Run Time Approximation of Non-blocking Service Rates for Streaming Systems

Jonathan Beard
and Roger Chamberlain

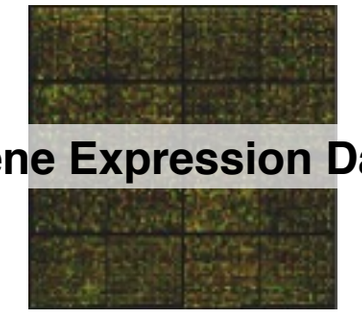
SBS
Stream Based
Supercomputing Lab
http://sbs.wustl.edu

Washington
University in St. Louis

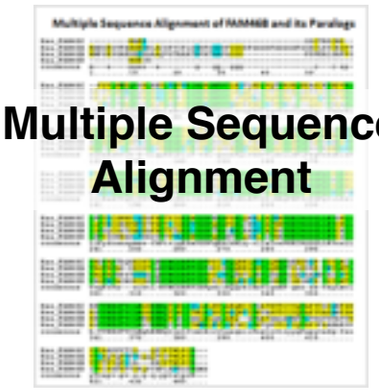
Work also supported by:

NSF exegy VELOCIDATA

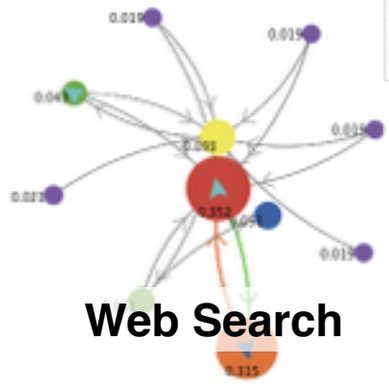
Some Big-Data Problems



Gene Expression Data



Multiple Sequence Alignment



Web Search

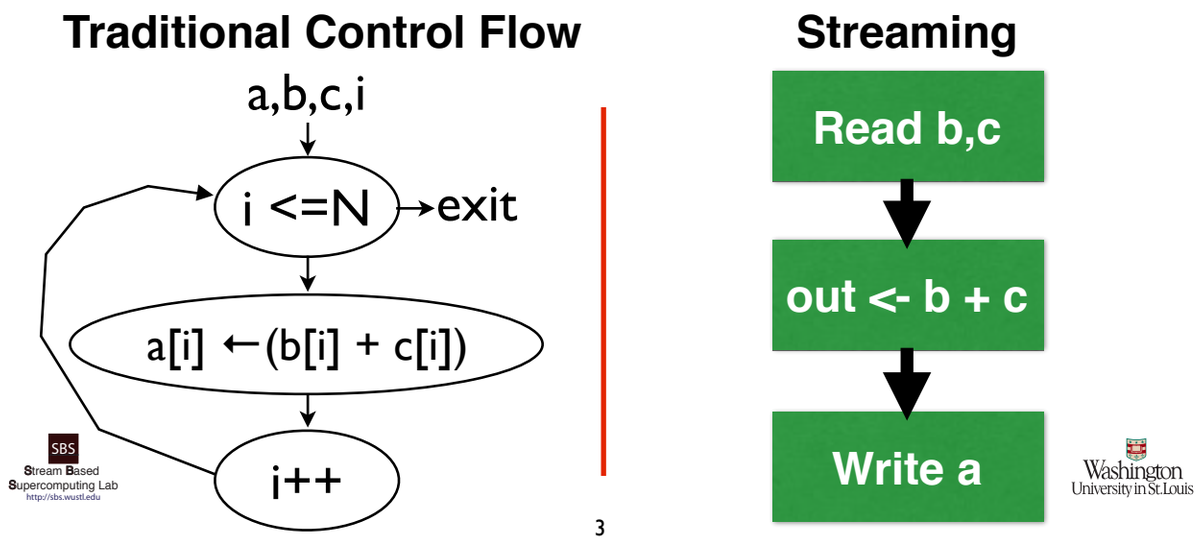
SBS
Stream Based
Supercomputing Lab
http://sbs.wustl.edu

Washington University in St. Louis

There's lots of data. Gene micro-arrays, once done completely by hand are now churned out by armies of robots. There's more sequence data than ever, they even have a USB stick for it. Of course, there's the one everyone is familiar with, web search.

Stream Processing

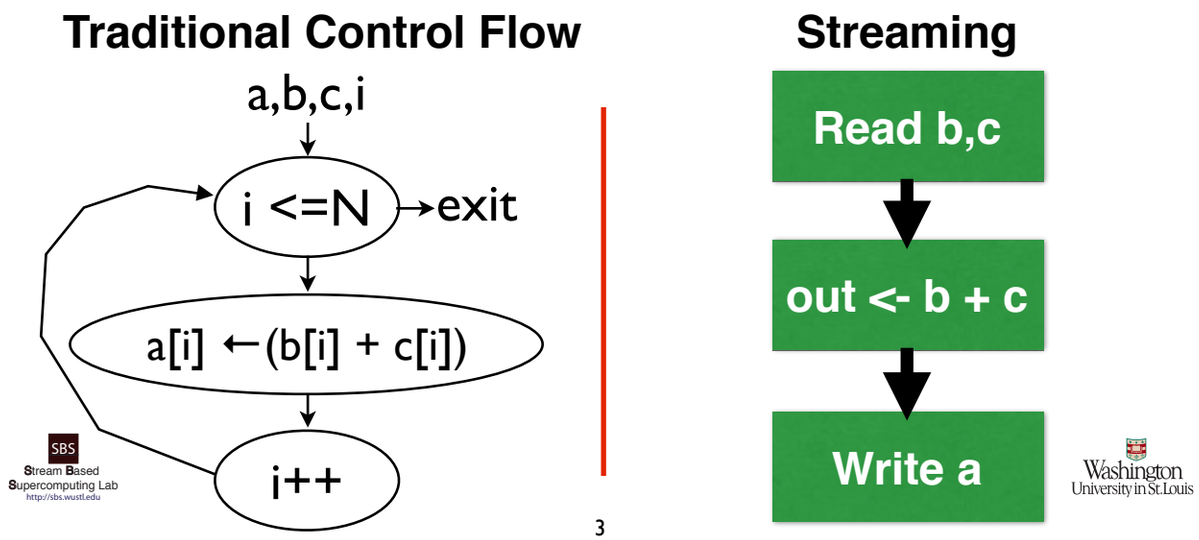
```
for i←0 through N do
   a[i] ←(b[i] + c[i])
   i++
end do
```



As a simple example, lets look at the algorithm in green above. Its a simple for loop that takes two elements from an array, adds them together, divides the sum by two and then assigns the result to the corresponding index in the third array. For a load/store architecture this loop is fairly efficient, but imagine how much simpler it can be with a data-flow architecture. We begin looking at each operation as a function connected by FIFO queues transmitting data between them. At right we can see one function (read) which supplies data, an add function which sends the sum of b and c, and a write function. Conceptually this allows pipelining of the application (each “function” can execute in parallel as soon as data is available to each), it also provides an easy way to expose instruction level parallelism that can be exploited on Load/Store architectures and in hardware (i.e. the b+c can be performed on as many elements as we have available for each firing of our kernel, or in a load store the limit is currently 8 32-bit elements at a time). If we don’t care about the order, we can also perform the add in parallel as well (multiple adds at the same time so that we can have more than three threads of execution concurrent).

