Throughput Optimization for Streaming Applications on CPU-FPGA Heterogeneous Systems

Xuechao Wei[1], Yun Liang[1], Tao Wang[1,3], Songwu Lu[2,1,3], Jason Cong[2,1,3]
[1]Center for Energy-Efficient Computing and Applications (CECA), School of EECS, Peking University, China
[2]UCLA Computer Science Department, U.S.A.
[3]PKU-UCLA Joint Research Institute in Science and Engineering, China
{xuechao.wei, ericlyun, wangtao}@pku.edu.cn, {slu, cong}@cs.ucla.edu

ABSTRACT

Streaming processing is an important technology that finds applications in networking, multimedia, signal processing, etc. However, it is very challenging to design and implement streaming applications as they impose complex constraints. First, the tasks involved in the streaming applications must complete the computation under a latency constraint. Second, streaming systems are built under more and more stringent power budget. Hence, power capping technique is employed to manage the power consumption for streaming systems. To accommodate these needs, heterogeneous systems that consist of CPUs and FPGAs are becoming increasingly popular due to their performance and power benefits.

In this paper, we optimize the throughput for streaming applications on CPU-FPGA heterogeneous system under latency and power constraints. We develop two algorithms to map the tasks onto the heterogeneous system and order their execution by exploiting the heterogeneity in architectural capabilities and task characteristics. We also employ pipelining to improve the throughput by overlapping the execution of different frames and use frequency scaling to adjust the execution of tasks for power saving. Experiments using a variety of streaming applications show that our heterogeneous solution can successfully meet the latency and power constraints for the cases where the CPU implementation fails. Furthermore, our technique can improve the throughput by 37.32% on average.

1. INTRODUCTION

Data streaming applications such as multimedia, signal processing, and network protocol are becoming prevalent due to the rapid growth of mobile phones, IoT devices, and wireless network connectivity. However, design and implementation of streaming applications raise several challenges. First, streaming applications typically have latency constraints. As a result, the tasks involved in the streaming applications must complete by a certain amount of time for responsiveness. Second, lots of systems running streaming applications are operated by stand-alone battery and thus power has become a growing concern for such systems. In order to increase the operation time, streaming systems have to cap their power footprint to the predetermined level.

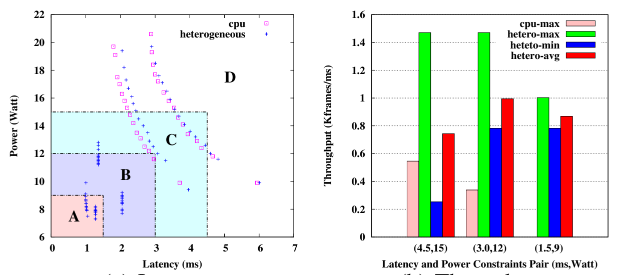
To overcome these challenges, heterogeneous systems that couple CPUs, GPUs and FPGAs have emerged as a promising solution, as it delivers orders of magnitude performance and energy benefits compared to general purpose processors [21]. The major heterogeneous System-on-Chips (SoCs) available in the market include NVIDIA Tegra serious with GPU, Xilinx Zynq with FPGAs, etc. In this paper, we present algorithms and modeling techniques in an attempt to map streaming applications onto heterogeneous systems that consist of CPUs and FPGAs. FPGA-based heterogeneous system has the potential to address the increasing latency and power challenge for streaming applications as its flexible architecture can enable efficient hardware customization for particular algorithms or tasks. For example, FPGA-based systems have been deployed to accelerate the Bing web search engine, leading to high throughput and low power [20]. More importantly, High Level Synthesis (HLS) has lowered the barrier to FPGA programming [16, 25]. Software programmers can use FPGA by writing high level programming codes in C, C++, SystemC, Haskell and CUDA [6].

In this paper, we are interested in maximizing throughput for streaming applications subject to latency and power constraints. Given a streaming application represented by synchronous data flow graph (SDFG) [15], an important problem is how to assign the tasks to heterogeneous resources and order their execution to maximize the throughput and keep the latency and power under constraints. The power constraint is defined for a heterogeneous computing node that consists of CPUs and FPGAs and latency constraint is defined for an entire frame of the streaming applications. Different task mapping strategies lead to different designs, which exhibit large variations in terms of latency, power and throughput. In addition, in order to improve the throughput we also employ pipeline design to overlap the executions of different frames by paralleling their tasks. Meanwhile, we use dynamic voltage and frequency scaling (DVFS) to stretch the execution on the CPU side and lower the power consumption. Collectively, the design space enabled by the heterogeneous architectures are expected to provide more flexibility for latency, power and throughput.

Researchers are always looking for better optimization techniques for streaming applications [14, 22, 10, 13, 19, 5, 4]. However, none of them consider the task heterogeneity within streaming applications. As far as we know, our work is the first throughput optimization technique for heterogeneous systems while considering both the latency and power constraints of the system.

