FLOOD: Adaptive, Accelerated Stream Processing

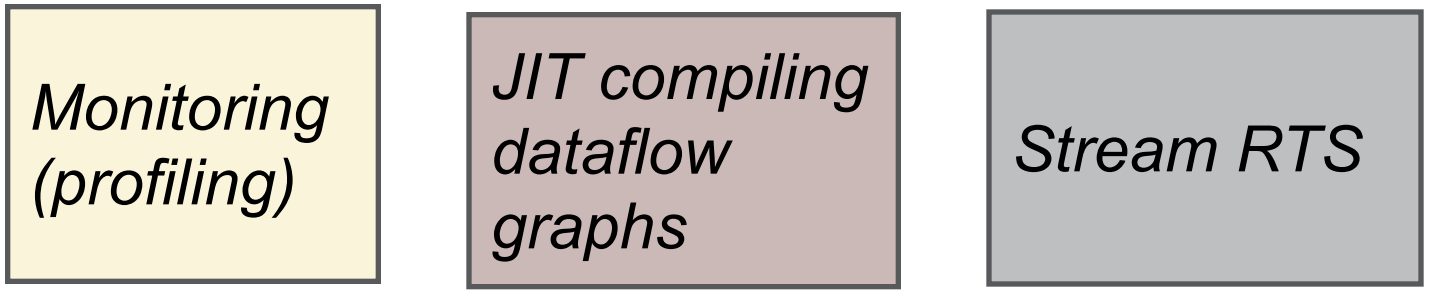
Ryan Newton

6/1/2015

INDIANA UNIVERSITY

FLOOD project

- Distributed stream-processing compiler & RTS
- Basic plan:
  - support JIT compilation and adaptation
  - use EDSL compilers for accelerators & vectorization
  - combine benefits of e.g. StreamIt & Twitter Storm

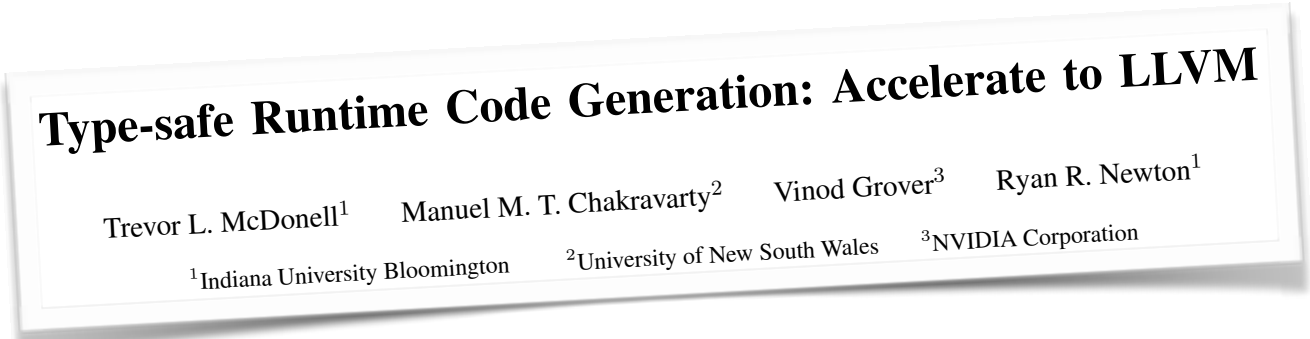


Embedded Languages: Lots of these — and growing

- Python — Copperhead, Continuum.io, Theano
- Haskell — Accelerate, Nikola, Paraiso, FeldSpar
- Scala — Delite/OptiML
- JavaScript — RiverTrail (Intel)
- LUA - Torch 7
- C++ — Intel ArBB (now defunct)
- Standalone — Harlan, Halide, StreamIt

Embedded Languages: Lots of these — and growing

- Haskell — Accelerate
  - Since 2009
  - Originally: multi-dimm array language
  - Partially formally verified