Stream Processing

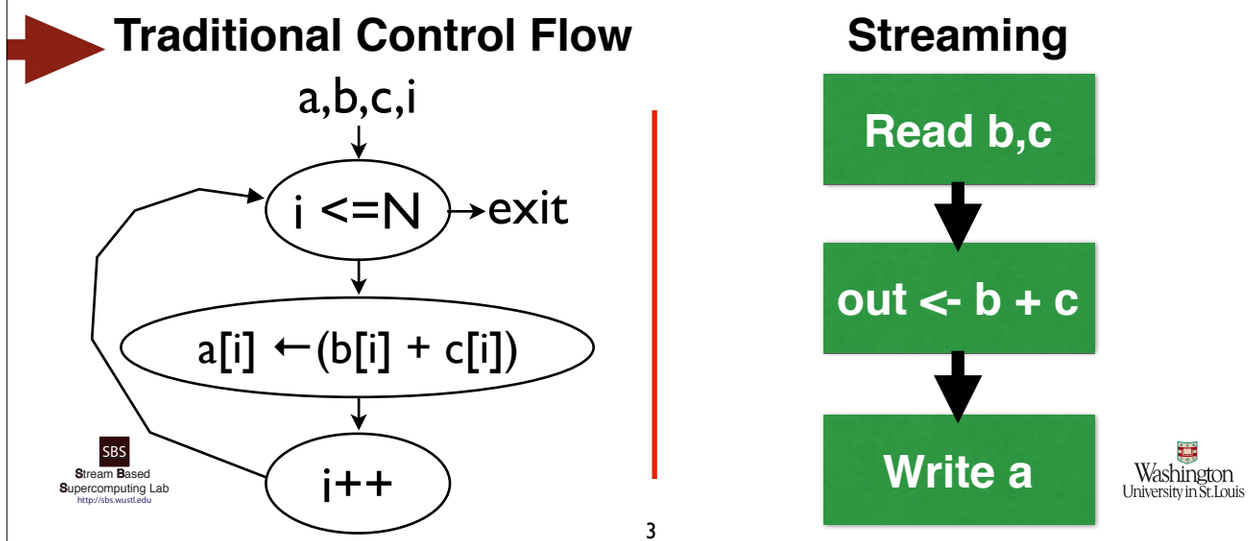
```
for i←0 through N do
   a[i] ←(b[i] + c[i])
   i++
end do
```



As a simple example, lets look at the algorithm in green above. Its a simple for loop that takes two elements from an array, adds them together, divides the sum by two and then assigns the result to the corresponding index in the third array. For a load/store architecture this loop is fairly efficient, but imagine how much simpler it can be with a data-flow architecture. We begin looking at each operation as a function connected by FIFO queues transmitting data between them. At right we can see one function (read) which supplies data, an add function which sends the sum of b and c, and a write function. Conceptually this allows pipelining of the application (each “function” can execute in parallel as soon as data is available to each), it also provides an easy way to expose instruction level parallelism that can be exploited on Load/Store architectures and in hardware (i.e. the b+c can be performed on as many elements as we have available for each firing of our kernel, or in a load store the limit is currently 8 32-bit elements at a time). If we don’t care about the order, we can also perform the add in parallel as well (multiple adds at the same time so that we can have more than three threads of execution concurrent).

Stream Processing

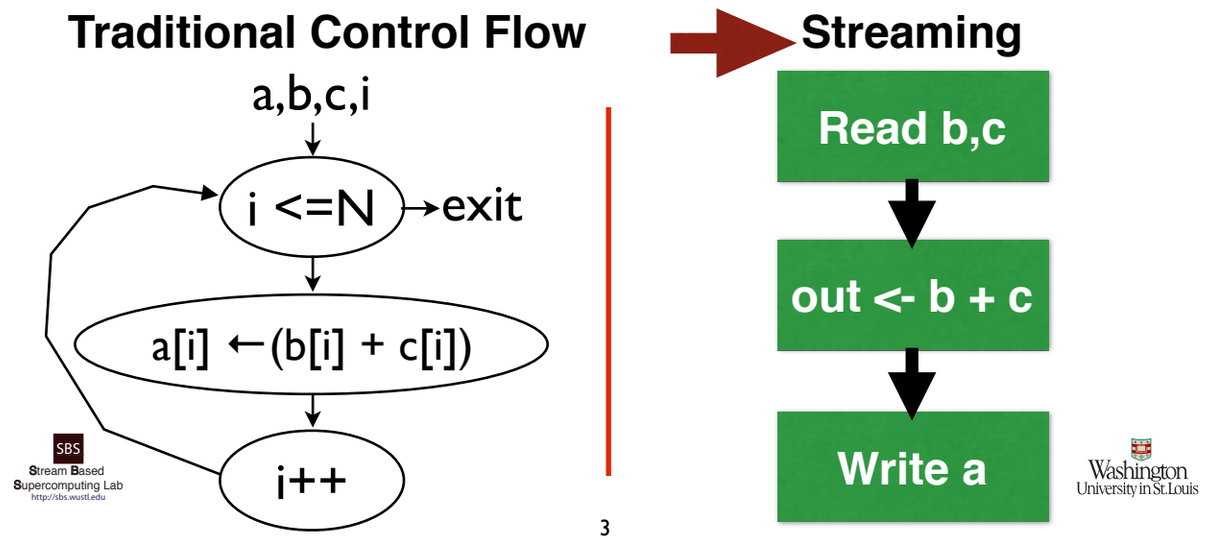
```
for i←0 through N do
   a[i] ←(b[i] + c[i])
   i++
end do
```



As a simple example, lets look at the algorithm in green above. Its a simple for loop that takes two elements from an array, adds them together, divides the sum by two and then assigns the result to the corresponding index in the third array. For a load/store architecture this loop is fairly efficient, but imagine how much simpler it can be with a data-flow architecture. We begin looking at each operation as a function connected by FIFO queues transmitting data between them. At right we can see one function (read) which supplies data, an add function which sends the sum of b and c, and a write function. Conceptually this allows pipelining of the application (each “function” can execute in parallel as soon as data is available to each), it also provides an easy way to expose instruction level parallelism that can be exploited on Load/Store architectures and in hardware (i.e. the b+c can be performed on as many elements as we have available for each firing of our kernel, or in a load store the limit is currently 8 32-bit elements at a time). If we don’t care about the order, we can also perform the add in parallel as well (multiple adds at the same time so that we can have more than three threads of execution concurrent).

Stream Processing

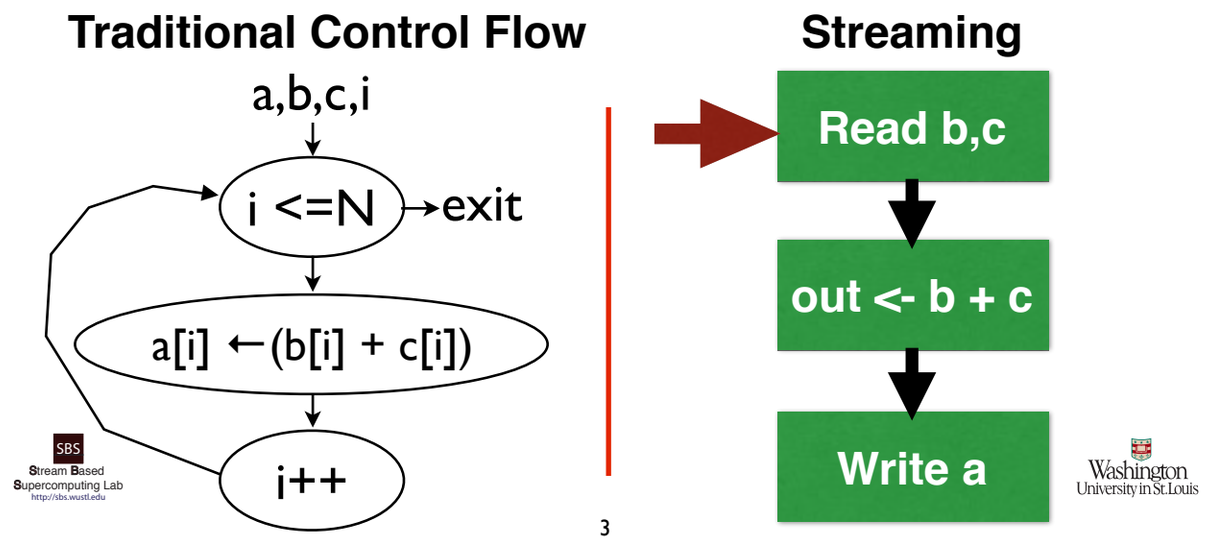
```
for i←0 through N do
   a[i] ←(b[i] + c[i])
   i++
end do
```



As a simple example, lets look at the algorithm in green above. Its a simple for loop that takes two elements from an array, adds them together, divides the sum by two and then assigns the result to the corresponding index in the third array. For a load/store architecture this loop is fairly efficient, but imagine how much simpler it can be with a data-flow architecture. We begin looking at each operation as a function connected by FIFO queues transmitting data between them. At right we can see one function (read) which supplies data, an add function which sends the sum of b and c, and a write function. Conceptually this allows pipelining of the application (each “function” can execute in parallel as soon as data is available to each), it also provides an easy way to expose instruction level parallelism that can be exploited on Load/Store architectures and in hardware (i.e. the b+c can be performed on as many elements as we have available for each firing of our kernel, or in a load store the limit is currently 8 32-bit elements at a time). If we don’t care about the order, we can also perform the add in parallel as well (multiple adds at the same time so that we can have more than three threads of execution concurrent).

