A short-transfer model for tightly-coupled CPU-FPGA platforms

Alexander Kroh
University of New South Wales
Sydney, Australia
alex.kroh@unsw.edu.au

Oliver Diessel
University of New South Wales
Sydney, Australia
o.diessel@unsw.edu.au

***Abstract*—Due to the cost of repeated data movement between CPU and FPGA, the use of FPGA-based accelerators has traditionally been limited to offloading long-running tasks from the CPU to programmable logic. Although modern heterogeneous platforms, such as Zynq and HARP, reduce the costs of CPU-FPGA data transfers, the traditional offload model is cemented as the popular choice.**

For these systems to become truly heterogeneous, the utilisation of all computational resources should be optimised. In particular, the CPU and FPGA should cooperate by dividing the workload between them so as to maximize system throughput.

We first derive a model that predicts the optimum partitioning of a workload between hardware and software. We then measure the performance of short transfers between CPU and FPGA on the Zynq CPU-FPGA platform. Such transfers are essential to efficiently synchronise between cooperating hardware and software tasks. Finally, we demonstrate how our derived model can be used to choose the optimum workload partitioning to within 8% of the optimum for an accumulator task and predict its execution time within 12%.

I. Introduction

As the compute power of traditional CPU technology has failed to keep up with today's processing demands, we look favourably towards FPGA technology as a valuable addition to compute platforms. The large latency of off-chip communication has traditionally limited the application domain of these FPGA-based accelerators to long-running tasks. For a long-running task, this latency is masked by a relatively long execution time and high throughput. For short-running tasks, the communication latency quickly masks any performance benefit that hardware acceleration may provide.

Tightly-coupled high performance CPU-FPGA systems have emerged in which a high-performance processor and programmable logic are located on the same device. Not only does this close proximity reduce the communication latency between them, it also allows hardware and software to access shared memory via the on-chip last level cache (LLC). Such low-latency memory access can extend the range of applications that are suitable for hardware acceleration from long-running tasks to short-running tasks.

In this work we investigate the partitioning of small workloads between hardware and software for cooperative processing. We imagine a system comprised of a sea of small accelerators that provide computational support for a range of general-purpose applications.

Our contributions are as follows:

- We derive a model that determines the optimal partitioning of a workload between CPU and FPGA for any given workload size. In this model we consider the overhead of communication in terms of CPU processing time and transfer latency.
- We predict for which workload sizes the execution time of a task will be reduced if some portion of the workload is processed in hardware.
- We measure the communication overheads of short transfers between the tightly-coupled CPU and FPGA that is provided by the Zynq SoC. These measurements provide key parameters for our partitioning model.
- We demonstrate that our model can be used to predict optimum workload partitioning between CPU and FPGA. We do that by studying a stream-based integer accumulation task.

II. Related work

Much research has been carried out on algorithms that partition an application between hardware and software. Surveys of this work can be found in [1] and [2]. However, those works do not consider cooperative processing. Rather, they assume the software is idle when control is passed to the hardware accelerator until control returns to the processor. This design pattern is also present in hardware-software communication frameworks such as RIFFA [3] and the communication templates instantiated by the high-level synthesis (HLS) tool LegUp [4], unless the application is multithreaded.

Although some hardware-software partitioning algorithms consider the cost of moving data between the CPU and programmable logic, the cost of control and synchronisation is rarely considered.

III. Communication model

The focus of our work is cooperative computation, in which two or more compute elements (CEs) perform a common function on a subset of the provided data (Fig. 1). Once all CEs have processed their workload partition, a nominated CE aggregates the partial results and returns the final result to the application.

The execution time T of a cooperative task is determined by the longest completion time across the FPGA (T_F) and CPU

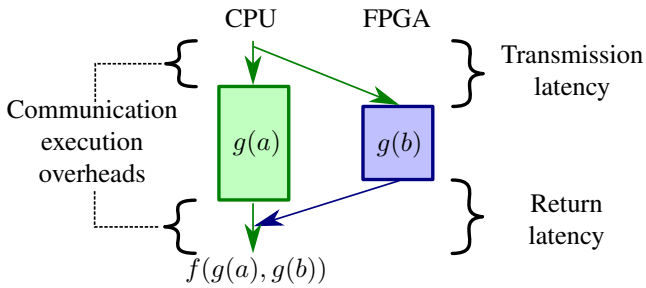

Fig. 1: Cooperative system architecture and overheads.

(T_C). If too much work is given to the FPGA, the CPU will become idle as it waits for the FPGA to complete. If too little work is sent to the FPGA, an opportunity for parallel execution is lost. Therefore, the workload must be carefully partitioned to minimise the completion time $T = max(T_F, T_C)$, which occurs when $T_F = T_C$.

