VWQS: a Dispatching Mechanism of Variable-Size Tasks in Heterogeneous Systems

George Kornaros and Menelaos Pratikakis
Department of Informatics Engineering
Technological Educational Institute of Crete
Heraklion, Crete, Greece
email: kornaros@ie.teicrete.gr

Abstract—Platforms with accelerators improve both performance and energy efficiency because they are specialized for specific (or specific classes of) computations, which leads today in high-performance systems that increasingly turn into heterogeneous architectures, thus outperforming multicore processors running a software routine, heterogeneity in the hypervisor and thus provide fair sharing of cores or effective utilization of heterogeneous hardware [4]. These functional asymmetric heterogeneous architectures require careful system management.

As recent studies show [6], [23], [3], heterogeneous computing platforms comprising multi-cores and accelerators can offer higher throughput than a CPU-only or (custom hardware)-only platforms. Scheduling tasks that consume computational resources in terms of time of processing elements and of amount of memory is important in order to provide support for guaranteed performance to applications that concurrently offload tasks. Selecting of course the most efficient granularity depends on the application algorithm and on the hardware environment in which it is run [11]. On top, data-dependent and non-predictable applications though, require runtime and adaptive treatment to achieve application service differentiation or efficient compute resource utilization.

Tasks that can run as compute functionality on dedicated processing element(s) or in hardware are referred to as kernels. Even though application-specific management of particular accelerators can potentially unlock performance, the programming effort can be hidden away by an efficient programmable resource dispatcher. In this scope, the goal of this work is to introduce a dispatching algorithm that offers programmable weighted offloading of kernels of variable length in terms of kernel execution time; this contribution, to the best of our knowledge, is the first in the context of multiple-ISA accelerators system, with the possibility of efficient hardware implementation. However, the algorithm is generic to enable resource management for effectively utilizing the on-chip resources (of the CPU and of the accelerators), for instance through adaptively reprogramming the weights in the dispatcher.

When dealing with multiple heterogeneous devices, it is hard to exploit inter-task parallelism on multiple such computing devices. We also do not consider the impact of scheduling uncertainties so as to choose among various schedules that guarantee robustness. We envision an independent hardware dispatcher or a dispatcher in an independent core coupled to the general-purpose processor, which is hardly affected by various run-time workload changes. Additionally, fixed-function accelerators in today’s SoCs are usually loosely-coupled with the rest of the system, incurring inefficient data movement and high software engineering cost while at the same time accelerator designs use scratchpad memory, i.e., software-managed on-chip SRAM with a dedicated address space, to store the accelerators local data. Thus, in this work we do not consider bringing cache hierarchies to accelerators.

I. INTRODUCTION

As hardware accelerators perform certain tasks more efficiently than processors running a software routine, heterogeneous architectures increasingly become the dominant architecture for high-performance systems that increasingly turn into heterogeneous architectures, thus outperforming multicore processors. GPUs are leveraged in various domains of general-purpose GPU (GPGPU) processing to facilitate data-parallel compute-intensive applications. New features, such as dynamic parallelism in Kepler GPUs [17] allow less-structured, more complex tasks to run easily and effectively, enabling programmers to expose more varied kinds of parallel work.

In servers, hypervisors or virtual machine managers (VMM) implement hardware virtualization techniques to manage a diverse pool of processing resources, and prioritize processing tasks and ensure fairness in allocating available processing accelerations resources. Recent works focus on enforcing heterogeneity awareness in the hypervisor and thus provide fair sharing of cores or effective utilization of heterogeneous hardware [4]. These functional asymmetric heterogeneous architectures require careful system management.

Furthermore, hardware accelerators are becoming increasingly popular and make their way into embedded (TIOMAP[18]) and integrated designs (e.g., AMD Fusion [1]). As recent studies show [6], [23], [3], heterogeneous computing
causing runtime irregularities.

The remainder of this document is concerned with presenting a scheduling algorithm to dispatch fine-grain computational tasks of varying length in heterogeneous systems. We focus on improving dispatching in SoCs with specialized programmable accelerators (i.e., with the capacity to execute kernels), or with hardware IPs, rather than with giving rigorous description of generic OS-specific scheduling.