Stream Processing

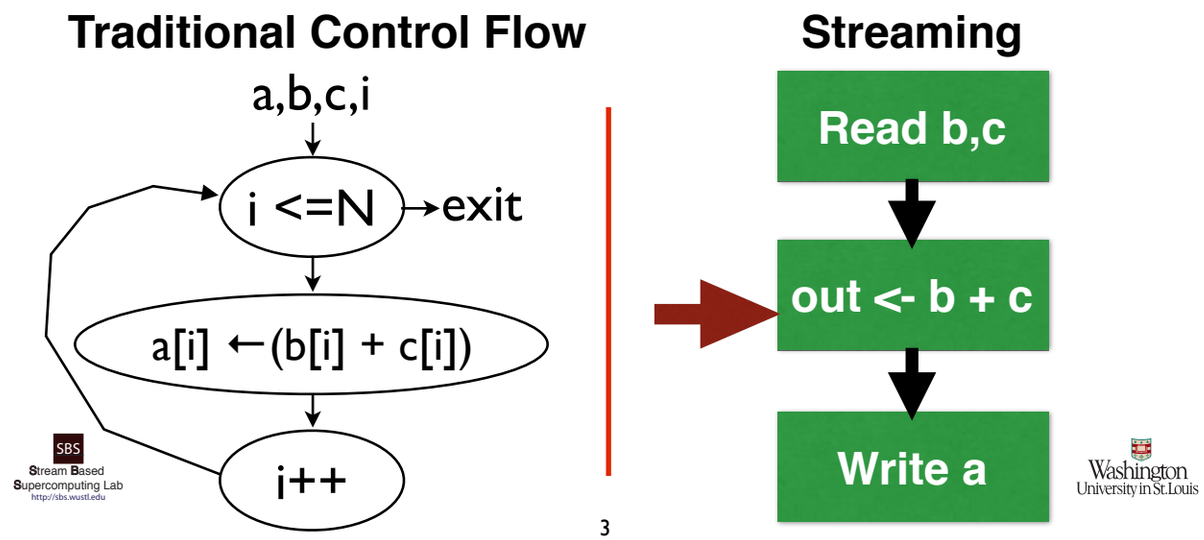
```
for i←0 through N do
   a[i] ←(b[i] + c[i])
   i++
end do
```



As a simple example, lets look at the algorithm in green above. Its a simple for loop that takes two elements from an array, adds them together, divides the sum by two and then assigns the result to the corresponding index in the third array. For a load/store architecture this loop is fairly efficient, but imagine how much simpler it can be with a data-flow architecture. We begin looking at each operation as a function connected by FIFO queues transmitting data between them. At right we can see one function (read) which supplies data, an add function which sends the sum of b and c, and a write function. Conceptually this allows pipelining of the application (each “function” can execute in parallel as soon as data is available to each), it also provides an easy way to expose instruction level parallelism that can be exploited on Load/Store architectures and in hardware (i.e. the b+c can be performed on as many elements as we have available for each firing of our kernel, or in a load store the limit is currently 8 32-bit elements at a time). If we don’t care about the order, we can also perform the add in parallel as well (multiple adds at the same time so that we can have more than three threads of execution concurrent).

Stream Processing

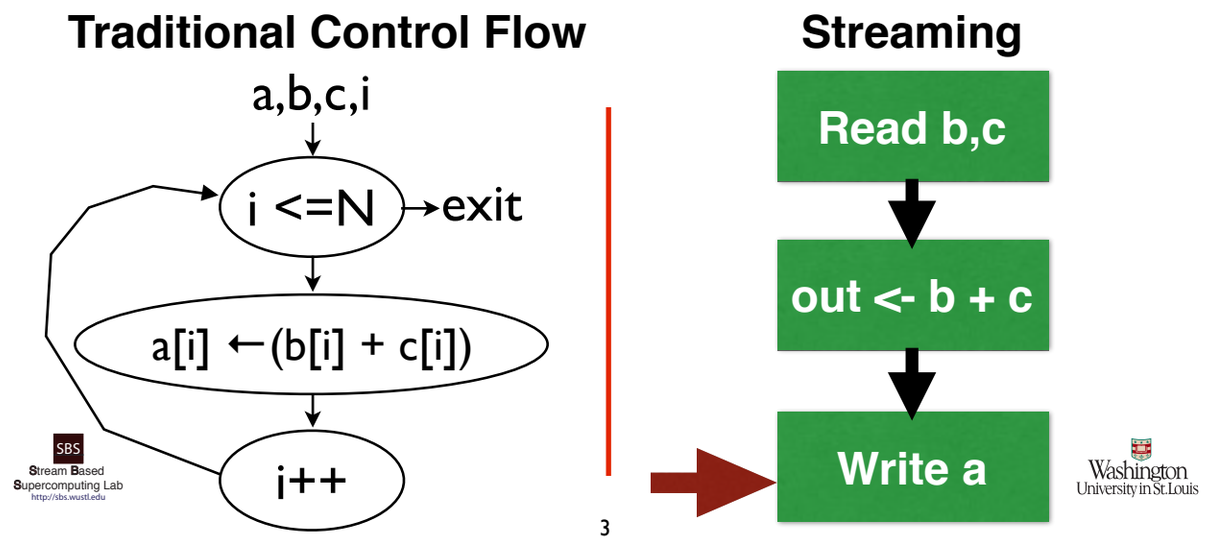
```
for i←0 through N do
   a[i] ←(b[i] + c[i])
   i++
end do
```



As a simple example, lets look at the algorithm in green above. Its a simple for loop that takes two elements from an array, adds them together, divides the sum by two and then assigns the result to the corresponding index in the third array. For a load/store architecture this loop is fairly efficient, but imagine how much simpler it can be with a data-flow architecture. We begin looking at each operation as a function connected by FIFO queues transmitting data between them. At right we can see one function (read) which supplies data, an add function which sends the sum of b and c, and a write function. Conceptually this allows pipelining of the application (each “function” can execute in parallel as soon as data is available to each), it also provides an easy way to expose instruction level parallelism that can be exploited on Load/Store architectures and in hardware (i.e. the b+c can be performed on as many elements as we have available for each firing of our kernel, or in a load store the limit is currently 8 32-bit elements at a time). If we don’t care about the order, we can also perform the add in parallel as well (multiple adds at the same time so that we can have more than three threads of execution concurrent).