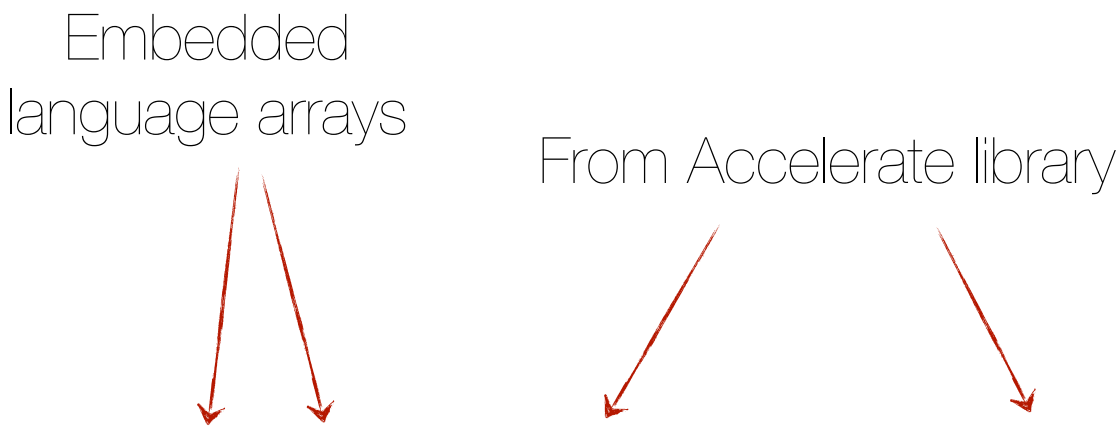


Type-safe Runtime Code Generation: Accelerate to LLVM

Trevor L. McDonell[1] Manuel M. T. Chakravarty[2] Vinod Grover[3] Ryan R. Newton[1]

[1]Indiana University Bloomington [2]University of New South Wales [3]NVIDIA Corporation

Accelerate hello world



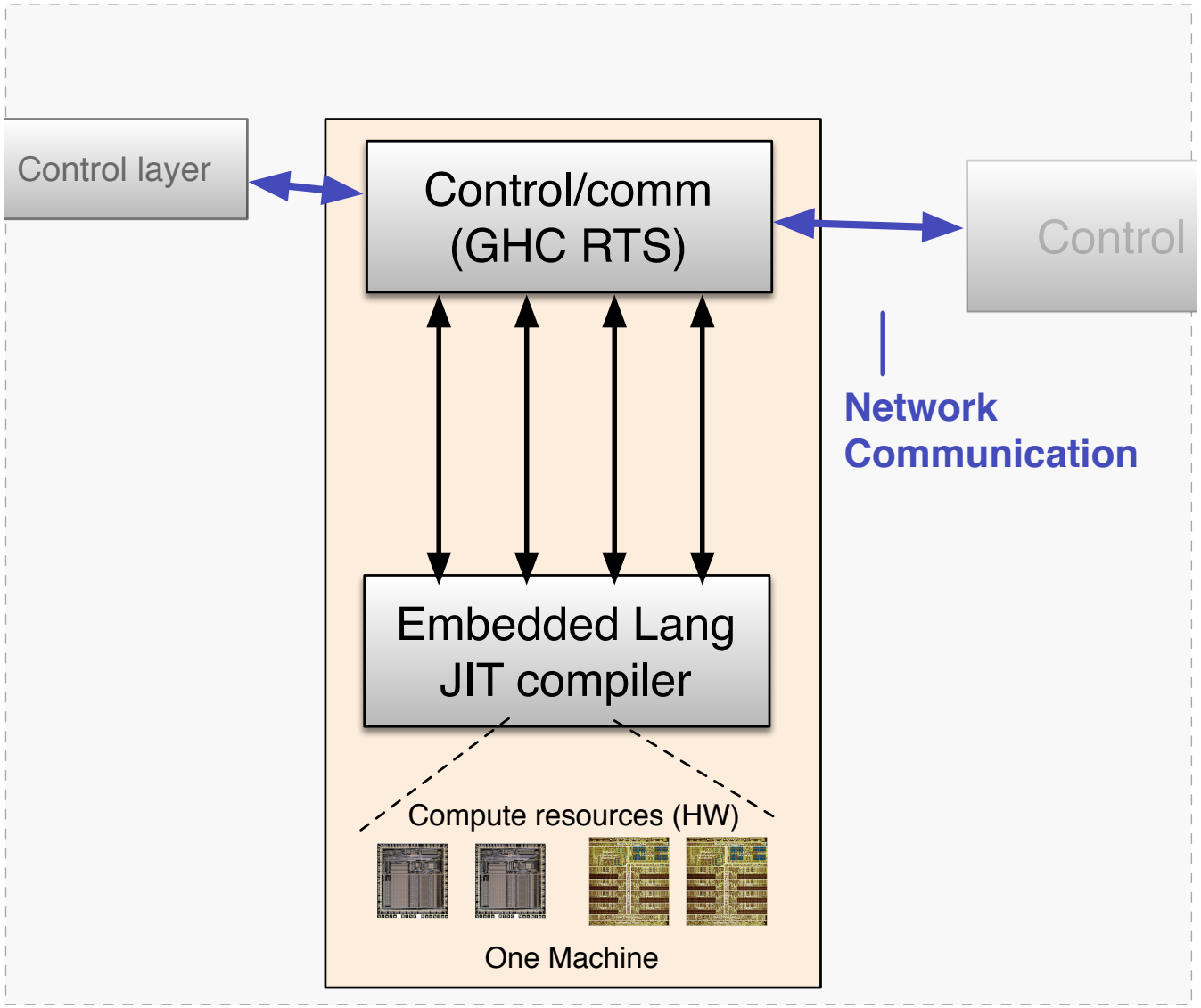

```
dotp xs ys = fold (+) 0 ( zipWith (*) xs ys )
```