Section II discusses related work. Section III presents an overview of the heterogeneous system architecture that integrates customized accelerator units. The kernel dispatching algorithm, is introduced in Section IV. Section V presents the evaluation of the proposed dispatcher, and Section VI summarizes and concludes this paper.

II. RELATED WORK

Computational accelerators are rapidly emerging for power-efficient computing through increased specialization. Managing hardware devices for which per-request engagement is too expensive as Aron and Druschel argued that per-packet interrupts are too expensive for fast networks and that batched processing through rate-based clocking and network polling is necessary for high efficiency [2]. Exceptionless system calls [20] are another mechanism to avoid the overhead of frequent kernel traps by batching requests. Kornaros has proposed a high-speed hardware scheduler to provide bounded jitter service, but a drawback is that variability in task execution time is not considered [15]. Hussain et al. [9] proposed a pattern-based memory controller for application specific accelerators in order to establish a fast communication with the host which is appropriate only for compile-time generated access patterns. In industry, architectures, such as Cayman [12], already include hardware dispatch processors, which are responsible for managing the currently executing kernels and scheduling wavefronts onto the general-purpose cores and fixed-function hardware. However, the dispatcher decisions are unaware of the type and complexity of the tasks.

Additionally, researchers have proposed integer-linear-programming techniques to efficiently schedule kernels to hardware accelerators in terms of energy and quality-of-service constraints[16][14]. On the basis of an execution model, the run-time sends the kernels across the accelerators in order to minimize energy. In another view, methods are developed to treat applications to heterogeneous computing platforms as a tuning and mapping problem [19]. Towards optimal utilization a software-level approach called kernel slicing [25] (partitioning into a number of blocks) attempts to enable true concurrency among kernels that have non-deterministic performance features; this scheme utilizes power-performance estimation modeling. Other past works that have focused on dynamic scheduling schemes for heterogeneous computing systems typically use the information such as device utilization, input data size and performance prediction to select the device between the CPU and the GPU dynamically, and they can be implemented through the OS [5], [10]. On top we propose an efficient dispatcher algorithm supporting weights to dynamically schedule variable-length tasks on custom accelerator computing engines. Moreover, the proposed algorithm manages irregularities, that is, misbehaving offloads (in terms of submitting out-of-profile kernels) with utterly no impact on the weights of other queues.

NVIDIA devices virtualize an unlimited number of compute units (CUs) on physical streaming multiprocessors (SMs) by quickly switching context of a workgroup to another using hardware scheduler, without allowing control or prioritization to the programmer. On the other hand, Heterogeneous System Architecture Foundation [7][8] offered Unified Virtual Address to provide an abstract view of unified memory system in separate physical memory in order to remove the burden of managing multiple memory spaces [13], but still leaves work distribution and scheduling as programmer’s responsibility. Using our approach the dispatcher can regulate variable-size tasks and dynamically be adjusted to the system’s constraints or to the programmer’s service requests.

III. SCHEDULING ON CUSTOM ACCELERATOR-ENABLED HETEROGENEOUS SYSTEMS

We assume a multi-accelerator system which uses packet-based offloading style. We used a simple task profilling method that provides sufficient information to determine the computational characteristics of kernels. The processing requirements are considered to be data-dependent and thus we consider the computational characteristics of each kernel task to change depending on workload data (which cannot be predicted). Therefore, we use a runtime profiling approach, where profiling information is collected while the system is operational.

Tasks characterization can be performed offline or online. Even though workload characterization is not included in the proposed scheduler approach, we investigate this process using an experimental platform. Ideally, to avoid inaccuracies of software instrumentation, online task characterization is done by reading values from hardware performance counters, by starting and stopping counters at task boundaries and hints are generated based on the values collected to characterize the weight of each task. Real runtimes actually capture application- and input-dependent behavior.

