

Balancing Graph Processing Workloads Using Work Stealing on Heterogeneous CPU-FPGA Systems

Matthew Agostini, Francis O'Brien and Tarek S. Abdelrahman

The Edward S. Rogers Sr. Department of
Electrical and Computer Engineering
University of Toronto

matt.agostini@mail.utoronto.ca

francis.obrien@mail.utoronto.ca

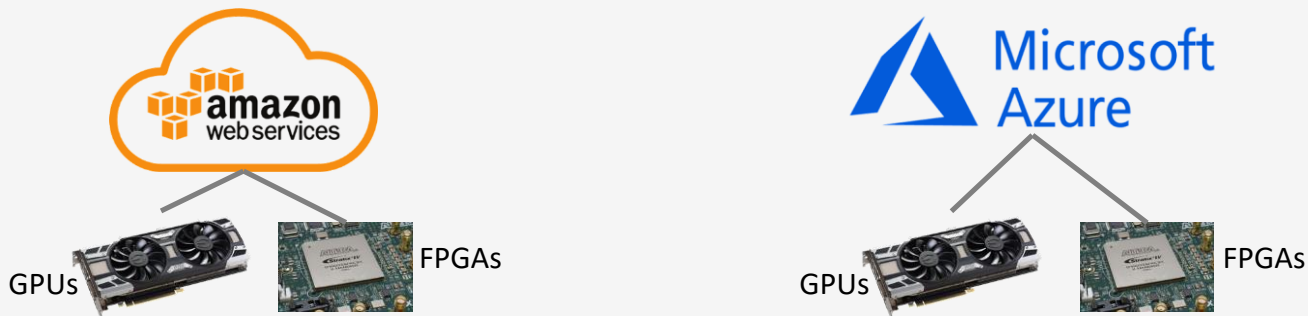
tse@ece.utoronto.ca

Outline

- Research context and motivation
- Work stealing
- Heterogeneous Work Stealing (HWS)
- Evaluation
- Related work
- Conclusions and future work

Accelerator-based Computing

- Accelerators are prevalent in computing, from personal to cloud platforms



- Field Programmable Gate Arrays (FPGAs)
 - Enable user-defined application-specific circuits
 - Potential for faster more power-efficient computing

Emerging FPGA-Accelerated Servers

- A new generation of high-performance systems *tightly integrate* FPGAs with multicores, targeting data centers
 - Exemplified by the Intel HARP, IBM CAPI and Xilinx Zynq systems
- An FPGA circuit can *directly* access system memory in a manner that is coherent with the processor caches
 - Enables CPU threads and FPGA hardware to *cooperatively* accelerate an application, sharing data in system memory
 - In contrast to typical *offload* FPGA acceleration that leaves the CPU idle during FPGA processing

Research Question

- The concurrent use of (multiple) CPU threads and FPGA hardware requires *balancing* of workloads
- **Research question:** how to balance workloads between software (CPU threads) and hardware (FPGA accelerator) such that:
 - The accelerator is fully utilized
 - Load imbalance is minimized
 - Scheduling overheads are reduced

Graph Analytics

- We answer our research question in the context of *Graph Analytics*: applications that process large graphs to deduce some properties
 - Prevalent in social networks, targeted advertising and web searches
- Processing graphs is notoriously *load imbalanced*
 - Graph structure (varying outgoing edge degrees)
 - Distribution of active/inactive vertices (computations vary across processing iterations)

This Work

- We develop *Heterogeneous Work Stealing (HWS)*: a strategy for balancing graph processing workloads on tightly-coupled CPU+FPGA systems
 - Identify and address some unique challenges that arise in this context
- We implement and evaluate HWS on the Intel HARP platform
 - Use it for 3 kernels processing large real-world graphs
 - Effectively balances workloads
 - Outperforms state-of-the-art strategies
- Supported by Intel Strategic Research Alliance (ISRA) grant

Work Stealing

```
Thread T[i](Workload:workItems)
while true do
  if has workItems then // Normal Execution
    Process(workItem)
  else // Steal
    AcquireWork(k) // k = id of victim thread
```

- Allows fine-grained workload partitioning with low overhead
- Previously considered unsuitable for heterogeneous systems due to explicit copying of data to accelerators

Work Stealing for Graph Processing

```
Thread T[i](Start, End, sync)
while true do
  if Start < End then           // Normal Execution
    Process(vtx[Start])
    Start = Start + 1

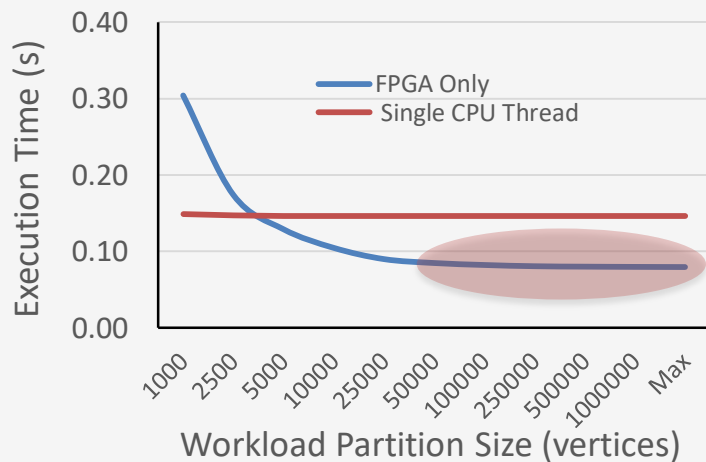
  else // Start == End; // Steal
    if CAS(T[k].sync) then // k = id of randomly chosen thread
      T[i].Start = (T[k].Start+T[k].End)/2 // Steal half
      T[i].End = T[k].End
      T[k].End = T[i].Start
```