When a workload N is large, the cost of the communication needed to initialise the accelerator can be ignored as it is small relative to the computation time. The completion time of the FPGA and CPU partitions can then be calculated using (1a) and (1b) respectively, where α^* is the fraction of the workload that should be processed by the FPGA. By equating (1a) and (1b), we see that α^* partitions the workload proportionally to the throughput provided by the FPGA (X_F) and the CPU (X_C) (2).

$$T_F = \frac{\alpha^* N}{X_F} \quad (1a) \qquad T_C = \frac{(1-\alpha^*)\,N}{X_C} \quad (1b)$$

$$\alpha^* = \frac{X_F}{X_F + X_C} \quad (2)$$

For small workloads, cooperative computation requires careful attention to data transfer costs in terms of both transfer latency (D_F) and CPU overhead (O_C) (Fig. 1). Transfer latencies on both the transmission and return paths reduce the amount of work the FPGA can complete by $T_F - T_C$. Programmable logic must wait for the processing command to arrive and ensure that the result is available to the CPU as soon as it is needed. The transfer of the processing command is performed by the CPU. The CPU execution cycles required to perform this transfer represent an opportunity cost to the CPU as these cycles could be used to process the workload. By considering both CPU overhead and transfer latency, the workload can be partitioned by α (4) such that the CPU and FPGA complete the processing of their respective parts at the same time.

$$T_F = \frac{\alpha N}{X_F} + D_F \quad (3a) \qquad T_C = \frac{(1-\alpha)\,N}{X_C} + O_C \quad (3b)$$

$$\alpha = \alpha^* \left[X_C (O_C - D_F) \frac{1}{N} + 1 \right] \quad (4)$$

Workload partitioning improves task completion time only if the time required for communication is masked by a reduced completion time. To determine the workload size N_L for which we benefit from using the programmable logic, we must ensure two conditions are met: First, the time to send some subset αN_L of the workload to the FPGA for processing, to compute $(1-\alpha)N_L$ work on the CPU and to receive the results is less than or equal to the time required to process N_L work on the CPU alone (5). Second, the time to process αN_L on the FPGA should also be less than or equal to the time to process N_L work on the CPU alone (6). We find N_L by solving these two equations simultaneously, where O_0 is the proportion of O_C associated with calling and processing the function in software with $N = 0$ work.

$$\frac{N_L}{X_C} + O_0 \geq \frac{(1-\alpha)\,N_L}{X_C} + O_C \quad (5)$$

$$\frac{N_L}{X_C} + O_0 \geq \frac{\alpha N_L}{X_F} + D_F \quad (6)$$

$$N_L \geq X_C \left[\frac{X_C\,(O_C - O_0)}{X_F} + D_F - O_0 \right] \quad (7)$$

IV. Evaluation

We used the Avnet Zedboard for our study. The Zedboard features a Xilinx XC7Z020 Zynq system on chip (SoC) that provides two ARM Cortex-A9 CPUs and programmable logic within the one package. An accelerator coherency port (ACP) provides a 64-bit data bus between programmable logic and the cache for shared memory access. Direct communication between the CPUs and the programmable logic is provided by a 32-bit wide general-purpose (GP) port.

A. CPU communication overhead evaluation

In order to use our model, we first found values for the CPU overhead and latency of CPU-FPGA communication. These communication overheads also assisted in choosing the most appropriate communication primitive for our accelerators.

In our experiments, direct communication to programmable logic was issued to an AXI memory controller that we connected directly to the GP port at the CPU-FPGA boundary. This controller was designed to respond immediately to all requests and thereby eliminated latency due to soft interconnects and peripherals.

An ACP read transaction to shared memory from programmable logic can be served from any level of the memory hierarchy, including the private L1 cache of either CPU. On the other hand, an ACP write transaction from programmable logic is always issued to the L2 cache. In this case any corresponding cache lines present in the private L1 cache of each CPU is invalidated. When the CPU polls shared-memory for changes, read requests are served by the L2 cache of the CPU. The CPU overheads of cache-coherent shared memory communication were therefore measured as the time to access L1 cache for writes and L2 cache for reads.

We used the CPU cycle counter of the ARM performance monitoring unit (PMU) to measure the above CPU overheads for reads and writes of various sizes using direct and shared memory communication. Out-of-order execution made it difficult to record precise timings so we performed two measurements. We measured a pessimistic execution time by flushing

TABLE I: Zynq CPU overheads for short transfers between CPU and FPGA, measured in CPU cycles.