Motivating Example. We now motivate the need of throughput optimization for streaming applications on heterogeneous CPU-FPGA system using application *DCT*. *DCT* is a classic streaming application that is widely used in image transformation and signal processing. It consists of two tasks. Given an input matrix, it invokes tasks *dct_2d* and *transpose* in sequence. We collect the latency and power characteristics of the two tasks on CPU and FPGA, respectively. For FPGA implementation, we use Vivado HLS tool to generate the HDL design from original C code. The results are shown in Table 1. As shown, different tasks exhibit heterogeneity in utilization of different computing resources. For example, in terms of latency, task *dct_2d* benefits from the hardware customization on FPGA, but task *transpose* prefers CPU execution. For both tasks, FPGA implementation shows remarkable power savings compared to CPUs thanks to its low frequency.

Figure 1 depicts the design space of CPU only and heterogeneous CPU-FPGA systems, respectively. For each design point,



(a) Latency (b) Throughput

Figure 1: Design space and throughput results.

it exhibits different tradeoff in latency, power and throughput. For both CPU only and heterogeneous systems, we can use pipeline design to parallelize *dct_2d* and *transpose*. If we use CPU only, there exists 30 design points by exploring different pipelining mechanisms and frequency levels. If we use heterogeneous CPU-FPGA system, the task mapping enlarges the design space, leading to 120 design points. We divide the design space into four areas, A, B, C, D as shown in Figure 1 (a). The power and latency constraints are relaxed as the design points move from area A to D.

It is clear that heterogeneous CPU-FPGA systems provide more opportunities to meet the stringent power and latency constraints for streaming applications as shown in Figure 1 (a). For example, dozens of design points from heterogeneous CPU-FPGA system satisfy the power and latency constraints of area A and B, but none of the design points from CPU only system falls in these areas. More importantly, different points in the same area yield different throughput. Figure 1(b) compares the maximal throughput of CPU system with the maximum, average and minimum throughput of heterogeneous CPU-FPGA system for area A, B and C, respectively. As shown, there is large variation in terms of the achieved throughput. Therefore, by exploring the design space, we have a large potential to improve the throughput for streaming applications under power and latency constraints.

Task	CPU		FPGA	
	time	power	time	power
dct_2d	1.80ms	20.1w	0.50ms	2.7w
transpose	0.17ms	19.5w	0.36ms	2.6w
transfer	0.08ms (3.125 GB/s)			

Table 1: Latency and power profiling of *DCT*'s tasks.

In this paper, we present a throughput optimization framework for streaming applications on heterogeneous CPU-FPGA system under power and latency constraints. We judiciously map the tasks onto CPU and FPGA to exploit the heterogeneity in tasks and architectures. The task mapping problem has been shown to be NP-complete [9]. In this work, we develop two algorithms. One is an optimal solution based on branch and bound. Another one is a heuristic algorithm. Our heuristic algorithm first balances the workload between CPU and FPGA using max-flow min-cut algorithm. Then, it migrates some tasks between FPGAs and CPU for further improvement. In addition, we also explore pipeline design to overlap the executions of different frames by parallelizing the tasks on CPUs and FPGAs together. The throughput of the pipeline design depends on the longest pipeline stage. We notice that different pipeline stages are varied, finishing the shorter stages earlier is not beneficial, thus we use dynamic voltage and frequency scaling (DVFS) to stretch its execution and lower the power consumption. Our main contribution includes,

- Framework. We develop a framework for optimizing throughput for power and latency constrained streaming applications on heterogeneous CPU-FPGA system.
- Algorithm. We develop task mapping algorithms that intelligently map the tasks onto CPUs or FPGAs based on architectural capabilities and task characteristics.
- Optimization. We present pipelining and frequency scaling optimization techniques to improve the throughput and power saving.

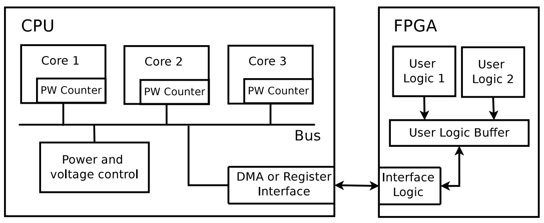
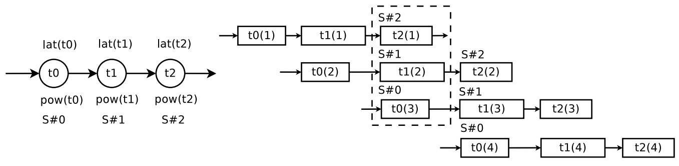


Figure 2: System architecture.



(a) SDFG labelled with latency and power attributes.

(b) Overlapped consecutive frames in different pipeline stages.

Figure 3: An SDFG example.

We implement our technique on a heterogeneous system that consists of an Intel multicore CPU and a Xilinx VC707 FPGA. Our experiments using a variety of streaming applications demonstrate that our algorithms not only can fulfill the power and latency constraints, but also improve the throughput by 37.32% on average compared to the CPU implementations.

2. SYSTEM ARCHITECTURE AND MODEL

In this section, we describe the architecture details of our heterogeneous CPU-FPGA system and the application model.