Challenges with Heterogeneity

- Non-Linear FPGA performance with workload size
- How to steal from hardware?
- Duplicate work caused by FPGA read latency
- Hardware Limitations

Non-Linear FPGA Performance

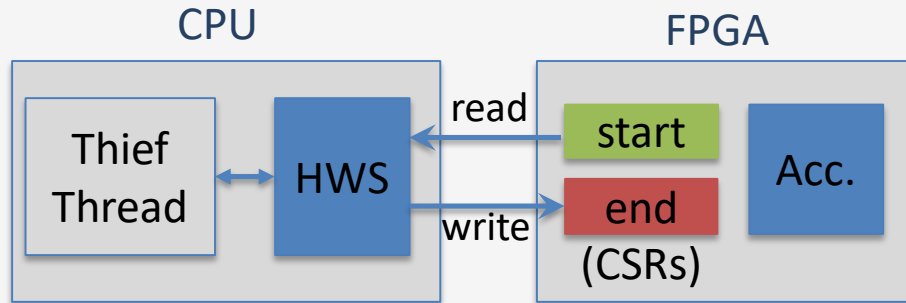
- The FPGA accelerator performance depends on the size of the workload assigned to it
 - Larger workloads better amortize accelerator startup and initial latency



- HWS assigns large enough workloads, stealing only when CPU threads idle

Steal Mechanism

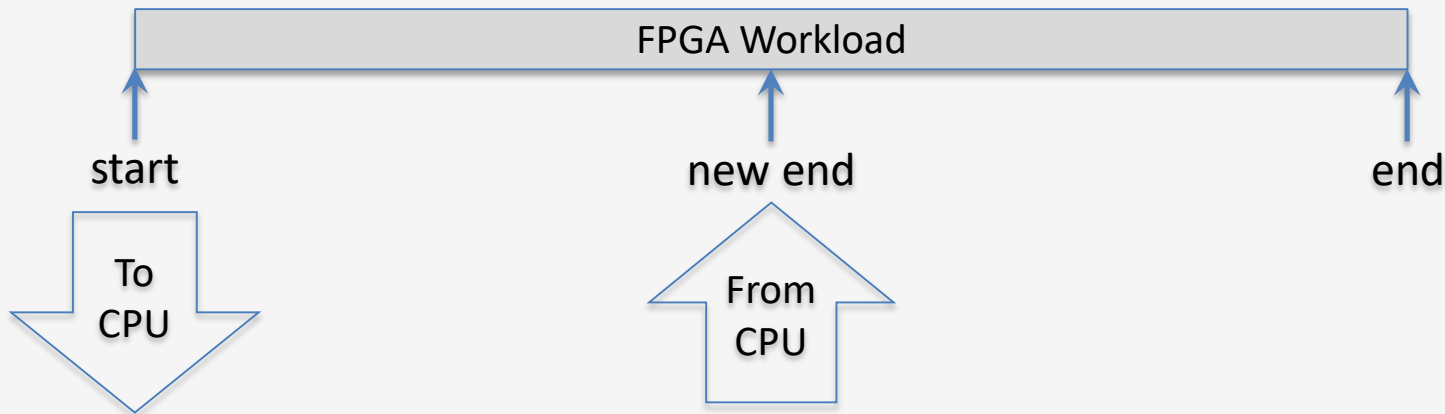
- How does software steal from hardware?
 - Internal accelerator state is often not accessible by software
- Accelerator exposes two CSRs: **start** and **end**
 - Thief thread reads **start** to determine how much to steal
 - Thief thread calculates new FPGA end bound and then writes to **end**



FPGA processing is
un-interrupted
during the steal

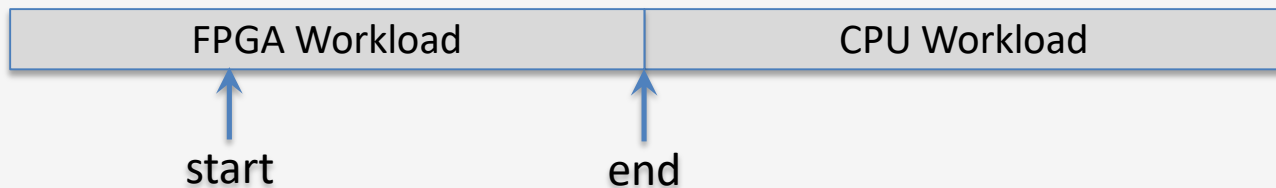
Duplicate Work

- The delay in reading CSR registers leads to potential work duplication
 - The value of the **start** register read by thief is stale