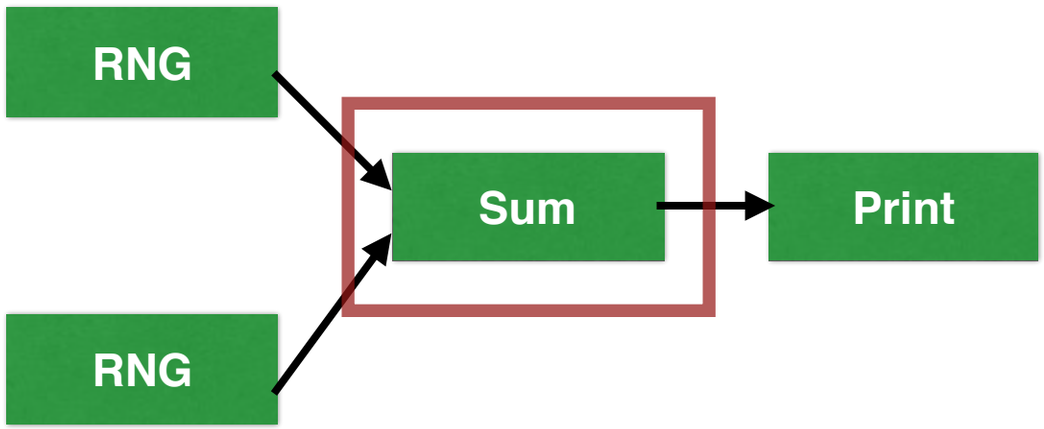
Stream Processing

```
for i←0 through N do
   a[i] ←(b[i] + c[i])
   i++
end do
```



As a simple example, lets look at the algorithm in green above. Its a simple for loop that takes two elements from an array, adds them together, divides the sum by two and then assigns the result to the corresponding index in the third array. For a load/store architecture this loop is fairly efficient, but imagine how much simpler it can be with a data-flow architecture. We begin looking at each operation as a function connected by FIFO queues transmitting data between them. At right we can see one function (read) which supplies data, an add function which sends the sum of b and c, and a write function. Conceptually this allows pipelining of the application (each “function” can execute in parallel as soon as data is available to each), it also provides an easy way to expose instruction level parallelism that can be exploited on Load/Store architectures and in hardware (i.e. the b+c can be performed on as many elements as we have available for each firing of our kernel, or in a load store the limit is currently 8 32-bit elements at a time). If we don’t care about the order, we can also perform the add in parallel as well (multiple adds at the same time so that we can have more than three threads of execution concurrent).

RaftLib Example



To make the rest of the work more concrete, we'll describe briefly the data-flow / streaming framework RaftLib which is used for all of our experiments. We'll start by talking about this simple "sum" application which was first used as an example of a data flow application by Dennis (doi: 10.1007/3-540-06859-7_145).

Stream Processing

```
template< typename A,
          typename B,
          typename C > class sum : public raft::kernel
{
public:
   sum() : raft::kernel()
   {
      input.addPort< A >( "input_a" );
      input.addPort< B >( "input_b" );
      output.addPort< C >( "sum" );
   }

   virtual raft::kstatus run()
   {
      /** look at mem on head of queue for a & b, no copy **/
      auto a( input[ "input_a" ].pop_s< A >() );
      auto b( input[ "input_b" ].pop_s< B >() );
      /** allocate mem directly on queue **/
      auto c( output[ "sum" ].allocate_s< C >() );
      (*c) = (*a) + (*b);
      /** mem automatically freed upon scope exit **/
      return( raft::proceed );
   }

};
```

The constructor (there are more efficient ways to declare ports, these used for clarity) declares two input ports “input_a” and “input_b,” and one output port “sum.” The second function “run()” is the worker which is called by the scheduler. It takes data from two input ports when it is available and pops an item from each input port and writes the sum to the output port. The return value indicates that nothing has happened to warrant exiting the program, although the program will exit on its own with it is provable that there is no further input available.

Stream Processing

```
template< typename A,
          typename B,
          typename C > class sum : public raft::kernel
{
public:
   sum() : raft::kernel()
   {
      input.addPort< A >( "input_a" );
      input.addPort< B >( "input_b" );
      output.addPort< C >( "sum" );
   }

   virtual raft::kstatus run()
   {
      /** look at mem on head of queue for a & b, no copy **/
      auto a( input[ "input_a" ].pop_s< A >() );
      auto b( input[ "input_b" ].pop_s< B >() );
      /** allocate mem directly on queue **/
      auto c( output[ "sum" ].allocate_s< C >() );
      (*c) = (*a) + (*b);
      /** mem automatically freed upon scope exit **/
      return( raft::proceed );
   }

};
```

The constructor (there are more efficient ways to declare ports, these used for clarity) declares two input ports “input_a” and “input_b,” and one output port “sum.” The second function “run()” is the worker which is called by the scheduler. It takes data from two input ports when it is available and pops an item from each input port and writes the sum to the output port. The return value indicates that nothing has happened to warrant exiting the program, although the program will exit on its own with it is provable that there is no further input available.

Stream Processing

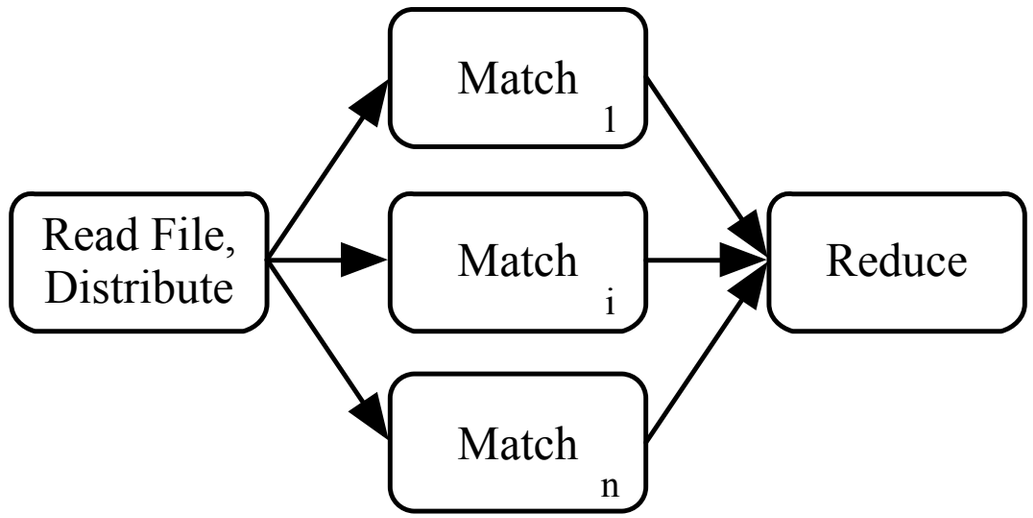
```
template< typename A,
          typename B,
          typename C > class sum : public raft::kernel
{
public:
   sum() : raft::kernel()
   {
      input.addPort< A >( "input_a" );
      input.addPort< B >( "input_b" );
      output.addPort< C >( "sum" );
   }

   virtual raft::kstatus run()
   {
      /** look at mem on head of queue for a & b, no copy **/
      auto a( input[ "input_a" ].pop_s< A >() );
      auto b( input[ "input_b" ].pop_s< B >() );
      /** allocate mem directly on queue **/
      auto c( output[ "sum" ].allocate_s< C >() );
      (*c) = (*a) + (*b);
      /** mem automatically freed upon scope exit **/
      return( raft::proceed );
   }

};
```

The constructor (there are more efficient ways to declare ports, these used for clarity) declares two input ports “input_a” and “input_b,” and one output port “sum.” The second function “run()” is the worker which is called by the scheduler. It takes data from two input ports when it is available and pops an item from each input port and writes the sum to the output port. The return value indicates that nothing has happened to warrant exiting the program, although the program will exit on its own with it is provable that there is no further input available.

RaftLib String Search



SBS
Stream Based
Supercomputing Lab
http://sbs.wustl.edu

Washington
University in St.Louis

Example of Boyer-Moore string search topology

RaftLib String Search

```
3 using strsearch = raft::search< raft::ahocorasick >;
4 std::vector< hit_t > total_hits;
5 auto kern_start( map.link< raft::out >( kernel::make< filereader >( file, offset ),
6                                         kernel::make< strsearch >( search_term ) ) );
7 map.link< raft::out >( &kern_start.getDst(),
8                         kernel::make<
9                            write_each< match_t > >(
10                              std::back_inserter( total_hits ) ) );
```

SBS
Stream Based Supercomputing Lab
http://sbs.wustl.edu

Washington University in St. Louis

implementation of the aho-corasick as a string searching library

RaftLib String Search

```
3  using strsearch = raft::search< raft::ahocorasick >;
4  std::vector< hit_t > total_hits;
5  auto kern_start( map.link< raft::out >( kernel::make< filereader >( file, offset ),
6                                          kernel::make< strsearch >( search_term ) ) )
7  map.link< raft::out >( &kern_start.getDst(),
8                         kernel::make<
9                            write_each< match_t > >(
10                              std::back_inserter( total_hits ) ) );
```

