceli, A. Tzannes, and U. Vishkin, "General-Purpose vs. GPU: Comparison of Many-Cores on Irregular Workloads," in *HotPar '10: Proceedings of the 2nd Workshop on Hot Topics in Parallelism*, 2010.

[30] R. Nagarajan, K. Sankaralingam, D. Burger, and S. W. Keckler, "A Design Space Evaluation of Grid Processor Architectures.," in *Proceedings of the 34th Annual International Symposium on Microarchitecture*, 2001, pp. 40-51.

[31] F. Song et al., "Evaluation Method of Synchronization for Shared-Memory On-Chip Many-Core Processor," in *IEEE International Symposium on Parallel and Distributed Processing with Applications*, 2009, pp. 571-576.

[32] S. Kumar, C. J. Hughes, and A. Nguyen, "Carbon: Architectural Support for Fine-Grained Parallelism on Chip Multiprocessors," in *Proceedings of IEEE/ACM International Symposium on Computer Architecture (ISCA)*, San Diego, California, June 2007.

[33] D. Sanchez, R. M. Yoo, and C. Kozyrakis, "Flexible Architectural Support for Fine-Grain Scheduling," in *Proceedings of the 15th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XV)*, March 2010.

[34] P. Evripidou, "Thread Synchronization Unit (TSU): A building block for High Performance Computers," in *Proceedings of the International Symposium on High Performance Computing, ISHPC*, 1997, pp. 107-118.

[35] N. Gupta, S.K. Mandal, J. Malave, A. Mandal, and R.N. Mahapatra, "A Hardware Scheduler for Real Time Multiprocessor System on Chip," in *23rd International Conference on VLSI Design (VLSID '10)*, 2010, pp. 264 - 269.

[36] M. Zhou, L. H. Shang, J. Zhang, and H. H. Jin, "Adaptive Hardware Real-Time Task Scheduler of Multi-Core ATPA Environment," in *NASA/ESA Conference on Adaptive Hardware and Systems, AHS 2009*, 2009, pp. 382 - 388.

[37] D.-S. Zhang, F.-Y. Chen, H.-H. Li, S.-Y. Jin, and D.-K. Guo, "An Energy-Efficient Scheduling Algorithm for Sporadic Real-Time Tasks in Multiprocessor Systems," in *IEEE 13th International Conference on High Performance Computing and Communications (HPCC)*, Sept. 2011, pp. 187-194.

[38] N. Bayer, "A Hardware-Synchronized/Scheduled Multiprocessor Model," Technion – Israel Institute of Technology, Thesis, English abstract online, http://webee.technion.ac.il/~ran/papers/NimrodBayerMScThesisAbstract1989.pdf January 1989.

[39] N. Bayer and R. Ginosar, "Tightly Coupled Multiprocessing: The Super Processor Architecture," in *Q. Jin et al (eds.) "Enabling Society with Information Technology"*. Tokyo: Springer, 2002, pp. 329-339, http://webee.technion.ac.il/~ran/papers/MP-Bayer-Ginos.

[40] N. Bayer and P. Aviely, "Shared Memory System for a Tightly-Coupled Multiprocessor," US patent 8,099,561 B2, January 17, 2012.

[41] O. Green and Y. Birk, "Scheduling Directives for Shared-Memory Many-Core Processor Systems," Electrical Engineering Dept., Technion, CCIT Report #803 January 2012.