Switching to and from streams

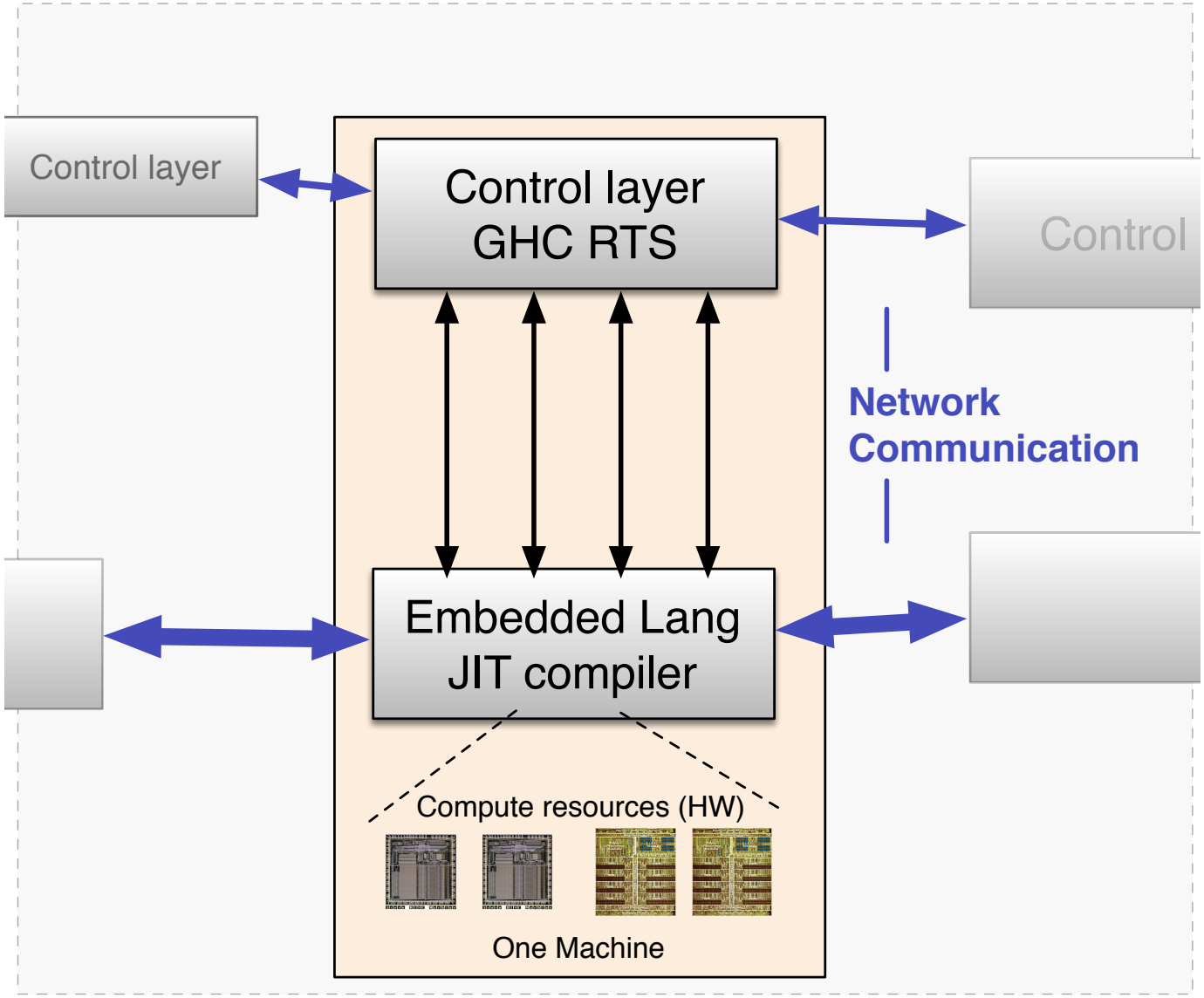
```
dotpSeq xs ys =
    collect
  $ foldSeqE (+) 0
  $ zipWithSeqE (*) (toSeqE xs) (toSeqE ys)
```