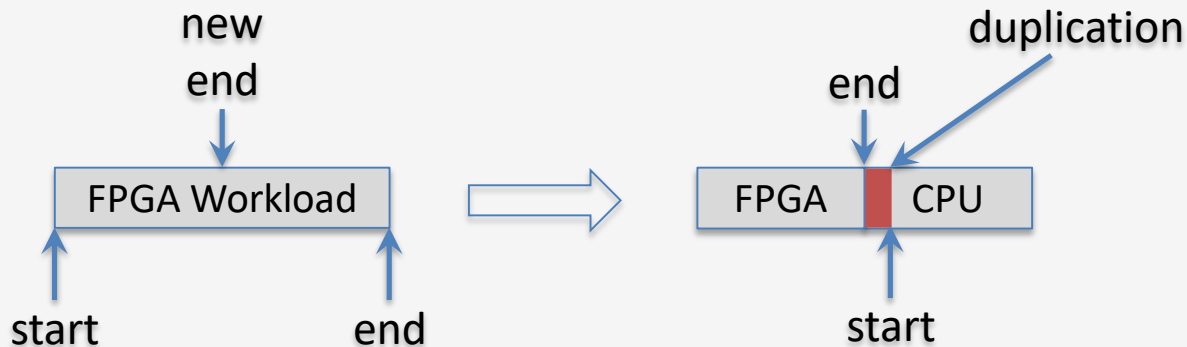
Duplicate Work

- The delay in reading CSR registers leads to potential work duplication
 - The value of the **start** register read by thief is stale



Duplicate Work

- The delay in reading CSR registers leads to potential work duplication
 - The value of the **start** register read by thief is stale
 - When a small amount of work remains, the thief may steal work already performed by the FPGA



Duplicate Work

- The delay in reading CSR registers leads to potential work duplication
 - The value of the **start** register read by thief is stale
 - When a small amount of work remains, the thief may steal work already performed by the FPGA
- We estimate FPGA progress P , ensuring that a steal from the FPGA fails if too small a workload remains
 - Enabled by the relatively deterministic nature of the accelerator

$$T[i].Start = ((T[k].Start + P) + T[k].End)/2$$

Hardware Limitations

- FPGA memory requests are aligned to cache lines
 - Misaligned requests can negatively affect performance
- HWS aligns FPGA workloads with cache lines and imposes a lower bound on stealing granularity
 - Only 8 vertices per cache line

Evaluation

- Graph Benchmarks
- Platform
- Metrics of performance
- Results
 - Load balancing effectiveness
 - Comparison to state-of-the-art
 - Steal characteristics
 - Graph processing throughput

Graph Benchmarks

- We use three common graph processing benchmarks:
 - Breadth-First Search (BFS)
 - Single Source Shortest Path (SSSP)
 - PageRank (PR)

Common benchmarks
BFS and SSSP used by Graph500
- Implemented in the *Scatter-Gather* paradigm
 - A common paradigm for graph processing
 - Scatter: sweep over vertices, producing updates to neighboring vertices
 - Gather: sweep over updates, applying them to destination vertices

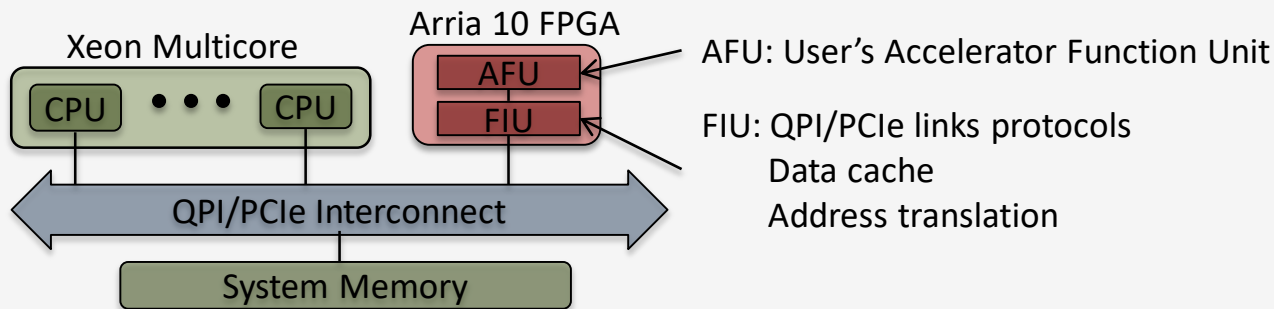
Evaluation Graphs

- Process 7 large graphs, mostly drawn from SNAP

Graph	Vertices	Edges	Description
Twitter	62M	1,468M	Follower data
LiveJournal	4.8M	69M	Friendship relations data
Orkut	3M	234M	Social connections
StackOverflow	2.6M	36M	Questions and answers
Skitter	1.7M	22M	2005 Internet topology graph
Pokec	1.6M	31M	Social connections
Higgs	460K	15M	Twitter subset

Platform

- Intel's Heterogeneous Architecture Research Platform (HARP)
 - Xeon E5-2680 v4 CPU + Arria 10 GX1150 FPGA
 - AFU issues cache coherent reads/writes to system memory