Modern hybrid FPGA SoCs (e.g. Xilinx’s ZYNQ) allow embedding soft-core processors such as MicroBlaze to enable the easy development of customized systems. Figure 1 shows a prototype system that we developed to emulate an accelerator-enhanced system which can execute different kernels. Each one kernel is executed on a dedicated MicroBlaze processor; the MicroBlaze communicates with the processing system through a shared register-based memory to get the next job and deliver the status of processing, while the data for each kernel are maintained in system memory. The user application is executing in the ARM Cortex dual A9 processing system and accesses the interface memory (which is exposed using a user-level uio driver) through mmap() Linux system call. The application delivers the corresponding pointers to the data inside 64-byte packets, which the application sends to the appropriate co-processor accelerator.
We ported a number of different kernels on the platform to run on the dedicated MicroBlazes as bare-metal applications. The MicroBlaze soft-core is configured with 2KB data cache. Table I summarizes the kernels execution times for different parameters for each one. In the image denoise example the kernel replaces each pixel by the median value of itself and its neighbors in a grayscale image. SHA-1 computes a 160 bit hash value for different message size. Kernels CRC-32, SHA-1 and BlackScholes exhibit a linear behavior as a function of data size, the image processing denoise shows an exponential behavior, while FFT is dependent on number of points.

The example kernels highlight the weight of each one in terms of computation intensity when a dedicated computation unit is engaged. Hence, on the basis of the type of computation that an application asks for assistance from an accelerator, the application (or the driver) can assign also the weight to a particular offload instance. By using a lookup table similar to Table I, the weight of a kernel can determine quite accurately its computation intensity, and consequently the compute resources that are needed and the corresponding energy for the particular offloading. Other scheduling proposals [24] also use the real execution time as the task weight, or estimated execution time with errors, such as randomly selected between: \( \{1-r/100\}, \{1+r/100\}\), where \( r \) is the absolute percentage error, and \( t' \) is the estimated time.

Notice that we consider two types of memory hierarchies: local scratchpads, in which scratchpads use DMA to transfer data, and GPU-like memories with ample bandwidth, so that captured measurements suffer of minimum inaccuracies due to memory bandwidth constraints.

The proposed dispatching algorithm can either be instantiated as a kernel module or even as a dedicated module to handle the dispatcher queues that store the offloaded jobs. In this way, CPU interference to the dispatcher performance can be negligible.

### IV. Dispatcher Algorithm

Our scheduler framework assumes \( N_q \) queues with a fixed weight that the system assigns to each queue. The weight of each queue represents the maximum time quantum allocated to this queue for occupying the accelerator. If higher priority applications than anticipated are executing, that request more accelerator resources or more time quanta, then, queue weights need to be reconfigured. Queue weights and even number of queues are programmable parameters delivering runtime adaptivity to the system. Each application enqueues tasks in queues. Actually, one job is an accelerator task that processes an amount of data and produces result data. One job is enqueued by using a fixed-size packet kernel that processes an amount of data and produces result data. The kernel replaces each pixel by the median value of itself and its neighbors in a grayscale image. SHA-1 computes a 160 bit hash value for different message size. Kernels CRC-32, SHA-1 and BlackScholes exhibit a linear behavior as a function of data size, the image processing denoise shows an exponential behavior, while FFT is dependent on number of points.

The parameters of each queue involve the weight for each queue \( \text{Weight}_\text{Queue}_i \), than can be configured before activating the dispatcher and one counter, the \( \text{CurrentTotalWeight}_i \). The parameters of the dispatcher for all queues include the \( \text{TotalWeight} \) which is the sum of all \( \text{Weight}_\text{Queue}_i \) and the corresponding counter \( \text{CurrentTotalWeight} \).

The pseudo-code shown in Algorithm 1 depicts the baseline VWQS dispatch algorithm. The conditions that are examined refer to the head packet of each queue.

Notice that the update operation of the counters in the case of identifying an eligible packet differs from the case that a packet is ready but violates the queue weight, or when the queue is empty. In the latter cases the counters need to be adjusted in order not to examine these queues in next iterations within the same service cycle (order). The typical update operations involve:

\[
\text{CurrentTotalWeight} = \text{Weight}_\text{Queue}_i + \text{packet}_\text{head}.\text{Weight},
\]

\[
\text{CurrentTotalWeight} = \text{CurrentTotalWeight} + \text{packet}_\text{head}.\text{Weight}.
\]

Figure 2 shows how the VWQS dispatches jobs using three queues. Given the configured weights as shown in Figure 2, the dispatcher can actually schedule jobs proportionally to these weights, if jobs are available in the queues and assuming that the accelerator service capacity has no impact on the dispatch mechanism.