Heterogeneous Systems. We consider the CPU-FPGA system as shown in Figure 2. The heterogeneous system consists of a FPGA coupled with a host CPU with high speed transfer integration. The CPU is composed of more than one cores and different cores can be used for executing different tasks concurrently. The voltage and frequency can be changed through the power controller. We use Vivado High Level Synthesis (HLS) tool to convert the high level description of the tasks into HDL on FPGAs.

Application Model. The streaming applications are modeled as $G(V, E)$ using SDFGs, where V represents the set of nodes and E represents the set of edges. Each node or actor models a task and the edges between the nodes model the dependencies between the tasks. The tasks are referred as actors that communicate with tokens sent from one actor to another through the edges. For each task, it is associated with a few attributes including its latency and power consumption if mapped to FPGA and CPU. For each edge, it is associated with attributes including the number of tokens and the memory required to transfer between the source and destination. SDFG has a periodic iteration behaviour [4]. A streaming application usually divides its input data into frames and fully processes each frame in one iteration before the next one arrives. Processing of consecutive frames can be overlapped in a pipeline manner. Figure 3(a) shows an example of SDFG containing 3 tasks. For each task, lat and pow denote its latency and power attributes respectively. When pipeline is applied, as Figure 3(b) shows, 3 tasks are divided into 3 pipeline stages ($S\#0$, $S\#1$ and $S\#2$) to do computation for 3 consecutive frames in parallel. The frames overlap with each other on different tasks. For example, frame 1, 2 and 3 overlap on task t2, t1 and t0, respectively. t0(3), t1(2) and t2(1) in $S\#0$, $S\#1$ and $S\#2$ denote this scenario in Figure 3(b).

3. PROBLEM FORMULATION

Given a stream of n actors/tasks, $T = \{t_1, \dots, t_n\}$, we optimize its throughput when mapped onto the heterogeneous CPU-FPGA system. We use $lat(t_i)$ and $pow(t_i)$ to represent the latency and power of task t_i. Obviously, $lat(t_i)$ and $pow(t_i)$ depend on where the task t_i is mapped to. For task t_i, we use binary variables m_i to represent its mapping,

$$m_i = \begin{cases} 1 & \text{if } t_i \text{ is mapped to FPGA} \\ 0 & \text{if } t_i \text{ is mapped to CPU} \end{cases} \quad (1)$$

Then,

$$lat(t_i) = m_i \times lat_{fpga}(t_i) + (1 - m_i) \times lat^f_{cpu}(t_i) \quad (2)$$

$$pow(t_i) = m_i \times pow_{fpga}(t_i) + (1 - m_i) \times pow^f_{cpu}(t_i) \quad (3)$$

where $lat_{fpga}(t_i)$ ($pow_{fpga}(t_i)$) denote the latency/power of task t_i on the FPGA, respectively. On the CPU, frequency level affects the latency and power. We use $lat^f_{cpu}(t_i)(pow^f_{cpu}(t_i))$ to denote the latency/power of task t_i on the CPU at frequency f. $lat_{fpga}(t_i)(pow_{fpga}(t_i))$ and $lat^f_{cpu}(t_i)(pow^f_{cpu}(t_i))$ are collected during the profiling stage as shown in Figure 4, which are the attributes of the nodes in SDFG.

In order to increase the throughput, we employ pipeline execution mechanism as shown in Figure 3(b). More clearly, given a stream of n actors/tasks, we first transform it into a linked list by merging the splitter node if there are any [8]. Then, we divide the n tasks into m stages $\{s_1, \dots, s_m\}$ for pipeline execution, where $1 \leq m \leq n$. A pipeline stage is formed by a set of consecutive tasks. If $m = 1$, then all the tasks will run sequentially. If $m = n$, then all the tasks will run in parallel. Figure 3(b) illustrates our pipeline execution. In this example, there are 3 tasks in the SDFG and each task forms a stage. Thus, we can process three consecutive frames in parallel by executing different tasks for them. When one frame finishes its execution, next frame will enter the pipeline.

For pipeline stage s_i, its latency models both the computation time of the tasks in it and communication time for tasks with dependency,

$$lat_{s_i} = \sum_{start(s_i) \leq k \leq end(s_i)} lat(t_k) + e(t_k, t_{k+1}) \quad (4)$$

where $start(s_i)$ and $end(s_i)$ represent the starting and ending task indices for pipeline stage s_i. $e(t_i, t_j)$ represents the communication time between task t_i and t_j. $e(t_i, t_j)$ not only depends on the size of the data transfer, but also where the source and destination are mapped to.