Target	Words = 1	2	3	4	5	6
In-order execution						
L2 Cache read	28	41	54	67	80	93
FPGA read	76	86	103	119	145	164
L1 Cache write	7	7	8	9	10	11
FPGA write	14	17	20	23	26	29
Out-of-order execution						
L2 Cache read	12	25	38	51	64	77
FPGA read	60	70	87	103	129	148
L1 Cache write	1	1	2	2	3	3
FPGA write	1	1	4	7	10	13

the CPU store buffers and execution pipeline before and after the transfer instruction(s) were issued. We also measured a best-case execution time by adding 12 instructions that could be executed out-of-order after the transfer instruction(s). The memory attributes of reads and writes to the FPGA were configured to *Device memory* [5] as this setting provides the best performance [6]. The median CPU execution cycles of 100 measurements for both reads and writes are reported in Table I.

Our results show that, when the *Device memory* attribute is used, CPU overheads for direct communication were similar to those obtained for L1 cache writes when out-of-order execution is possible. This is because the *Device memory* attribute allows the CPU to continue to execute after issuing the transaction to the interconnect. On the other hand, the CPU overhead of reading directly from the FPGA is much higher than the overhead of an L2 cache read because the CPU stalls until the read transaction completes.

B. CPU-FPGA latency evaluation

A common reference of time was required when measuring the latency of communication between CPU and FPGA. We used the send event (SEV) instruction of the ARM-based Zynq to provide a low-latency signal at key points in the instruction stream. We assumed that this signal has no latency since it does not propagate through interconnects within the CPU or programmable logic.

We used the integrated logic analyser (ILA) soft IP core provided by Vivado to measure the elapsed FPGA clock cycles between events in programmable logic and the CPU. For direct communication from CPU to FPGA, we executed the SEV instruction immediately before a write instruction. We then measured the FPGA cycles from when the SEV instruction was observed until the data arrived at the CPU-FPGA boundary. The latency of a direct communication from FPGA to CPU was measured as the time from when the CPU executed the read instruction until the corresponding data arrived back at the CPU. For this reason, we used CPU overhead without out-of-order execution as our value for latency (Table I).

Shared memory communication latency was measured by first initialising a pre-determined word in memory to a known value. The sender was then configured to change the value of this word and the receiver was configured to continuously read this word until it observed that the value had changed. When the FPGA is the sender, we programmed the CPU to execute the SEV instruction when it observed a change in memory content. The latency was then measured as the time from when the write transaction arrived at the CPU-FPGA boundary, until the CPU executed the SEV instruction. When the FPGA was the receiver, the CPU executed the SEV instruction immediately before writing to shared memory. The latency was then measured as the time from when the SEV instruction was observed until the updated value appeared at the CPU-FPGA boundary.

TABLE II: Zynq communication latency between CPU and programmable logic.

Method	Direction	Latency (CPU cycles)
Direct	FPGA→CPU	76
	CPU→FPGA	32
Shared memory	FPGA→CPU	36
	CPU→FPGA	42

The results of our experiments (Table II) show that the latency of short transfers from the CPU to the FPGA are lowest when direct communication is used. For transfers from FPGA to CPU, shared memory provides the lowest latency.

Summarising our findings for both CPU overhead and latency of short transfers, direct communication should be used for short transfers from the CPU to the FPGA while shared memory communication should be used for short transfers from the FPGA to the CPU.

C. Hardware accumulator evaluation

We chose an accumulator task for our evaluation because the workload is easily divided and cooperatively executed on both CPU and FPGA. Although this application is trivial, it allows us to focus on workload partitioning and communication overhead, rather than on the underlying data processing algorithms. The methods used for this application can be applied directly to other fixed-size streaming workloads, such as vector multiplication.

We connected our accumulator core to a soft DMA controller, which provided a stream of integers from shared memory for processing. The DMA engine was configured using two direct writes from the CPU to the DMA engine via the GP port. These writes provided the address of the first integer, the number of bytes to transfer and also instructed the DMA engine to begin the transfer.

The DMA engine provided independent channels for up- and down-stream transfers. We used the second channel to transfer the result of the computation back to shared memory once the last integer had been accumulated. We assumed that this needed only to be configured once to configure a fixed location in shared memory for the result of all accumulator operations. For this reason, the overhead of programming the second channel was excluded from our experiments.

The DMA engine was connected to the ACP to avoid the overheads of cache-maintenance. Our IP was capable of processing two 32-bit words concurrently in a single cycle to

TABLE III: Accumulator model parameters given a 667 MHz CPU clock frequency.