** (Streaming Accelerate - with collaborators at UNSW in Sydney)*

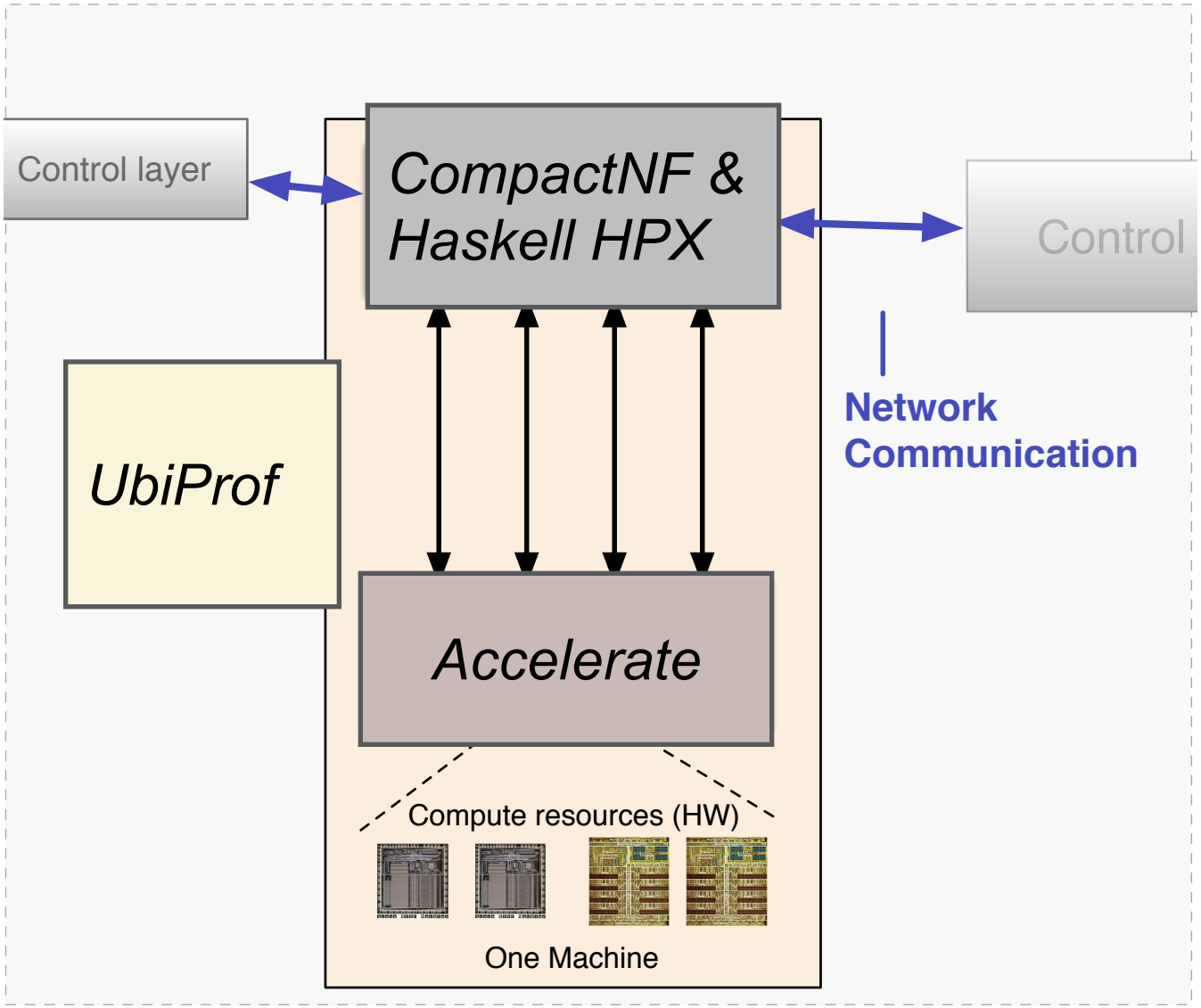
Current Architecture



Eventual Architecture

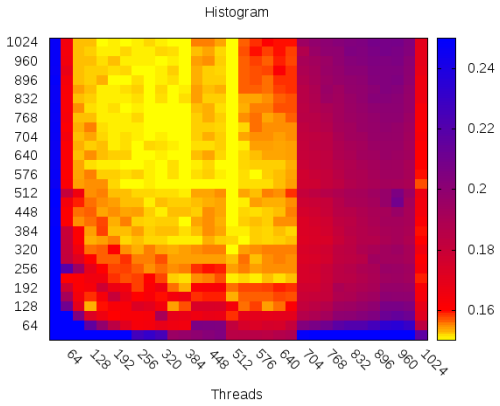


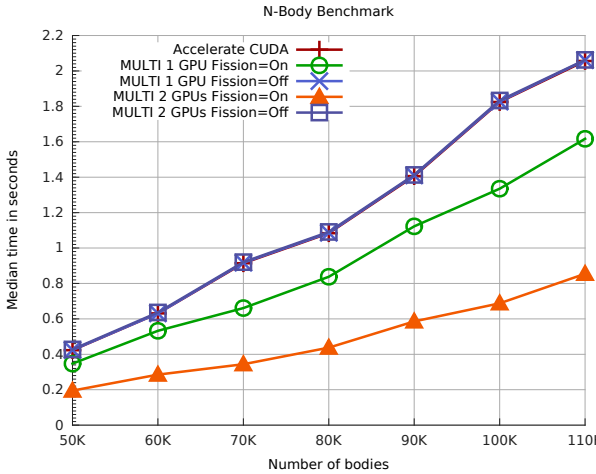
Current Architecture



Ongoing work on *adaptation*

- Multi-device - cpus/gpus
- Auto-tune
  - Params
  - Rewrite rules
  - Device-specific backend opts





$$\begin{aligned}
\text{fold } f\ e\ æ &\Rightarrow \text{let } (x, y) = \text{split}_{-1}\ æ \text{ in} \\
&\quad \text{zipWith } f\ (\text{fold } f\ e\ x)\ (\text{fold } f\ e\ y) \\
\text{map } f\ æ &\Rightarrow \text{let } (x, y) = \text{split}_i\ æ \text{ in} \\
&\quad \text{concat}_i\ (\text{map } f\ x, \text{map } f\ y) \\
\text{generate } \sigma\ f &\Rightarrow \text{concat}_i\ (\text{generate } \sigma[\sigma_i := \lfloor \sigma_i/2 \rfloor]\ f, \\
&\qquad \text{generate } \sigma[\sigma_i := \lceil \sigma_i/2 \rceil] \\
&\qquad\quad (\lambda x_0 \ldots x_n.\ f x_0 \ldots \lfloor x_i + \sigma_i/2 \rfloor \ldots x_n))
\end{aligned}$$