The latency of different pipeline stages may vary. However, the throughput of the streaming applications is determined as the inverse of the longest pipeline stage as follows,

$$Throughput = \frac{1}{max_{s_1 \leq s_i \leq s_m} lat_{s_i}} \quad (5)$$

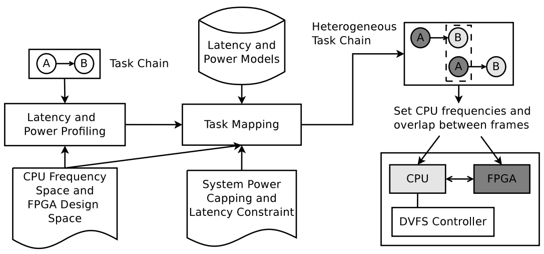
Streaming systems are operated under complex latency and power constraints. We model these two factors as follows,

Latency Model. For streaming systems, the end-to-end latency of one frame should not violate the constraint. We define the latency of one frame,

$$Lat = m \times max_{1 \leq s_i \leq s_m} lat_{s_i} \quad (6)$$

Power Model. The system power Pow is composed of four parts, which include the system active power P^{active}, idle power of CPU P^{idle}_{cpu} and FPGA P^{idle}_{fpga}, and the power of communication logic on FPGA P^{comm}_{fpga},

$$Pow = \sum_{s_1 \leq s_i \leq s_m} P^{active}_{s_i} + P^{idle}_{cpu} + P^{idle}_{fpga} + P^{comm}_{fpga} \quad (7)$$



A
B
Task Chain
Latency and Power Models
Heterogeneous Task Chain
Latency and Power Profiling
Task Mapping
Set CPU frequencies and overlap between frames
CPU Frequency Space and FPGA Design Space
System Power Capping and Latency Constraint
CPU
FPGA
DVFS Controller

Figure 4: Optimization flow.

where $P^{active}_{s_i}$ is the active power consumption of pipeline stage s_i. It is computed as the ratio of active energy of this stage and the longest pipeline stage. We use pow^{wait}_{cpu} to denote the power consumption of CPU when it is waiting for FPGA to finish, or waiting for stage ends after finish all tasks' computation. Then,

$$P^{active}_{s_i} = \frac{E^{active}_{s_i}}{max_{s_1 \leq s_i \leq s_m} lat_{si}} \quad (8)$$

$$\begin{aligned} E^{active}_{s_i} = & \sum_{start(s_i) \leq k \leq end(s_i)} (pow(t_k) \times lat(t_k) \\ & + m_i \times pow^{wait}_{cpu} \times lat(t_i)) \\ & + pow^{wait}_{cpu} \times (Lat - lat_{s_i}) \end{aligned} \quad (9)$$

P^{idle}_{cpu} and P^{idle}_{fpga} can be measured when there is no workload on the CPU and FPGA. And P^{comm}_{fpga} can be integrated in P^{active} when there is task executing on FPGA.

Finally, we formulate the optimization problem as follows,

PROBLEM 1. *Let $T = \{t_1 \cdots t_n\}$ be a stream of n actors executing on a CPU-FPGA heterogeneous system. Let Pow_{cap} be the predetermined power capping for the heterogeneous system and Lat_{cap} be the latency deadline for one frame of the streaming application, we aim to maximize the throughput subject to $Pow \leq Pow_{cap}$ and $Lat \leq Lat_{cap}$.*

4. OPTIMIZATION OVERVIEW

Figure 4 depicts the optimization flow. Given a streaming application represented by SDFG, we first profile each actor/task to collect its latency and power consumption on CPU and FPGA. For the CPU profiling, we collect the latency and power consumption for each task at different frequency level; for the FPGA profiling, we execute the task on FPGA and measure the latency and power. Then, we annotate the SDFG with those characteristics.

The core function is the task mapping component. The goal of task mapping algorithm are two folds. First, it determines how the tasks are mapped to the heterogeneous system. More clearly, for each task the algorithm determines the processor (FPGA or CPU) it is mapped to. If the task is mapped to CPU, it also determines the corresponding execution frequency. Second, we employ pipelining to overlap the execution of different tasks for high throughput. Our algorithms determine the number of pipeline stages and the composition of each stage in terms of tasks. In the following, we will formulate the optimization problem and describe the task mapping algorithms.

5. ALGORITHMS

We propose two algorithms for Problem 1. One is optimal algorithm based on branch and bound. Another one is a heuristic algorithm.

5.1 Optimal Algorithm

We first devise an optimal algorithm that explores the design space using branch and bound. Algorithm 1 presents the details.

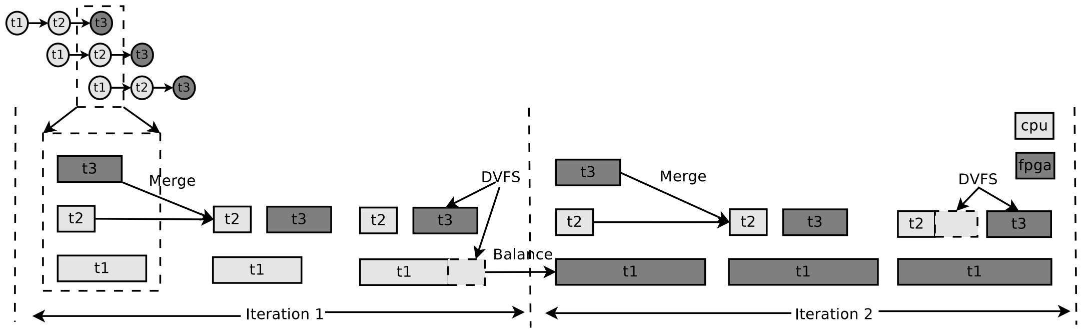


Figure 5: Heuristic algorithm case.