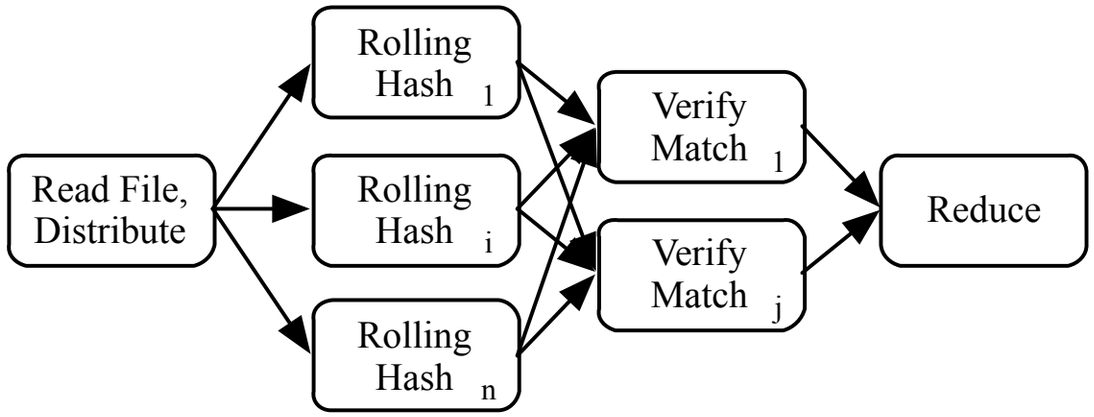

implementation of the aho-corasick as a string searching library

RaftLib String Search

```
3 using strsearch = raft::search< raft::ahocorasick >;
4 std::vector< hit_t > total_hits;
5 auto kern_start( map.link< raft::out >( kernel::make< filereader >( file, offset ),
6                                         kernel::make< strsearch >( search_term ) ) );
7 map.link< raft::out >( &kern_start.getDst(),
8                         kernel::make<
9                            write_each< match_t > >(
0                               std::back_inserter( total_hits ) ) );
```


implementation of the aho-corasick as a string searching library

Not Just Simple Split / Join



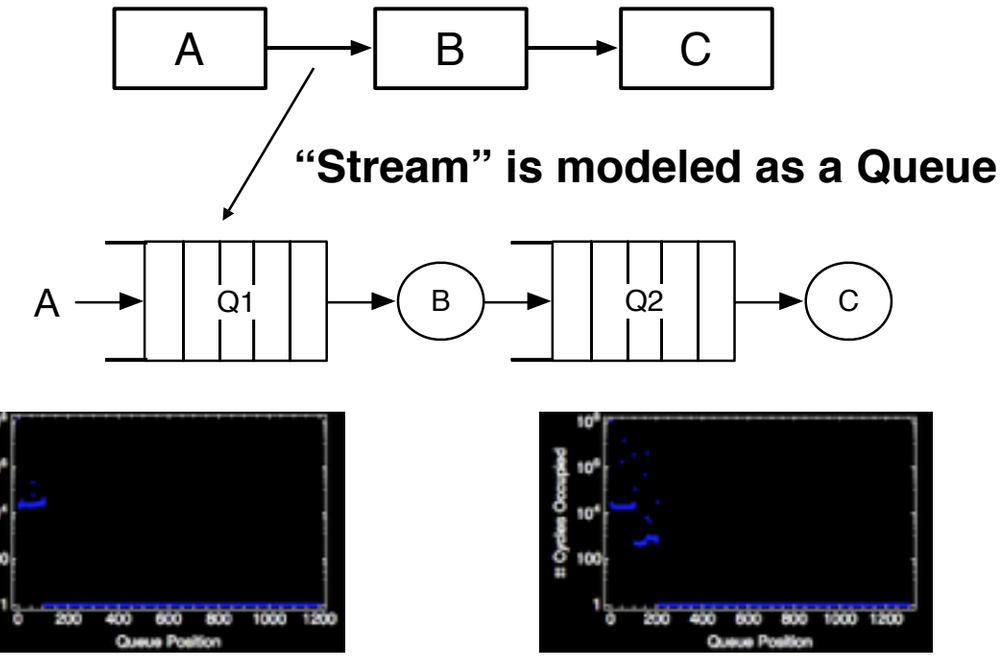
Rabin-Karp String Search

SBS
Stream Based Supercomputing Lab
http://sbs.wustl.edu

Washington University in St.Louis

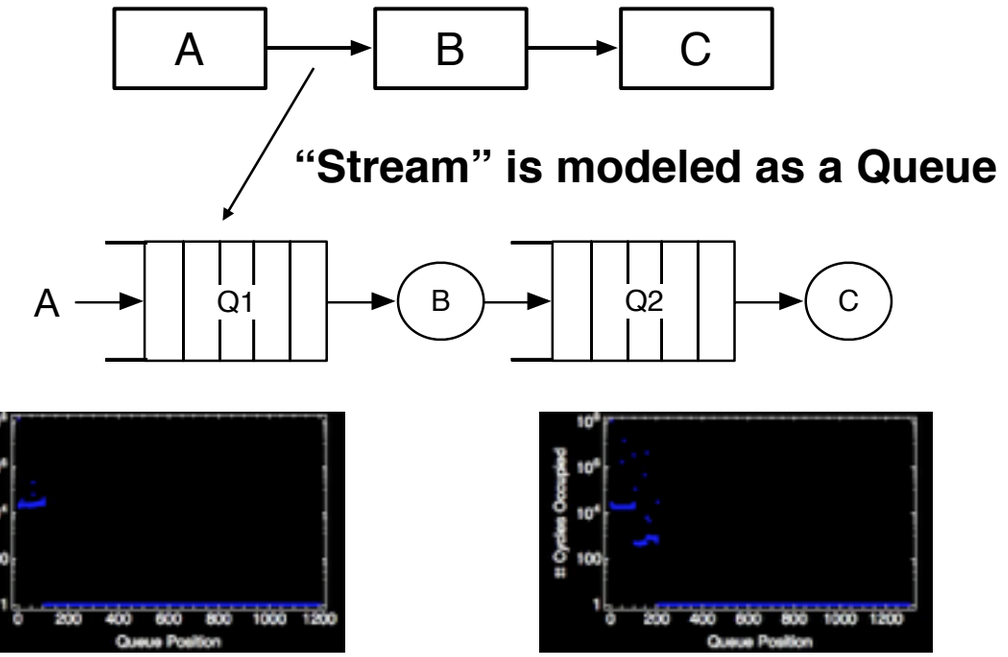
We can make all kinds of pipeline/task parallel topologies without explicit split / join. This is the strength of stream processing in that we break the mold of the fork/join model. This lack of explicit synchronization gives stream processing a unique ability to exploit extreme levels of parallelism.

Modeling Streams as Queues



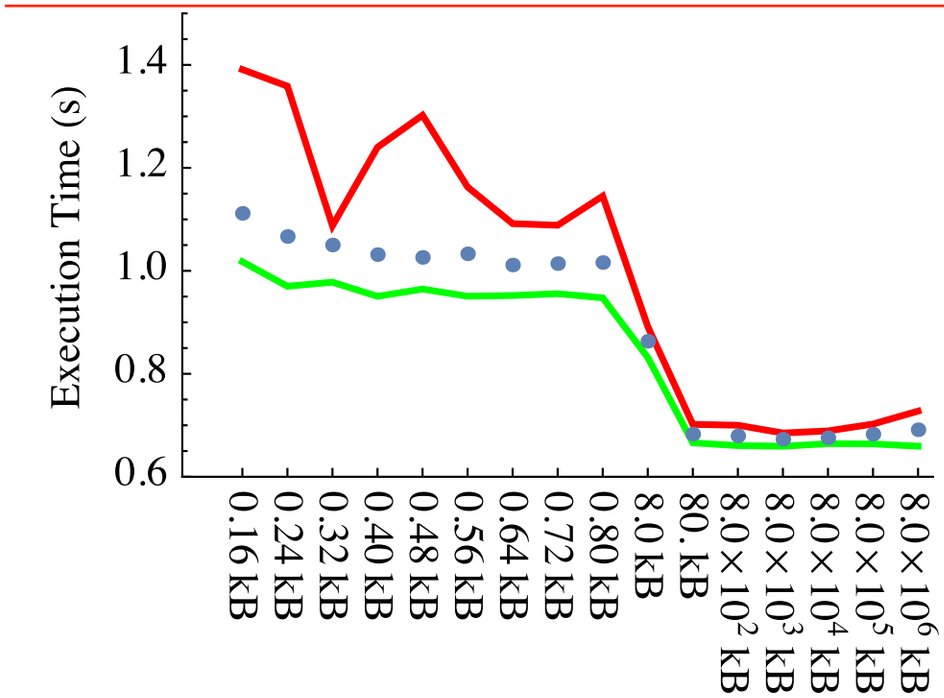
At the top there's an example of a simple streaming application. Each stream can be modeled as a queue. At bottom is an example of the queue activity of our streaming application example. The x-axis is the queue position and the y-axis represents the # of cycles occupied within each time frame. The front that is stable is what we're interested in, can we find this quickly? One way to do that s to use queueing models, but they require some idea of service rate.