- AFUs for the *scatter phase* of graph processing [O'Brien 2020]
 - The gather phase is done by CPU threads

Performance Metrics

- **Execution time**: time for processing, excluding loading graph into memory
- **Load imbalance**: the maximum useful work time of a thread relative to the average useful work time

$$\lambda = \frac{\max(W_1..W_N)}{\text{avg}(W_1..W_N)} \quad \text{ideally, } \lambda \text{ is } 1$$

- **Throughput**: the number of traversed edges per second (MTEPS)

Comparisons

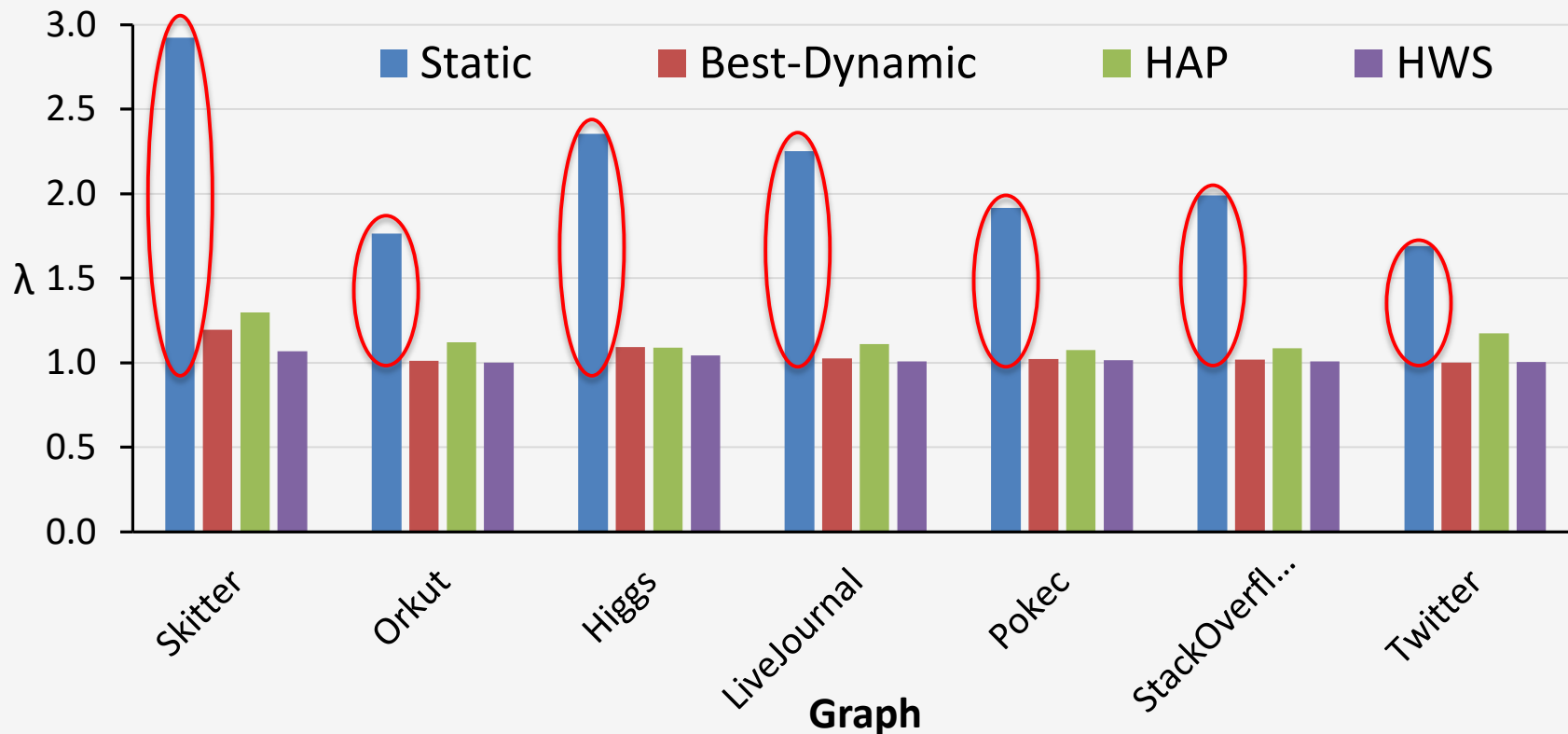
- We compare HWS to different load balancing strategies
 - **Static**: equal sized partitions to all threads, giving FPGA 2.5X more
 - **Best-Dynamic**: a chunk self-scheduling load balancer with **a priori** knowledge of the optimal chunk size
 - **HAP**: Heterogeneous Adaptive Partitioning scheduler [Rodriguez 2019]
- We define **speedup** as the ratio of the execution time of static to that of a load balancing strategy

Disclaimer

- The results in this paper were generated using pre-production hardware and software, and may not reflect the performance of production or future systems.