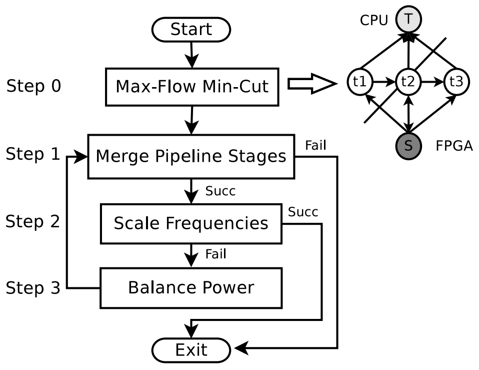


Figure 6: Flowchat of the heuristic algorithm.

Given n tasks/actors, there exists 2^{n-1} ways to divide the tasks into pipeline stages. In Algorithm 1, we first explore different pipeline mechanisms (lines 1-5) and then for each task, we explore its mapping (CPU or FPGA) (lines 3-5) and frequency if (line 5) it is mapped to CPU. Each call of the `DFSFreqSpace` for the last task generates a final mapping, where we can derive the throughput, latency and power results. Obviously, Algorithm 1 runs in $O(2^n 2^n f^n)$ time in the worst case.

Algorithm 1: Optimal algorithm.

```
Data: stream $S$ with $n$ tasks, frequency space $FS$ with $f$ frequencies,
      latency constraint $Lat_{cap}$ and power constraint $Pow_{cap}$
Result: A mapping $M$ for $S$ and a maximal throughput $Throughput$
        under $Lat_{cap}$ and $Pow_{cap}$
1  for $i = 0$ to $2^{n-1} - 1$ do
2      $PipelineStageDivide(S, M, i)$
3      for $j = 0$ to $2^n - 1$ do
4          $ProcTypeMap(S, M, j)$
5          $DFSFreqSpace(0)$
6  DFSFreqSpace ($k$) begin
7      if $k = n$ then
8          if $Lat_{cap}$ and $Pow_{cap}$ are not violated then
9              Update $M$ and $Throughput$ if necessary
10     for $i = 0$ to $f - 1$ do
11         $TS[k].SetFreq(FS(i))$
12         $DFSFreqSpace(k + 1)$
```

We develop pruning strategies for efficiency. The idea is to stop further search as soon as the power and latency constraints are violated. First, we explore the frequency on the CPU in the ascending order. If the task mapped on CPU leads to a power cap violation at certain frequency, then we stop searching the frequencies greater than it. Second, after we set frequencies for all the actors in a pipeline stage, we test whether the length of current stage will violate the latency constraint. These strategies could be applied before `DFSFreqSpace` for task `k+1`.

5.2 Heuristic Algorithm

The complexity of the optimal algorithm is exponential in the worst case. Hence, we also develop an efficient heuristic algorithm. Figure 6 depicts the flow of our heuristic algorithm. We first formulate a max-flow min-cut problem, denoted as step 0. The goal of this step is to minimize the total latency and data transfer time for the tasks mapped onto the heterogeneous CPU-FPGA system. More clearly, we add one source node to represent CPU and one sink node to represent the FPGA into the SDFG. For each node v, we connect it with source and sink node. The edges between v and the source (sink) are weighted by the latency on the sink (source) node. For the other edges, they are weighted by transfer time if the source and destination node are mapped to different processor.

The output of the max-flow min-cut step is an initial task mapping. Then, the heuristic algorithm tentatively iterates three steps. In step 1, initially, we divide n tasks into n stages, each task corresponds to one stage. Then, it merges the neighboring stages until the latency is within the latency constraint. In step 2, it attempts to scale down the frequency for the tasks that have latency slack compared with the longest stage, in order to decrease the total power consumption under power capping. If the power still cannot satisfy the power constraint, we will enter step 3. In step 3, we implement a power balancing function to choose a task to do pipeline merge again or offload it onto FPGA. We define a metric for each task to measure the power balancing benefit after the merging or offloading, as shown in Equation 10.

$$BLCBenefit_i = PowerDec_i / LatencyInc_i \tag{10}$$

In Equation 10, $PowerDec_i$ denotes the power decrement after balancing the ith task, and $LatenchInc_i$ denotes its latency increment. Larger $BLCBenefit_i$ means more benefit we could get by choosing t_i to do balancing. We offload the task with the largest $BLCBenefit$ at each iteration. After step 3, it will go back to step 1 until latency and power constraints are both satisfied.

Figure 5 presents an example depicting how the heuristic algorithm works on a 3-task stream. Initially, $t1$ and $t2$ are mapped on CPU, and $t3$ is mapped on FPGA by the max-flow min-cut step. They forms a 3-stage pipeline. At step 1 in the first iteration, tasks $t2$ and $t3$ are merged to form a new stage from the initial pipeline organization. Then at the second step, $t1$ is stretched to save power. Unfortunately, power capping is stall not satisfied now. Then step 3 chooses $t1$ as it has the highest balancing benefit. And its balancing method is being offloaded from CPU onto FPGA. In the second iteration, after merging of $t2$ and $t3$, and then scaling frequency of $t1$, $t2$ and $t3$, we get a legal mapping satisfying all constraints.