Modeling Streams as Queues



At the top there's an example of a simple streaming application. Each stream can be modeled as a queue. At bottom is an example of the queue activity of our streaming application example. The x-axis is the queue position and the y-axis represents the # of cycles occupied within each time frame. The front that is stable is what we're interested in, can we find this quickly? One way to do that s to use queueing models, but they require some idea of service rate.

Buffer Sizing

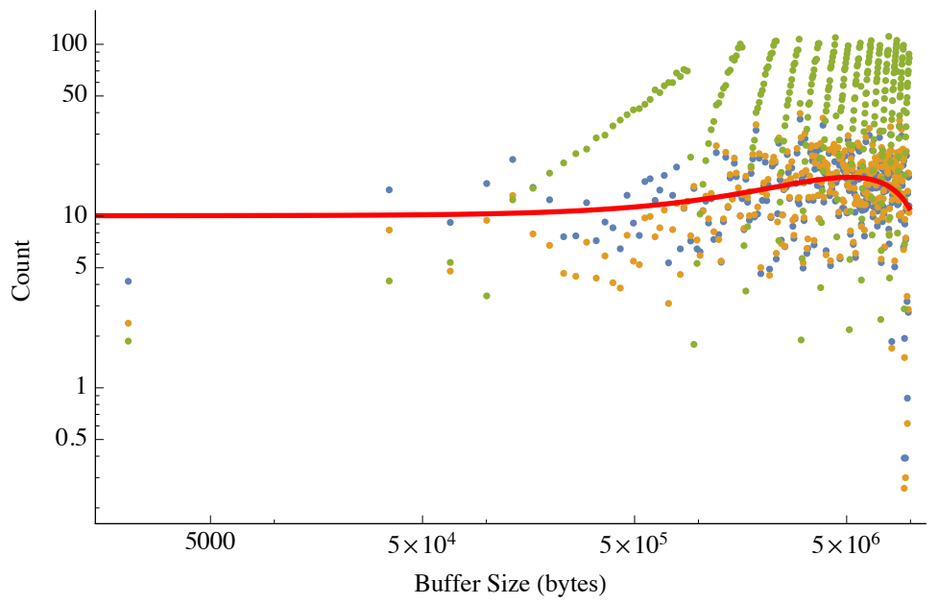


SBS
Stream Based Supercomputing Lab
http://sbs.wustl.edu

Washington University in St. Louis

Does buffer sizing have an impact on overall throughput? YES! The front from the previous slide corresponds to approximately 80 kB on this slide. Any smaller and we stifle performance. Too much larger and we start loosing performance, but why?

Buffer Sizing

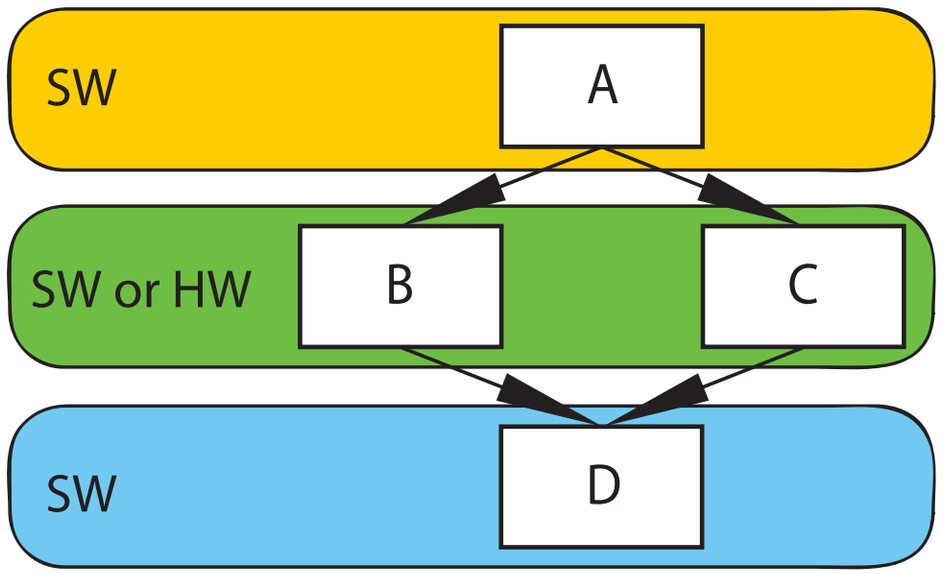


SBS
Stream Based
Supercomputing Lab
http://sbs.wustl.edu

Washington
University in St.Louis

50 MB at the far right, green dots are page faults, blue and orange dots are L1/L2 misses respectively. The basic premise is that you end up with quite a bit of locality with small buffers that are fairly cacheable, but going above a certain size eliminates the possibility that most of the buffer can end up in the cache. Big is often good for performance but too big is bad. On shared systems with lots of executing threads this can be very bad.

Partitioning Problem

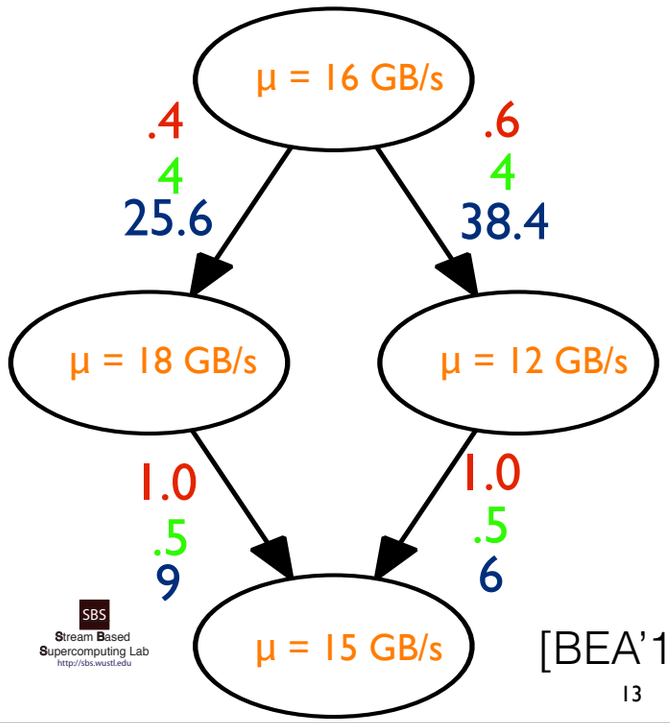


SBS
Stream Based
Supercomputing Lab
http://sbs.wustl.edu

Washington
University in St.Louis

Non-blocking service rate can also be useful for partitioning an application between compute resources. Offline heuristics work pretty well for providing a starting partition, they don't work well online. Most are too slow, in general partitioning is NP-Hard….There are plenty of decent heuristics, we're going to focus elsewhere so just keep in mind that this is another potential usage of service rate.