The queue counter \( \text{CurrentTotalWeight}_i \) maintains the sum of jobs that are dispatched during one service cycle; when it is shaded in the figure, then the weight of this queue is served in
TABLE I: Kernels workload execution on a MicroBlaze CPU at 100Mhz; FFT, BlackScholes asset pricing, CRC32, image denoise, and SHA-1 runs for scaling parameters

<table>
<thead>
<tr>
<th>Nx</th>
<th>FFT Time</th>
<th>B'Scholes</th>
<th>Time</th>
<th>CRC</th>
<th>Time</th>
<th>Denoise</th>
<th>Time</th>
<th>SHA-1</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(cc)</td>
<td>(N Days)</td>
<td>(cc)</td>
<td>(Bytes)</td>
<td>(cc)</td>
<td>(Row*Col)</td>
<td>(cc)</td>
<td>(bytes)</td>
<td>(cc)</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>31186</td>
<td>10</td>
<td>1078228</td>
<td>60</td>
<td>3357</td>
<td>5x5</td>
<td>2091</td>
<td>225</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>204701</td>
<td>20</td>
<td>2114111</td>
<td>160</td>
<td>7521</td>
<td>10x10</td>
<td>73279</td>
<td>425</td>
</tr>
<tr>
<td>12</td>
<td>16</td>
<td>457306</td>
<td>30</td>
<td>3277024</td>
<td>310</td>
<td>13842</td>
<td>20x20</td>
<td>276893</td>
<td>625</td>
</tr>
<tr>
<td>17</td>
<td>32</td>
<td>1008303</td>
<td>40</td>
<td>4296202</td>
<td>410</td>
<td>18265</td>
<td>30x30</td>
<td>602092</td>
<td>825</td>
</tr>
<tr>
<td>27</td>
<td>32</td>
<td>1019225</td>
<td>50</td>
<td>5480005</td>
<td>560</td>
<td>24321</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>32</td>
<td>1024358</td>
<td>60</td>
<td>6515138</td>
<td>610</td>
<td>26354</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>64</td>
<td>2265023</td>
<td>70</td>
<td>7553654</td>
<td>660</td>
<td>28257</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Algorithm 1 Variable-Weight Queue Scheduling algorithm

1: function VWQS($N_Q$)
2:     i ← NextStartQ
3:     if queues are empty then
4:         return -2
5:     end if
6:     while i< $N_Q$ do
7:         if (empty Q) then
8:             return -2
9:         end if
10:        if (packet not found) then
11:            return -1
12:        end if
13:        NextStartQ ← i + 1
14:        CurrentTotalWeight = CurrentTotalWeight + CurrentTotalWeight
15:        CurrentTotalWeight = CurrentTotalWeight + CurrentTotalWeight
16:        return i
17:        end while
18: end function

the current service cycle and will not be visited again until the following service cycle. Thus, the remaining queues are visited in round-robin fashion till the CurrentTotalWeight reaches the TotalWeight.

Notice that after the service cycle (i.e., order 0) is completed, the new service cycle is invoked starting with queue number two, since the last serviced queue was number one (as shown at the bottom instance).

A. Weighted-VWQS Variable-Weight Queue Scheduling Algorithm

In this section we present a modified version of the VWQS algorithm that favors the potential appearance of many fine-grain jobs in a single queue while respecting the weighted difference among the neighbor queues. As the size of many real-life problems (e.g. medical or security applications) does not provide sufficient parallelism for the underlying hardware accelerators, it is desirable to reduce queuing latency. The principle idea is to dispatch all jobs from one queue as long as the sum of their weights does not exceed the difference between the weight of the current queue with the weight of the next queue to serve.

If one application enqueues many fine-grain kernels then the baseline VWQS would dispatch only one such kernel, and then the algorithm would visit all the other queues before considering again this queue. With the weighted-VWQS version though, the dispatcher will serve as many kernels from a single queue continuously until the sum of the dispatched kernels reaches the weight difference with the next queue. The VWQS baseline algorithm is modified by conditioning the increment of the pointer to the next queue (please refer to line 15 in Algorithm 1). A local variable maintains the total weight of the dispatched jobs if the conditions at lines 8-11 are met. This local sum is constrained by the weighted difference with the next queue.