In the worst case where latency constraint can be met while power constraint cannot be satisfied, the main loop runs in $O(n^2 f)$ time. Thus the heuristic algorithm runs in $O(n^2 max\{f, n\})$ time, where n is the number of tasks, and f is the number of frequencies.

6. EXPERIMENT

6.1 Experimental Setup

The heterogeneous CPU-FPGA system consists of an Intel Core i5 CPU processor and a Xilinx VC707 FPGA. The CPU has four cores. Each CPU core supports 12 different frequency levels, which ranges from 1.6GHz to 3.1GHz. The communication between CPU and FPGA is through PCI Express. In Gen2 mode and 8 lanes configuration, the maximal data rate could achieve up to 3.125 GB/s (each lane supports maximal 800 MB/s data rate).

We use a benchmark suite consisting of several representative streaming applications from various areas. *AES* (2 actors) and *MD5* (2 actors) are from security area, *DCT* (4 actors) and *JPEG* (6 actors) are from image processing area, and a *simple LTE receiver* (3 actors) and a *complete LTE receiver* (8 actors) are from wireless communication area. For each application, we first profile the latency and power of each actor under different operating frequency levels on CPU and FPGA platform, respectively[1]. The power consumed by CPU is measured through Intel's Running Average Power Limit (RAPL) interface. On FPGA, we use Xilinx's Vivado tool to implement the optimized hardware designs. We obtain a design's estimated latency through high level synthesis process by Xilinx's High Level Synthesis (HLS) tool. The power consumption on FPGA by the design is reported by Xilinx's power estimator.

We implement our optimal and heuristic algorithms by SDF^3 tool set [23]. We use POSIX Threads interfaces to realize concurrency of actors in different pipeline stages. And threads synchronizations are manually inserted after two actors when they are both mapped on FPGA and reside on different pipeline stages. These synchronizations guarantee the correct read and write orders during their communication with FPGA.

6.2 Results

6.2.1 *Performance improvement*

We compare three solutions, which are the optimal solution on CPU only (*cpu-opt*), the optimal and heuristic solution on heterogeneous CPU-FPGA (*hetero-opt*, *hetero-heu*). For the optimal solution on CPU, we use the same algorithm (Algorithm 1) but only map to CPU cores. For each application, we set nine pairs of latency and power constraints.

We first compare *cpu-opt* and *hetero-heu* on heterogeneous CPU-FPGA solution. The results in Figure 7 show that of all 54 constraint pairs, *hetero-heu* satisfies them all, while *cpu-opt* fails 15 pairs. Of the pairs for which both *cpu-opt* and *hetero-heu* could satisfy, the *hetero-heu* can achieve 37.32% throughput improvement on average compared with *cpu-opt*. CPU usually consumes much more power than FPGA for most tasks in the streaming applications. As power constraint becomes rigid, we cannot map tasks of a streaming application totally on CPU. For example, offloading *AES*'s *inversshiftrow* task can reduce the application's power from 11 Watt to 8 Watt under latency constraint 2 milliseconds and 10 Watt power constraints because *inversshiftrow* consumes 10.7 Watt on CPU and 0.64 Watt on FPGA. On another hand, as latency constraint becomes rigid, the heterogeneity in those applications helps reduce latency. For example, in the case of *DCT* under latency constraint 2 milliseconds and power constraint 30 Watt, *cpu-opt* cannot satisfy the latency constraint because task *dct_2d* executes 1.8 milliseconds on CPU and 0.5 milliseconds on FPGA. For the cases where both cpu-opt and hetero-heu satisfy constraints, heterogeneity again helps improve throughput. The case of *DCT* under 4 milliseconds and 20 watt illustrates this due to the same reason of *dct_2d*'s latency benefit on FPGA.

[1]For the FPGA platform, there is only one operating frequency level are profiled.

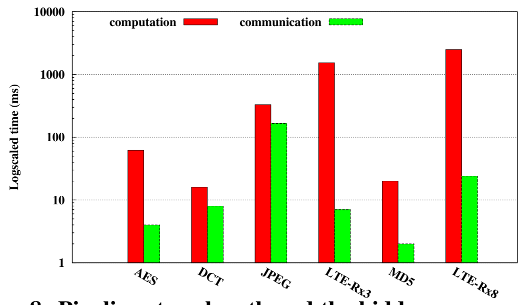


Figure 8: Pipeline stage length and the hidden communication overhead.

We also compare the two heterogeneous solutions from the perspectives of accuracy and running time. Figure 7 shows that of all pairs of latency and power constraints, *hetero-heu* could achieve optimal throughput out of a total of 39 pairs. Overall, the throughput of *hetero-heu* is within 4.95% of *hetero-opt*. But *hetero-heu* achieves this accuracy in a very short running time. Table 2 compares the run-time of *hetero-heu* and *hetero-opt*. It shows that our heuristic algorithm is much efficient than the optimal solution.

6.2.2 *Hiding of communication overhead*

Here we demonstrate how our algorithms overlap the communication with computation. Figure 8 shows the computation and communication statistics for all the benchmarks under the the first constraint setting in Figure 7. The *computation* denotes the length of the longest pipeline stage and *communication* denotes the sum of communication overhead between CPU and FPGA across other stages. We could see that by our algorithms, the communication time in pipeline stages is overlapped with the longest stage. For all the other settings in Figure 7, the same conclusion holds.