Flow Model



C - capacity for each edge product of:

μ - service rate of kernel

F_r- fraction of data along kernel out-edges

γ - gain function of upstream kernel

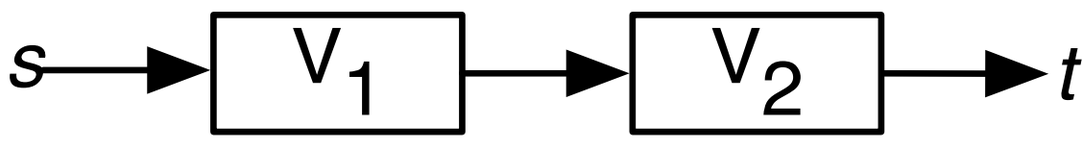
SBS Stream Based Supercomputing Lab http://sbs.wustl.edu

[BEA'13]

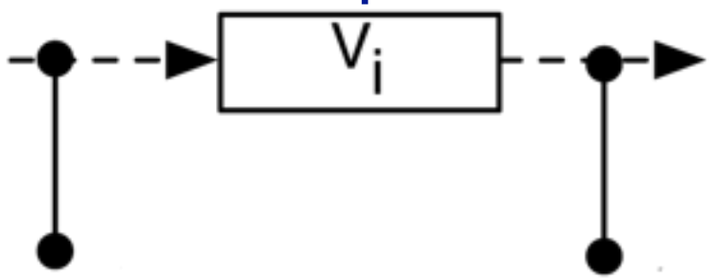
Washington University in St.Louis

In prior work I introduced using gain/loss flow models for calculating the throughput through a queueing network. The one thing that we couldn't get at the time was the *mu* on the slide (orange), our method of online service rate determination enables the use of this method during execution for things like thread migration decisions.

Traditional Service Rate



Isolated Compute Kernel



Counter - In

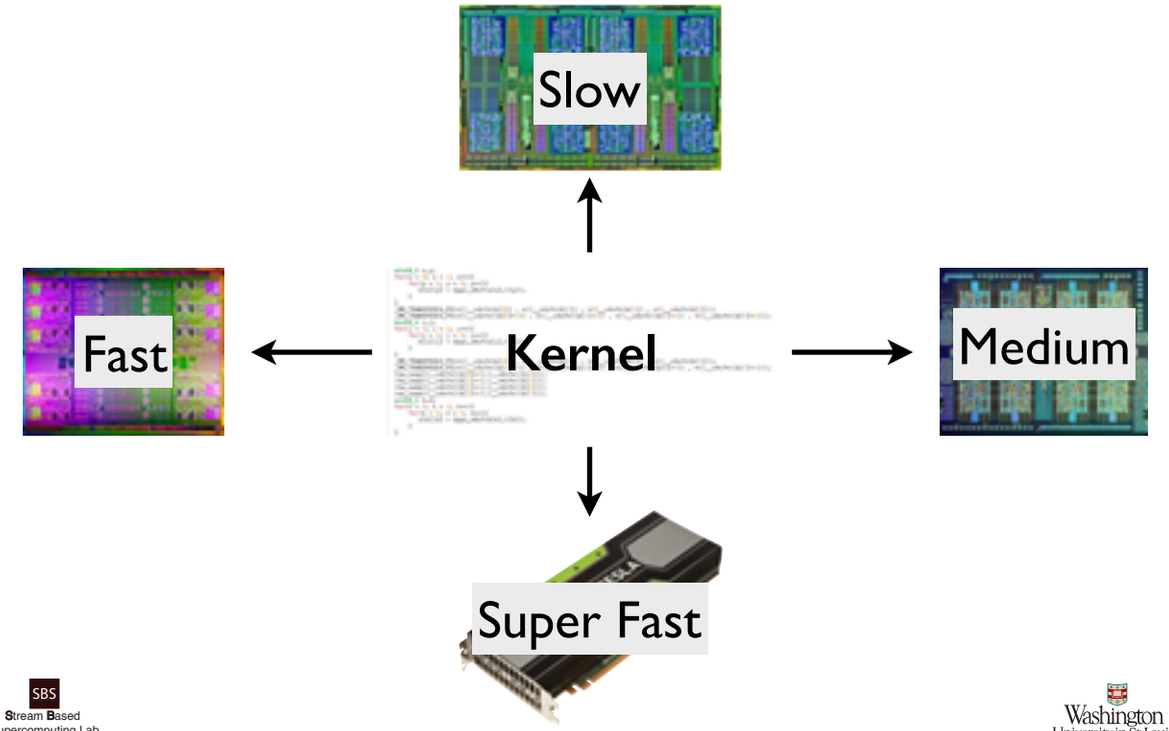
Counter - Out

SBS
Stream Based
Supercomputing Lab
http://sbs.wustl.edu

Washington
University in St.Louis

Here's the old way of figuring out how fast a compute kernel could execute outside of its network. Each kernel is characterized on its intended compute platform with its intended environment. It takes time.

How does our kernel perform on each compute resource?

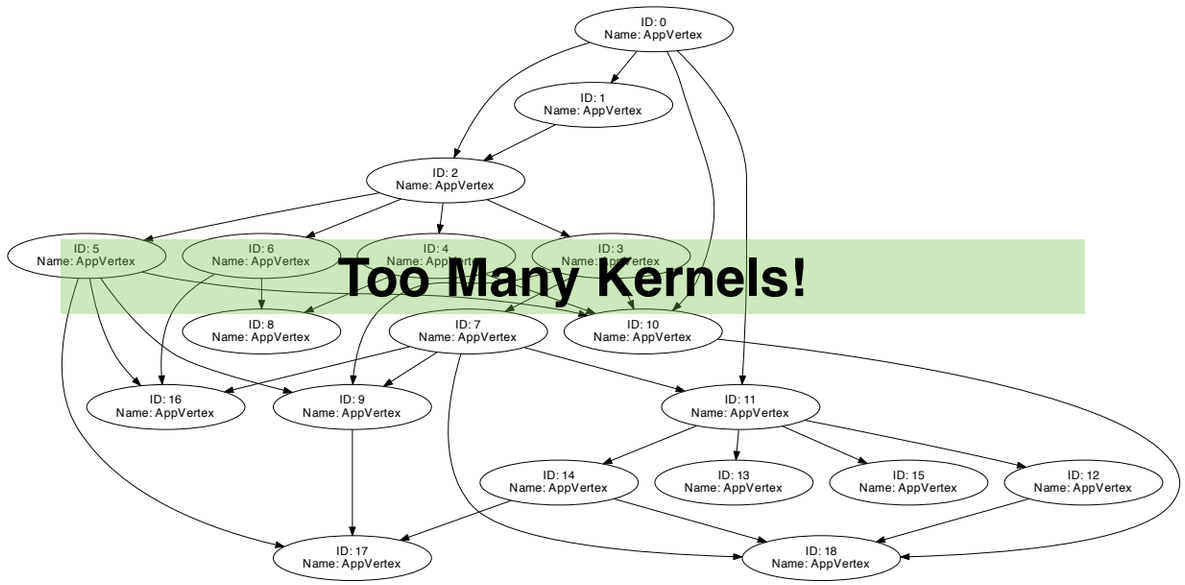


SBS
Stream Based
Supercomputing Lab
http://sbs.wustl.edu

Washington
University in St. Louis

Every time we change the assignment of a kernel to a compute resource, we have to re-characterize it. This takes a lot of time.

More Complex Example



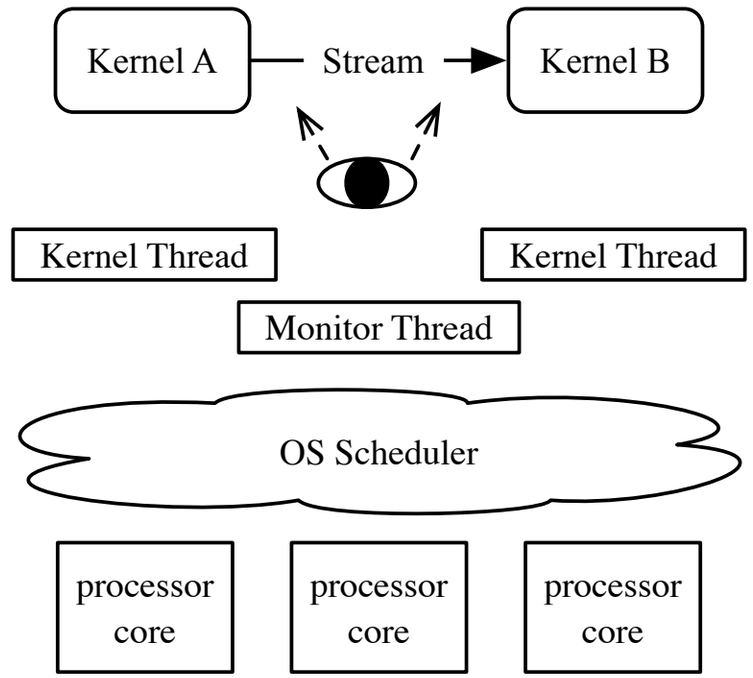
Too Many Kernels!