V. Evaluation Measurements

In this section we present different simulation results to demonstrate the VWQS algorithm behavior. We assume one accelerator throughout the experiments. Similarly, on Fermi GPUs for instance, one kernel can take the entire GPU if it has sufficient thread blocks to occupy all the multi-processors. Concurrent execution of two such kernels almost degrades to sequential execution of individual kernels.
Given eight queues we assigned the weights of the queues as follows: Q0:1000, Q1:900, Q2:800, Q3:700, Q4:600, Q5:500, Q6:400 and Q7:300. For instance, Q0 can serve packets with total weight at most 1000. The maximum total weight that the dispatcher can service in a cycle sums to the weights of all queues, which is 5200. In all scenarios an additional delay of 2 time quanta is added per dequeue operation to account for transferring each job to the accelerator.

Figure 3 shows the jobs latency when dispatching from the eight queues at runtime; the horizontal axis represents the service cycles. The number of served jobs is proportional to the weight of each queue. In this time period (during each service cycle), a number of jobs are inserted to enque many packets as can be served in a service cycle, on the basis of their total weight. For example in Q0, packets with maximum total weight equal to 1000 are enqueued, in Q1 packets with maximum total weight equal to 900 are enqueued, and so on. So Q0 has weight 1000 and can accept maximum 10 jobs of 100 each. The packets are enqueued just before the new cycle starts. This means that all queues are empty when the new packets are inserted.

Table II depicts the impact of changing the configuration and assigning small jobs from queues with lower weight to queues with larger weight and assigning large jobs from queues with larger weight to queues with lower weight. Even though it is expected that performance change is proportional to the difference of the weights of the queues, it is noticeable that the impact to latency is smaller when we migrate small jobs; the bigger difference is less than 10%. However, when big jobs (time-consuming) shift from queues of large weight to queues with small weight, then the impact to latency exceeds 22%.

Consequently, one obvious benefit is that for jobs reaching a critical point in terms of performance and service latency we can dynamically shift them to higher priority queues. Using recent software monitoring methods [21] for running applications we can detect such deviations from pre-set goals in terms of performance and react accordingly. Alternatively, from a different perspective, we can temporarily regulate the weight of the queue, instead of moving a task, which can entail larger complexity.

A. Weighted-VWQS Results

In this section we study the impact of the weighted-VWQS dispatcher with respect to the baseline algorithm. In each service cycle the scheduling algorithm dispatches kernels proportional to the weight of each queue. In this time period (during each service cycle), a number of jobs are inserted to
(a) Large jobs in queues with large weights, small jobs in queues with small weight.

(b) Small jobs in queues with large weights, large jobs in queues with small weight.

Fig. 4: Latency of job queues for VWQS dispatcher serving eight queues with weights 1000, 950, 900, 850, 800, 750, 700, 650 and jobs with average weights of 100 for queues q0..q3 and weights 50 for queues q4..q7 (scenario a), and jobs with average weights of 50 for queues q0..q3 and weights 100 for queues q4..q7 (scenario b).

TABLE II: Impact of mixing large and small jobs

<table>
<thead>
<tr>
<th>Migration - Small Jobs</th>
<th>Performance increase (%)</th>
<th>Latency (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q4 → Q0</td>
<td>26.0</td>
<td>2.2</td>
</tr>
<tr>
<td>Q5 → Q1</td>
<td>27.9</td>
<td>5.8</td>
</tr>
<tr>
<td>Q6 → Q2</td>
<td>28.9</td>
<td>7.2</td>
</tr>
<tr>
<td>Q7 → Q3</td>
<td>31.3</td>
<td>9.5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Migration - Large Jobs</th>
<th>Performance increase (%)</th>
<th>Latency (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q0 → Q4</td>
<td>-31.3</td>
<td>-22.3</td>
</tr>
<tr>
<td>Q1 → Q5</td>
<td>-22.0</td>
<td>-22.3</td>
</tr>
<tr>
<td>Q2 → Q6</td>
<td>-23.1</td>
<td>-22.4</td>
</tr>
<tr>
<td>Q3 → Q7</td>
<td>-25.0</td>
<td>-23.5</td>
</tr>
</tbody>
</table>