CompactNF

```
do c ← newCompact (buildTree x)
   sendCompact sock c
```

- (See ICFP'15)
- Puts data in a managed heap region
  - GHC uses a block structured heap
- Send via RDMA
  - relocatable if necessary
- (similar to *region based* me mgmt)

UbiProf

- Cheaper dynamic probes (v.s. DTrace)
  - Uses Intel's __notify_intrinsic
- For streaming: create a labeled graph
  - Edges: data volume
  - Vertices: compute latency
- (See poster)

Compiler & Runtime Techniques for Adaptive Stream Processing

Ryan Newton, Joel Svensson, Trevor McDonell,
Michael Vollmer, Ömer Sinan Ağacan, Buddhika Chamith

{rrnewton, joelsven, mcdonelt, vollmerm, oagacan, budkahaw}@indiana.edu

Introduction

Distributed stream processing requires a combination of technologies to monitor workloads and map them onto the local resources of worker machines. Here we highlight three sub-projects that address distinct aspects of this problem:

- ***Compile:*** JIT compiling dataflow graphs for available parallel architectures (Accelerate)
- ***Communicate:*** Sending irregular data efficiently between nodes (Compact Normal Form)
- ***Monitor:*** Profiling native-code programs based on binary self-modification and cross-modification (Ubiprof)

Accelerate

In a distributed execution plan, a subgraph of a stream dataflow graph must map onto the hardware of a worker node and achieve throughput. Our approach is a DSL JIT compiler called *Accelerate*.

- Accelerate [6] takes a graph of data-transformations and generates CUDA or LLVM code to run on CPU or GPU.
- Accelerate can launch concurrent GPU kernels on each new input (stream element) that arrives
- Accelerate is a (partially) formally verified compiler [4]

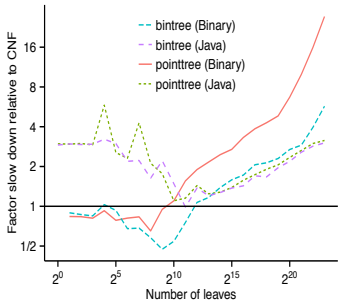
Compact Normal Form (CNF)

Distributed stream processing systems (like Twitter Storm) must route streams over network links. Data (de)serialization consumes significant time, especially for irregular and pointer-based data structures.

For immutable data in high-level languages, we explore an alternate heap representation: *Compact Normal Form* **[1]**. CNF allows regions of the heap to directly be:

- sent over the network (including RDMA)
- stored to disk
- skipped over by GC as *one* object

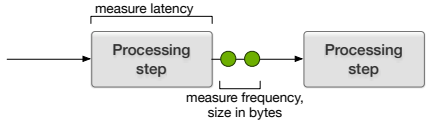
We have implemented CNF for the Glasgow Haskell Compiler (GHC). Our total speedup for sending, e.g., large binary trees through the network with CNF can exceed 16X:



Likewise, reading twitter data from disk is faster when it can be mmap'd directly into GHC's heap, whether reading one record or all of them:

UbiProf

Profiling streaming dataflow graphs for scheduling purposes has different requirements than traditional time profiling.



Capturing processing latency and data size requires sampling *intervals* rather than *instants*, and injecting custom code to compute data structure sizes. Thus dynamic probes, such as in DTrace, are appropriate.

Yet overheads for these probes are high. Thus we are exploring new approaches with an order of magnitude improvement in probe cost, and a probing infrastructure that is:

- Intra-process, user space
- Multicore scalable
- Free when not engaged

We have also used these probes to build a general profiling tool, UbiProf **[5]**. UbiProf dynamically activates and deactivates probes to stay under a given allowable overhed, using a *backoff* approach for hot functions to disable their own probes.

UbiProf controls a backoff threshold and the frequency with which deactivated probes are reactivated (sampling epoch).