Symbol	Value
X_F	594 MB/s
X_C	231 MB/s
D_F	401 ns
O_0	36 ns
O_C	81 ns
α	-56 N^{-1} + 0.72
N_L	96 B

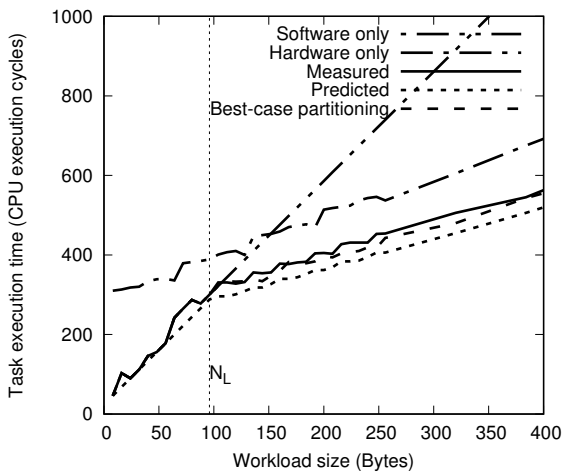

Fig. 2: Accumulator execution time for α partitioning.

match the 64-bit data bus of the ACP. The system throughput was thus limited by the bandwidth provided by the ACP.

We measured the final component of CPU overhead, O_0 by using the CPU cycle counter to measure the cost of calling the accumulator function with 0 integers to accumulate.

The final component of latency was found by using the ILA to measure the time between the DMA controller transaction arriving at the CPU boundary and the first integer pair arriving at the accelerator. Alternatively, this could be found from the DMA and interconnect IP core specifications.

Finally, the throughput of the CEs (X_F and X_C) was measured using the CPU cycle counter when calling the accumulator function with a very large workloads – the communication overheads can be ignored in this case. The complete set of parameters, including α and N_L, are shown in Table III.

We measured the completion time of the accumulator when using software- and hardware-only, as well as a cooperative system in which the partitioning was chosen by our model (Fig. 2). Included in our results is the expected execution time as predicted by our model. We also include the best-case result for each workload size, which was found experimentally by varying the partitioning ratio from 0.0 to 1.0 in increments of 0.001 and executing the task to completion.

Our results show that our model predicted the workload size N_L, for which the cooperative approach begins to outperform the software-only approach. This prediction was within 8% of the measured optimal value for the accumulator application. Our model is able to predict the completion time of the partitioned cooperative system with a mean relative error (MRE) of 12% for workloads in the range 8 B to 1 KB. Over the same range, the MRE between the modelled partitioning approach and the best-case observations was 2%.

V. Future work

We plan to apply our cooperative execution model to common short-running shared-library functions. In this way, many programs will benefit from hardware acceleration without modification. Shared libraries generally provide a blocking interface to the caller. For small workloads, we expect that the performance benefit of hardware acceleration will only be realised if the CPU processes some portion of the workload while it waits for the partial result to be returned by the accelerator.

VI. Conclusion

In this paper we presented a model for partitioning a task between a CPU and tightly-coupled programmable logic for all workload sizes. This model can be applied to short-running tasks as it considers both the CPU overhead and latency of communication between CE.

We have also reported the CPU overhead and latency of short direct and shared memory communication between the CPU and the FPGA on the Zynq device. These parameters, as well as interconnect delays, were used in our model to estimate the partitioning and completion time of the applications that we studied.

Our model was able to predict, within 8%, the workload size at which we benefit from task partitioning for a generic stream based application. The task completion time was predicted by our model with a MRE of 12%. We used our model to calculate a partitioning ratio that reduced task completion time to be within 2% of the best case.

Acknowledgment

This research was supported through an Australian Government Research Training Program Scholarship.

References

[1] J. W. Tang, Y. W. Hau, and M. Marsono, "Hardware/software partitioning of embedded system-on-chip applications," in *2015 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, pp. 331–336, Oct 2015.

[2] M. López-Vallejo and J. C. López, "On the hardware-software partitioning problem: System modeling and partitioning techniques," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 8, pp. 269–297, July 2003.

[3] M. Jacobsen and R. Kastner, "RIFFA 2.0: A reusable integration framework for FPGA accelerators," in *2013 23rd International Conference on Field programmable Logic and Applications*, pp. 1–8, Sept 2013.

[4] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, "Legup: High-level synthesis for fpga-based processor/accelerator systems," in *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '11, (New York, NY, USA), pp. 33–36, ACM, 2011.

[5] ARM limited, *ARMv7-A Architecture Reference Manual DDI 0406C.b*, 2005.

[6] A. Powell and D. Silage, "Statistical performance of the ARM Cortex A9 accelerator coherency port in the Xilinx Zynq SoC for real-time applications," in *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pp. 1–6, Dec 2015.