each queue that is equal to the weight of each queue with an average weight of 20. Notice that the weight difference of two queues is set to 100. The baseline algorithm serves a number of jobs that decreases near linearly with the weight of each queue. In both scenarios we insert in total ten thousand packets with the input process to generate a number of packets for each queue (using a Poisson generator) so as to insert a workload comprised by the maximum number of packets which have a total weight that is less or equal to the queue weight. We maintain the input traffic stable for a range of almost ten thousand offloads.

As the rightmost three columns of table III summarize, this modification achieves an improvement to the average latency of the queues 0 to 3 up to 3.89%, while the peak latency is almost the same. However, the lower priority queues are subject to larger latency compared to the baseline algorithm. The impact to the latency is proportional to the size of the fine-grained jobs that are offloaded in relation to the difference of the weight between two queues.

B. Sensitivity Analysis

We introduced irregularities in the behavior of queue four in random, in order to study the interference to the dispatching process of the neighbor queues. In addition, the high number of requests generated by more accelerator threads may cause congestion in the memory subsystem, degrading overall accelerator system performance; however these effects are not considered next. The queues are configured with weights 1000 down to 300 (queue-0 down to queue-7) and a Poisson generator inserts fine-grain, small, kernels of average weight equal to 50; the leftmost column in plot (a) depicts a linear relation of the mean weight that is served by each queue. In the next scenarios (middle and rightmost columns) the generator of queue number four inserts coarse-grain jobs of an average weight of 150 and 400. Plot 5 shows for all three scenarios the total weight served by each queue, which is normalized to the weight of queue seven, while plot (b) on the right shows the latency in the third scenario. While in the second case, queue four accepts jobs of $3 \times$ weight, the mean served weight of queue four shows 8% deviation; when we feed jobs of $8 \times$ weight to queue four, the mean served weight of queue four shows 25% deviation.

We further optimized the Weighted-VWQS dispatching algorithm to consider corner cases (i.e., out of profile) since the offloading of coarse-grain, apriori unknown, or non-deterministic jobs can potentially become the source of irregularities. These kernels with weight larger than the difference of the weights between adjacent queues are considered, by enabling the Weighted-VWQS dispatcher to penalize such a queue when such a large job is scheduled; the next job from the same queue is artificially increased with the difference of the extra weight of the previously served job. The penalized scoring is applied also for kernels that are submitted to queues with smaller weight. Essentially, an application can now offload a job of any size to a queue; the dispatcher guarantees to schedule even this job and at the same time respect the queue weight. This is
TABLE III: Baseline-VWQS(B) and Weighted-VWQS(W) dispatching of jobs with average size (weight) of 20