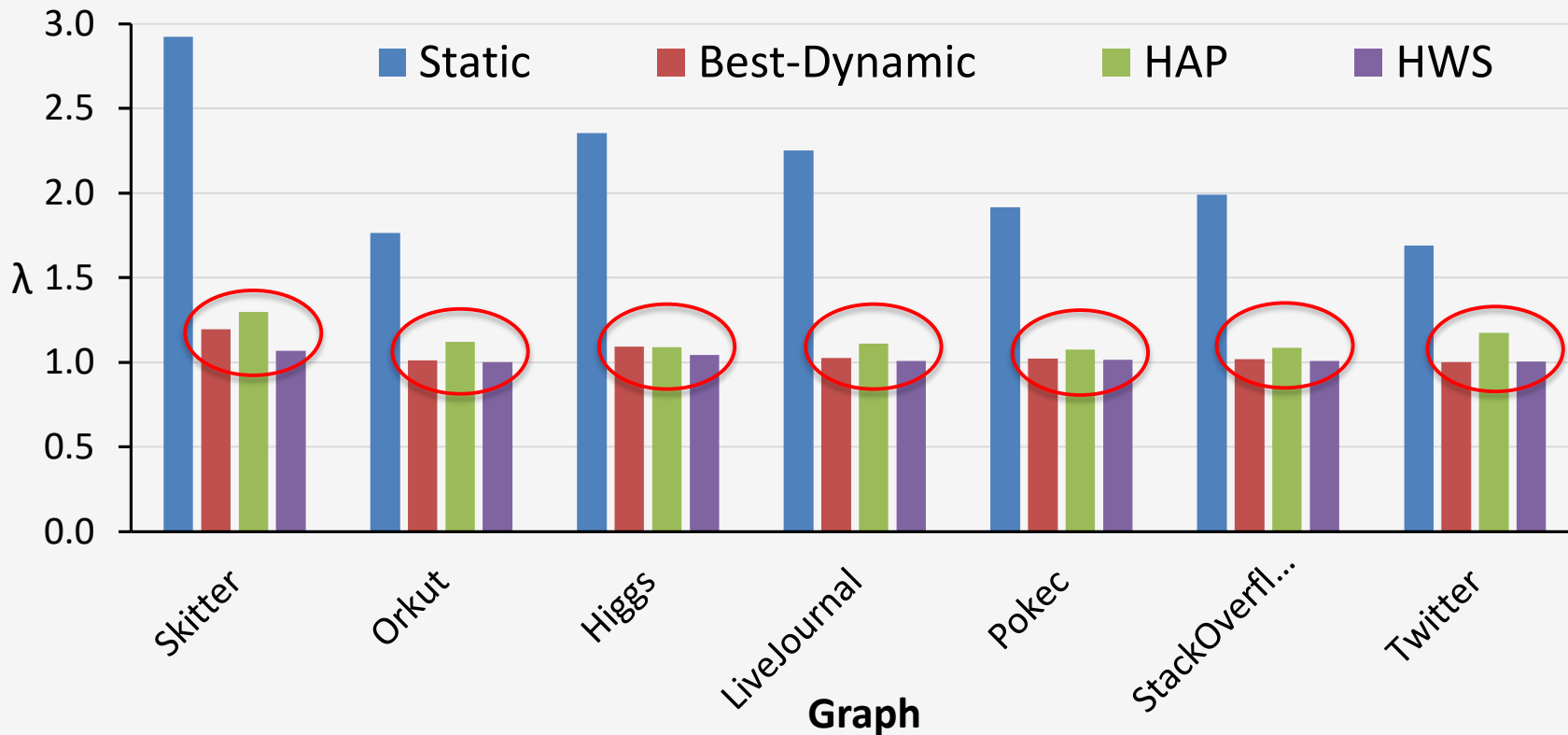
BFS Scatter λ

HARPV2 15 Threads + AFU



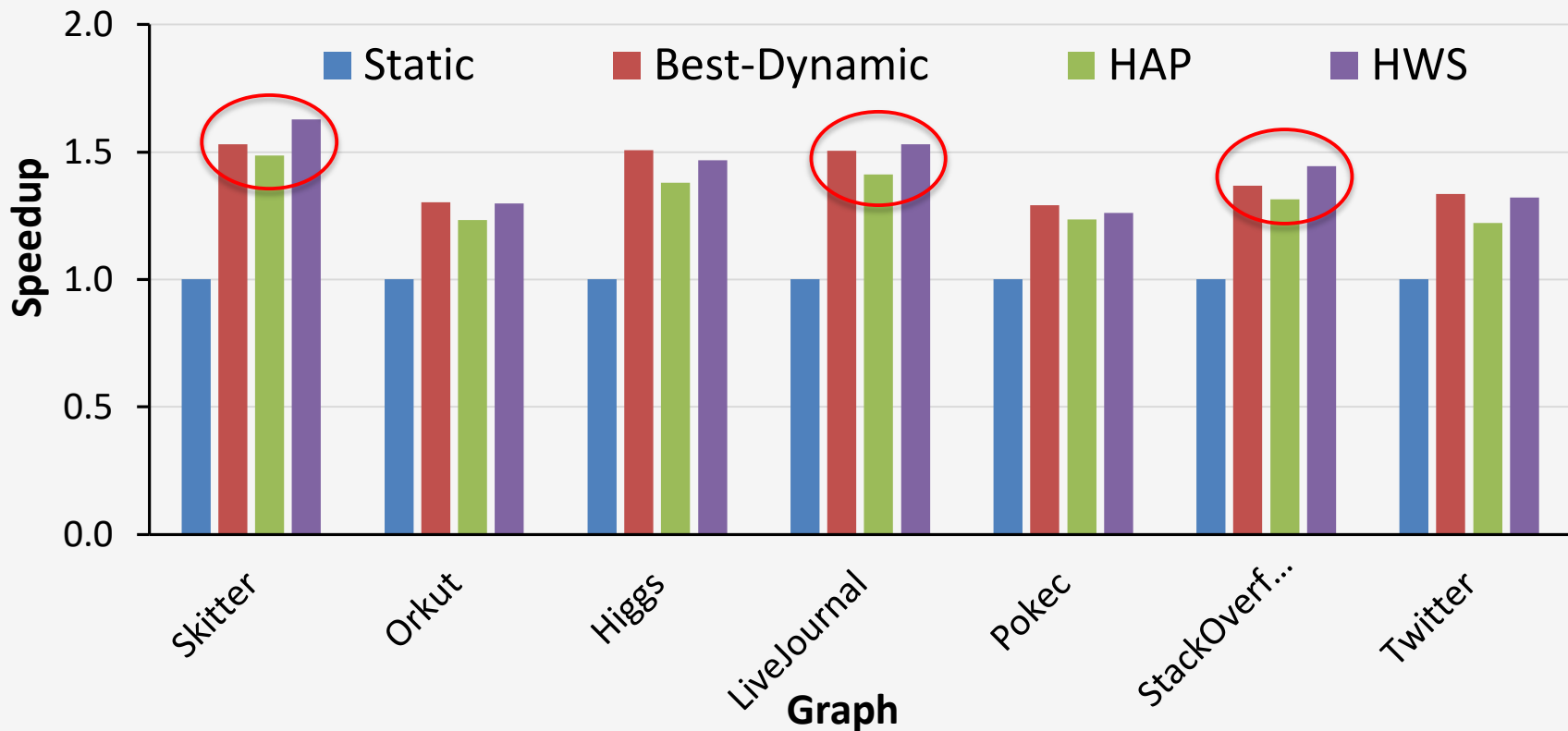
BFS Scatter λ

HARPV2 15 Threads + AFU



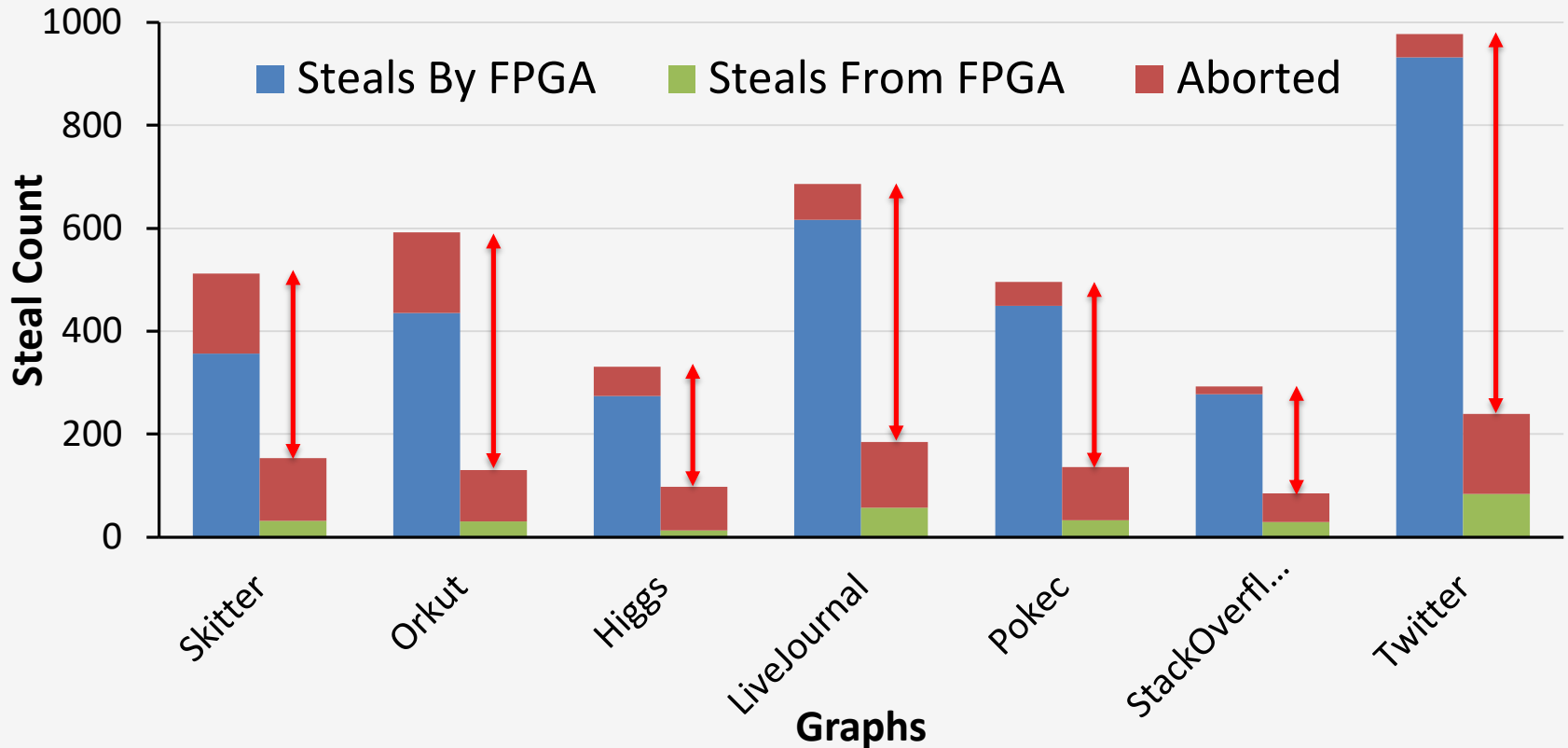
BFS Scatter Performance

HARPV2 15 Threads + AFU



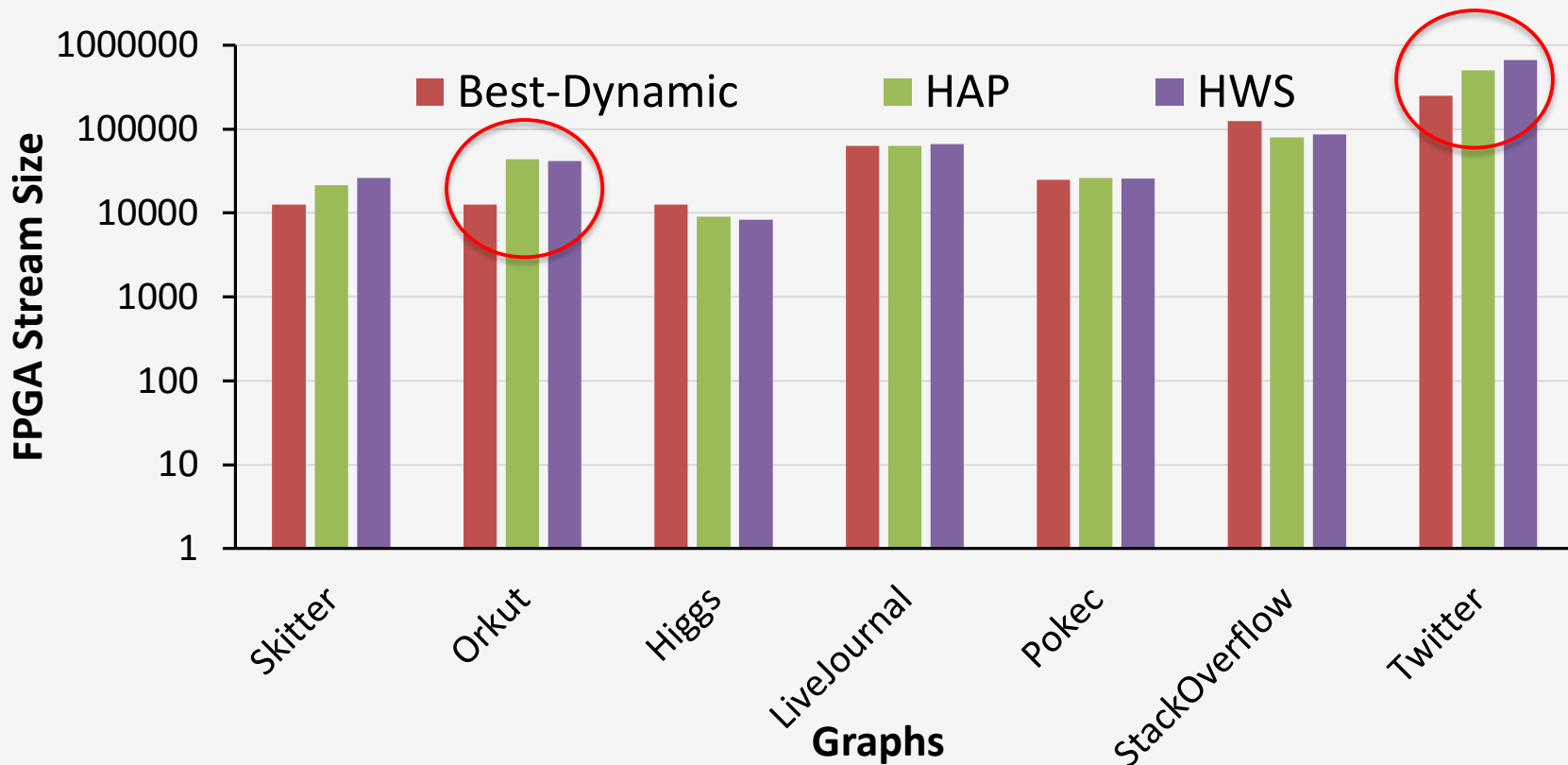
HWS Steal Characteristics

HARPV2 SSSP 7 Threads + AFU

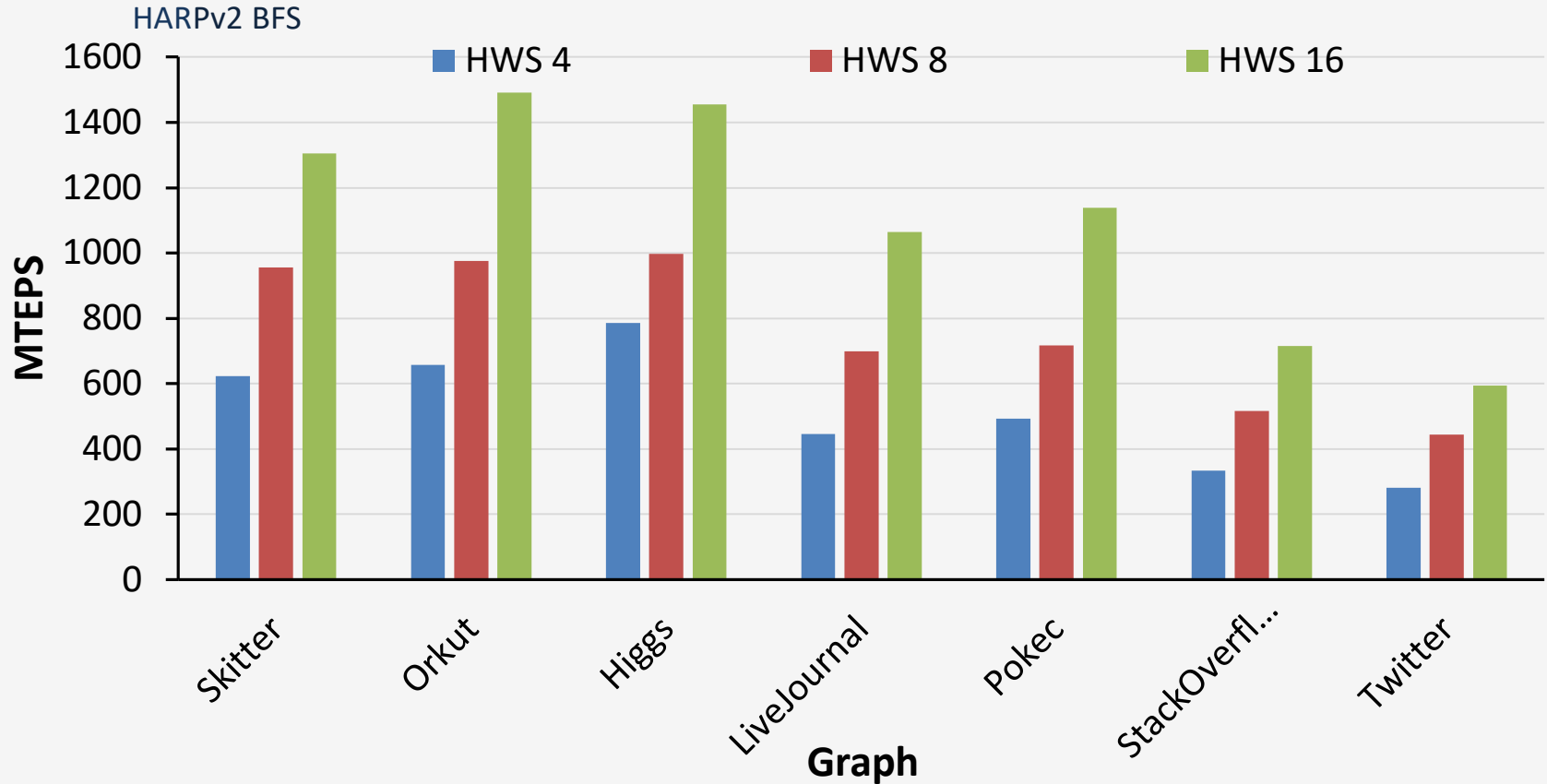


Average FPGA Chunk Size