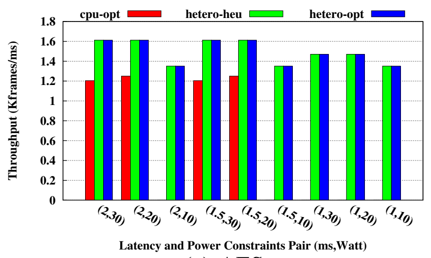
Table 2: Runtime comparison between two algorithms.

Benchmark	Hetero-Heuristic	Hetero-Optimal	Speedup
AES	0.29ms	1.20ms	4.14
DCT	1.67ms	40.13ms	24.47
JPEG	6.49ms	105.60s	16271
LTE-Rx3	0.43ms	12.53ms	29.14
LTE-Rx8	20.18ms	>4 hours	N/A
MD5	0.24ms	2.07ms	8.63

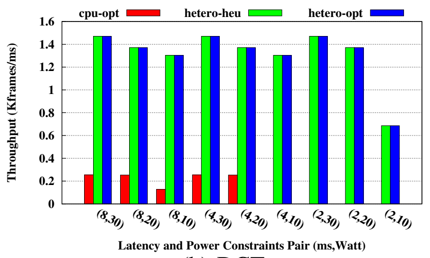
7. RELATED WORK

There exists a rich set of studies on optimization for streaming applications. FPGAs have been used to realize the streaming applications efficiently. The work in [11] implements a compilation strategy to map streaming application onto FPGA by several stream specific optimizations to improve performance. By judiciously replicating the bottleneck actors in a stream, the work in [10] develops an algorithm to optimize throughput of streaming applications on FPGA subject to FPGA area and latency constraints. The authors of [5, 4] minimize total FPGA area cost subject to throughput constraint. They firstly use replication and module selection together to improve throughput [5]. Then, they take communication into consideration to achieve the area optimization objective [4]. Multicore architecture has been used for streaming application, too. The work in [8] uses a greedy algorithm to judiciously split or fuse together tasks in stream according to the relationship between tasks' computation requirements and the number of cores. The authors of [7, 14] also consider the effect of communication when doing the mapping. The work in [7] reduces communication cost introduced by replication by fusing and replicating filters, while the work in [14] proposes a graph partitioning strategy to overlap communication and computation.

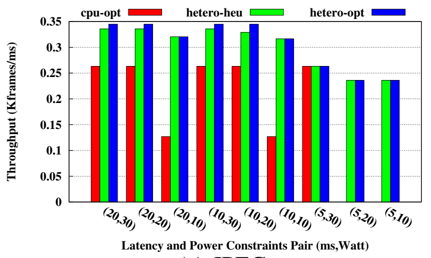
Power and energy are also considered when optimizing streaming applications. Dynamic voltage and frequency scaling for slack



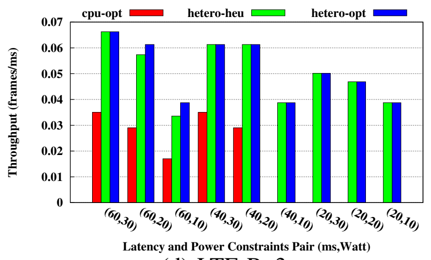
(a) AES



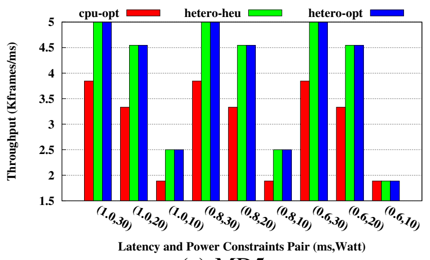
(b) DCT



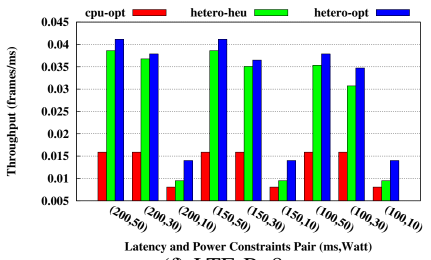
(c) JPEG



(d) LTE-Rx3



(e) MD5



(f) LTE-Rx8

Figure 7: Throughput improvement.

reclamation to save power has been used on Multiprocessor Systems-on-Chip (MPSoC) [22, 1, 18, 24, 2, 3, 17]. On-line DVFS techniques are applied in [1, 18, 24] to utilize dynamically created slack to reduce overall energy consumption. The work in [22] aims to minimize energy consumption of streams using both off-line and on-line analysis, subject to throughput constraints. Power gating is also used to reduce energy consumption of streaming applications. The authors of [12] proposes a technique to improve the energy efficiency of FPGA devices by exploiting power gating during idle periods in streaming applications. Compared to prior works, our work optimizes throughput subject to both power and latency constraints for heterogeneous CPU-FPGA system for the first time. This is a more challenging problem as it requires careful modeling the trade-off among latency, power and throughput.