<table>
<thead>
<tr>
<th>Q#</th>
<th>Queue weight</th>
<th>Lat_B (avg)</th>
<th>Lat_B (peak)</th>
<th>Jobs_B served</th>
<th>Lat_W (avg)</th>
<th>Jobs_W served</th>
<th>Lat_B - W (avg%)</th>
<th>Jobs_B - W (peak%)</th>
<th>Jobs_B - W (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1000</td>
<td>3581.87</td>
<td>5656</td>
<td>1993</td>
<td>3566.33</td>
<td>1983</td>
<td>-0.43</td>
<td>0.088</td>
<td>2.533</td>
</tr>
<tr>
<td>1</td>
<td>900</td>
<td>3277.01</td>
<td>5576</td>
<td>1751</td>
<td>3149.32</td>
<td>1773</td>
<td>-3.89</td>
<td>0.538</td>
<td>1.256</td>
</tr>
<tr>
<td>2</td>
<td>800</td>
<td>3041.78</td>
<td>5438</td>
<td>1565</td>
<td>2978.54</td>
<td>1591</td>
<td>-2.08</td>
<td>2.648</td>
<td>1.663</td>
</tr>
<tr>
<td>3</td>
<td>700</td>
<td>2750.71</td>
<td>5091</td>
<td>1359</td>
<td>2758.50</td>
<td>1369</td>
<td>-0.329</td>
<td>7.837</td>
<td>0.735</td>
</tr>
<tr>
<td>4</td>
<td>600</td>
<td>2429.02</td>
<td>4707</td>
<td>1161</td>
<td>2490.88</td>
<td>1181</td>
<td>2.54</td>
<td>8.030</td>
<td>0.611</td>
</tr>
<tr>
<td>5</td>
<td>500</td>
<td>2101.82</td>
<td>3540</td>
<td>773</td>
<td>1900.42</td>
<td>784</td>
<td>10.98</td>
<td>5.113</td>
<td>1.423</td>
</tr>
<tr>
<td>6</td>
<td>400</td>
<td>1712.41</td>
<td>3540</td>
<td>773</td>
<td>1900.42</td>
<td>784</td>
<td>10.98</td>
<td>5.113</td>
<td>1.423</td>
</tr>
<tr>
<td>7</td>
<td>300</td>
<td>1341.07</td>
<td>2807</td>
<td>589</td>
<td>1568.49</td>
<td>582</td>
<td>16.96</td>
<td>29.426</td>
<td>-1.118</td>
</tr>
</tbody>
</table>

(a) Normalized service to queue seven, when queue number four exhibits irregular behavior; the average weight of all queues is 50, while queue four serves jobs of average size 50, 150 and 400

(b) Latency when offloading 10K jobs; queue four serves over-sized jobs (exceeding the weight of queue four)

Fig. 6: Operation analysis perspective of the baseline algorithm using Xilinx’s Vivado high-level synthesis tools
done by delaying the next jobs by the difference of the weight of the large job. Hence, as plot in figure 5 shows, the large jobs in queue 4 cause 8% and 25% reduced service in the second and in the third scenario. This effect has no implication to the dispatching of the jobs that reside in the other queues; in all cases the interference remains quite less than 1%. Figure (b) depicts the average latency of all queues; jobs in queue four suffer the penalty that the dispatcher applies after serving large jobs.

C. Hardware Realization

We used Xilinx’s Vivado high-level synthesis [22] to realize the baseline algorithm for an 8-queue hardware dispatcher mapped in a ZYNQ-7000 device. Figure 6 shows the sequence of operations to deliver a scheduling decision with a latency of 19 clock cycles. The design is constrained to operate with a 4.71 ns clock cycle while it occupies 920 LUTs and 1246 FFs. All data structures are realized using dual-ported memories to enable scalability to larger number of queues. Therefore, the only limitation is the storage capacity of the device. For small number of queues we can consider collapsing memory structures and use registers, and thus the latency can be reduced significantly.

VI. Conclusions

The effectiveness in terms of performance of multi-ISA heterogeneous multi-cores and of systems with hardware accelerators is directly dependent on how well a dispatching algorithm can schedule jobs on the compute engines. Incorrect scheduling decisions can unnecessarily degrade performance, waste energy/power, and fail to guarantee real-time or mixed-criticality requirements. In this scope, we proposed
a dispatching algorithm that controls variable-size jobs in a weighted manner and provides isolation of service. VWQS uses collected run times to guide dispatching decisions. We evaluated the behavior of VWQS using scaling actual runtimes on an emulation FPGA platform. We conclude that our dispatcher provides an efficient and flexible infrastructure, realizable also as a hardware component, to accurately control and adapt CPU-accelerator performance trade-offs in heterogeneous SoCs. Actually, the schemes proposed in this paper create a foundation for high performance variable-size computation scheduling.

In the future we intend to explore data, task and pipeline parallelism using the VWQS dispatcher, its impact on very fine-grain tasks, and to integrate the hardware version of the dispatcher with the multiple-accelerator platform. The goal is to realize the dispatcher to a real platform by embedding several hardware accelerators, even with different architectures.

ACKNOWLEDGMENT

This work was partially supported by the European Union (EU) FP7 project SAVE under contract FP7-ICT-2013-10 No 610996. We would also like to thank the anonymous reviewers for their helpful comments and suggestions.

REFERENCES