HARv2 SSSP 7 Threads + AFU



Overall Throughput



Related Work

- Heterogeneous scheduling: Tripp (2005), Belviranli (2013), Vilches (2015), Song (2016), Navarro (2019), Rodriguez (2019), Wang (2019)
 - Focus is on adaptive chunk size selection
 - We introduce work stealing and demonstrate its effectiveness
- Work stealing: Acar (2013), Cong (2008), Dinan (2009), Hendler (2002), Khayyat (2013), Nakashima (2019)
 - We extend work stealing to heterogeneous systems
- FPGA accelerators for graph processing: Dai (2016), Engelhardt (2016), Zhou (2017), Zhou (2019)
 - We extend to concurrent CPU-FPGA usage

Concluding Remarks

- HWS addresses some unique challenges when processing graphs on CPU-FPGA systems
 - HWS maximizes FPGA throughput ensuring large workloads
 - HWS achieves perfect load balance ($\lambda = 1$)
 - HWS outperforms competitively against other schedulers
- Our results collectively demonstrate that work stealing is an effective solution for balancing graph processing workloads on tightly-coupled heterogeneous CPU-FPGA systems

Future Work

- Heterogeneous acceleration of the gather phase of graph processing
- Integration of HWS with multiple FPGAs
- Work stealing optimizations: priority-based victim selection
- Processing of dynamically changing graphs

Thank You