8. CONCLUSION

We present a throughput optimization framework for streaming applications on heterogeneous CPU-FPGA system under power and latency constraints. By mapping the tasks onto heterogeneous system and employing pipelining and frequency scaling, we aim to optimize the throughput. We formulate the problem and develop two algorithms for it. Experiments using a variety of streaming applications show that our heterogeneous solution can successfully meet the latency and power constraints for the cases where the CPU implementation fails. Furthermore, our technique can improve the throughput by 37.32% on average.

9. ACKNOWLEDGMENTS

This work was supported in part by the National Natural Science Foundation of China under Grant 61672048.

10. REFERENCES

[1] J.-J. Chen, C.-Y. Yang, and T.-W. Kuo. Slack Reclamation for Real-time Task Scheduling Over Dynamic Voltage Scaling Multiprocessors. In *SUTC'06*.

[2] P. Choudhury, P. Chakrabarti, and R. Kumar. Online Dynamic Voltage Scaling using Task Graph Mapping Analysis for Multiprocessors. In *VLSI'07*.

[3] J. Cong and K. Gururaj. Energy efficient multiprocessor task scheduling under input-dependent variation. In *DATE'09*.

[4] J. Cong, M. Huang, and P. Zhang. Combining Computation and Communication Optimizations in System Synthesis for Streaming Applications. In *FPGA'14*, pages 213–222.

[5] J. Cong, M. Huang, P. Zhang, and Z. Yi. Combining Module Selection and Replication for Throughput-driven Streaming Programs. In *DATE'12*, pages 1018–1023.

[6] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang. High-Level Synthesis for FPGAs: From Prototyping to Deployment. *TCAD*, 30:473–491, 2011.

[7] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting Coarse-grained Task, Data, and Pipeline Parallelism in Stream Programs. In *ASPLOS'06*, pages 151–162.

[8] M. I. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, A. A. Lamb, C. Leger, J. Wong, H. Hoffmann, D. Maze, and S. Amarasinghe. A Stream Compiler for Communication-exposed Architectures. In *ASPLOS'02*.

[9] R. Graham, E. Lawler, J. Lenstra, and A. Kan. Optimization and Approximation in Deterministic Sequencing and Scheduling: a Survey. *Annals of Discrete Mathematics*, 5:287–326, 1979.

[10] A. Hagiescu, W. fai Wong, D. F. Bacon, and R. Rabbah. A Computing Origami: Folding Streams in FPGAs. In *DAC'09*, pages 282–287.

[11] A. Hormati, M. Kudlur, S. Mahlke, D. Bacon, and R. Rabbah. Optimus: Efficient realization of streaming applications on fpgas. In *CASES'08*, pages 41–50.

[12] M. Hosseinabady and J. Nunez-Yanez. Energy Optimization of FPGA-based Stream-Oriented Computing with Power Gating. In *FPL'15*.

[13] H. P. Huynh, A. Hagiescu, W.-F. Wong, and R. S. M. Goh. Scalable framework for mapping streaming applications onto multi-gpu systems. In *PPoPP'12*.

[14] M. Kudlur and S. Mahlke. Orchestrating the Execution of Stream Programs on Multicore Platforms. In *PLDI'08*.

[15] E. Lee and D. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, Sept 1987.

[16] Y. Liang, K. Rupnow, Y. Li, D. Min, M. N. Do, and D. Chen. High-level synthesis: Productivity, performance, and software constraints. *JECE*, 2012:1:1–1:1, Jan. 2012.

[17] J. Luo and N. Jha. Static and dynamic variable voltage scheduling algorithms for real-time heterogeneous distributed embedded systems. In *ASP-DAC'02*.

[18] P. Malani, P. Mukre, Q. Qiu, and Q. Wu. Adaptive Scheduling and Voltage Scaling for Multiprocessor Real-time Applications with Non-deterministic Workload. In *DATE'08*.

[19] D. Nguyen and J. Lee. Communication-aware mapping of stream graphs for multi-gpu platforms. In *CGO'16*.

[20] A. Putnam, A. Caulfield, E. Chung, and et al. A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services. In *ISCA'14*.

[21] Y. S. Shao, B. Reagen, G.-Y. Wei, and D. Brooks. Aladdin: A pre-rtl, power-performance accelerator simulator enabling large design space exploration of customized architectures. In *ISCA'14*.

[22] A. K. Singh, A. Das, and A. Kumar. Energy Optimization by Exploiting Execution Slacks in Streaming Applications on Multiprocessor Systems. In *DAC'13*.

[23] S. Stuijk, M. Geilen, and T. Basten. SDF^3: SDF For Free. In *ACSD'06*.

[24] D. Zhu, R. Melhem, and B. Childers. Scheduling with Dynamic Voltage/Speed Adjustment using Slack Reclamation in Multiprocessor Real-time Systems. *IEEE Transactions on Parallel and Distributed Systems*, 2003.

[25] W. Zuo, Y. Liang, P. Li, K. Rupnow, D. Chen, and J. Cong. Improving high level synthesis optimization opportunity through polyhedral transformations. In *FPGA'13*, pages 9–18, 2013.