SBS
Stream Based
Supercomputing Lab
http://sbs.wustl.edu

Washington
University in St.Louis

for huge compute graphs individual characterization (necessary for accurate modeling) is really not feasible.

Online Instrumentation

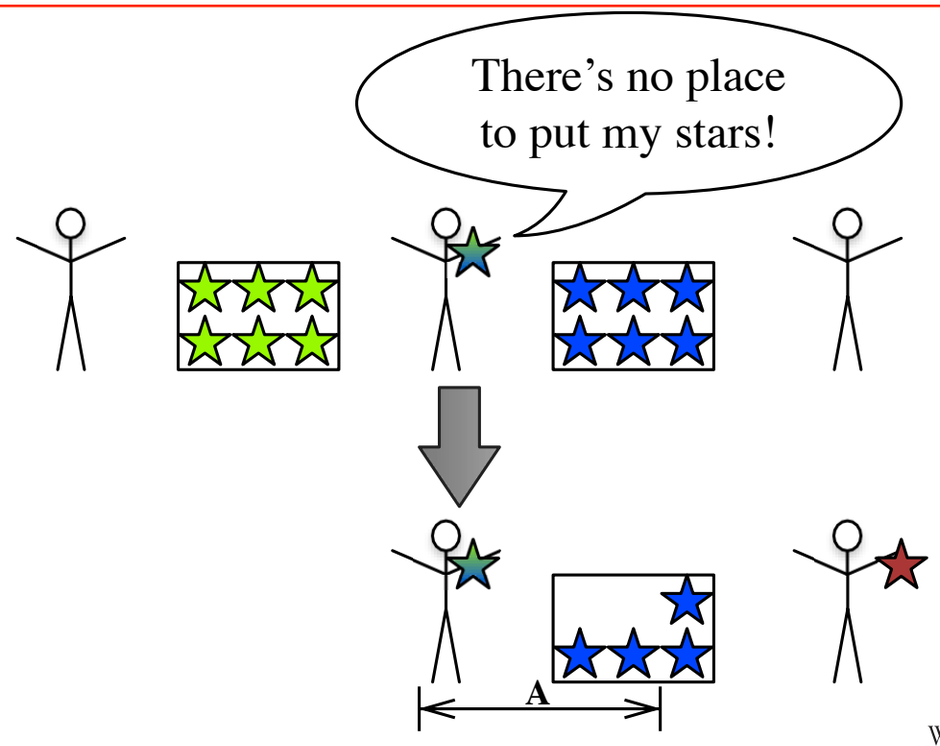


SBS
Stream Based Supercomputing Lab
http://sbs.wustl.edu

Washington University in St. Louis

The monitor thread takes samples of non-blocking reads and writes from the queue it is observing. We process these as a small window, saving only very small bits of summary data which is used to estimate the service rate

What We Want to Find

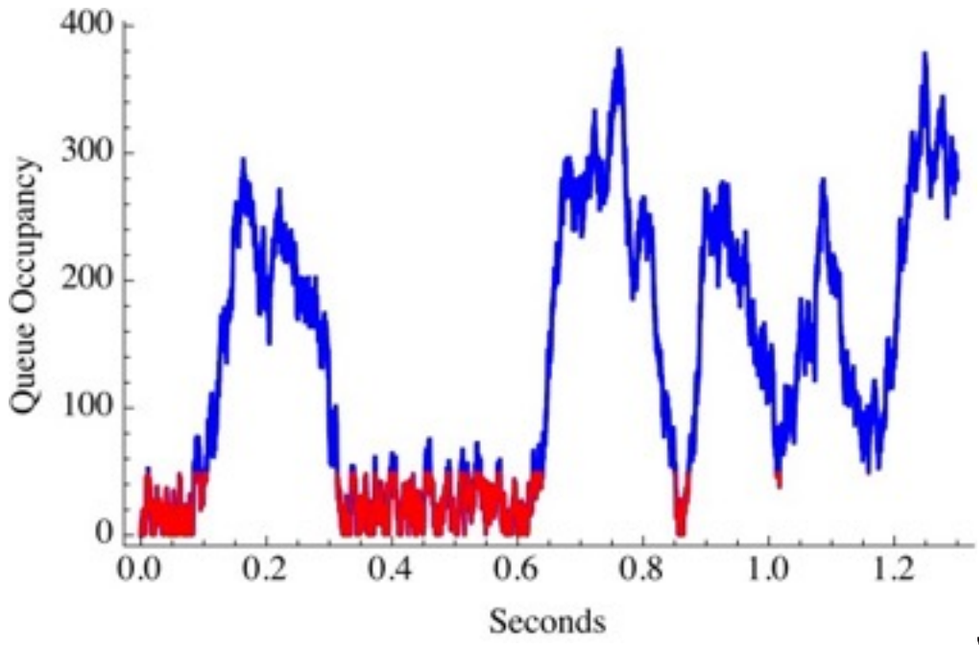


SBS
Stream Based
Supercomputing Lab
http://sbs.wustl.edu

Washington
University in St.Louis

We want to find the segment given by A, in the instant that the middle worker has an opening to add stars, then we can figure out how fast he can execute un-encumbered by the last worker.

What We Want to Find

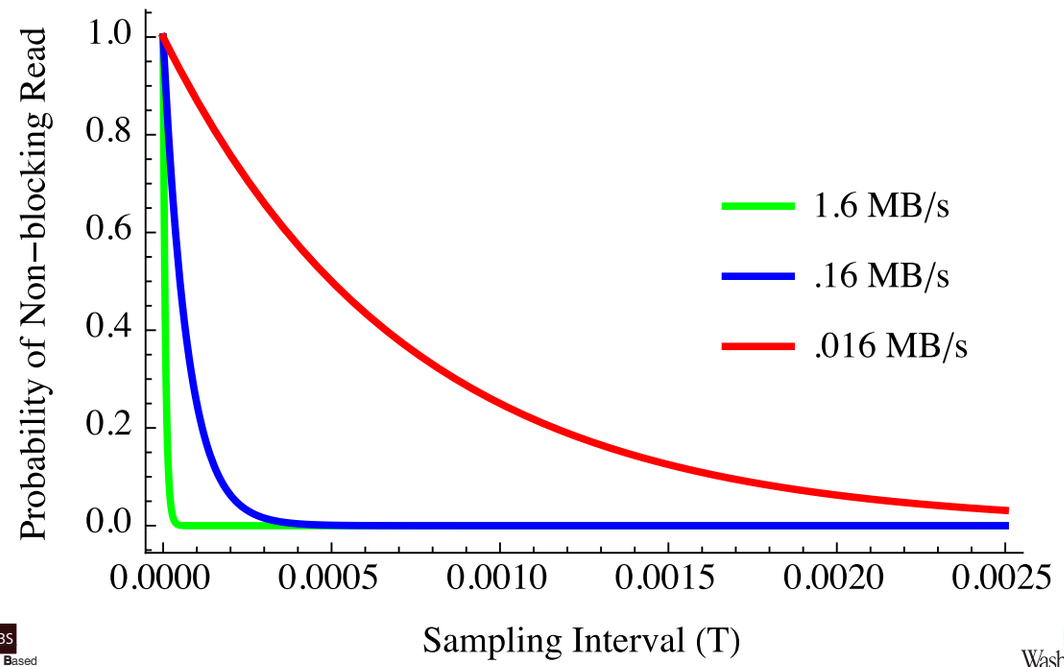


SBS
Stream Based
Supercomputing Lab
http://sbs.wustl.edu

Washington
University in St.Louis

In a high utilization real M/M/1 system, this is what it looks like. For the most part, the queue is highly occupied. Occasionally though we can find segments (colored in red) which are amenable for determining the service rate.

Mission Impossible

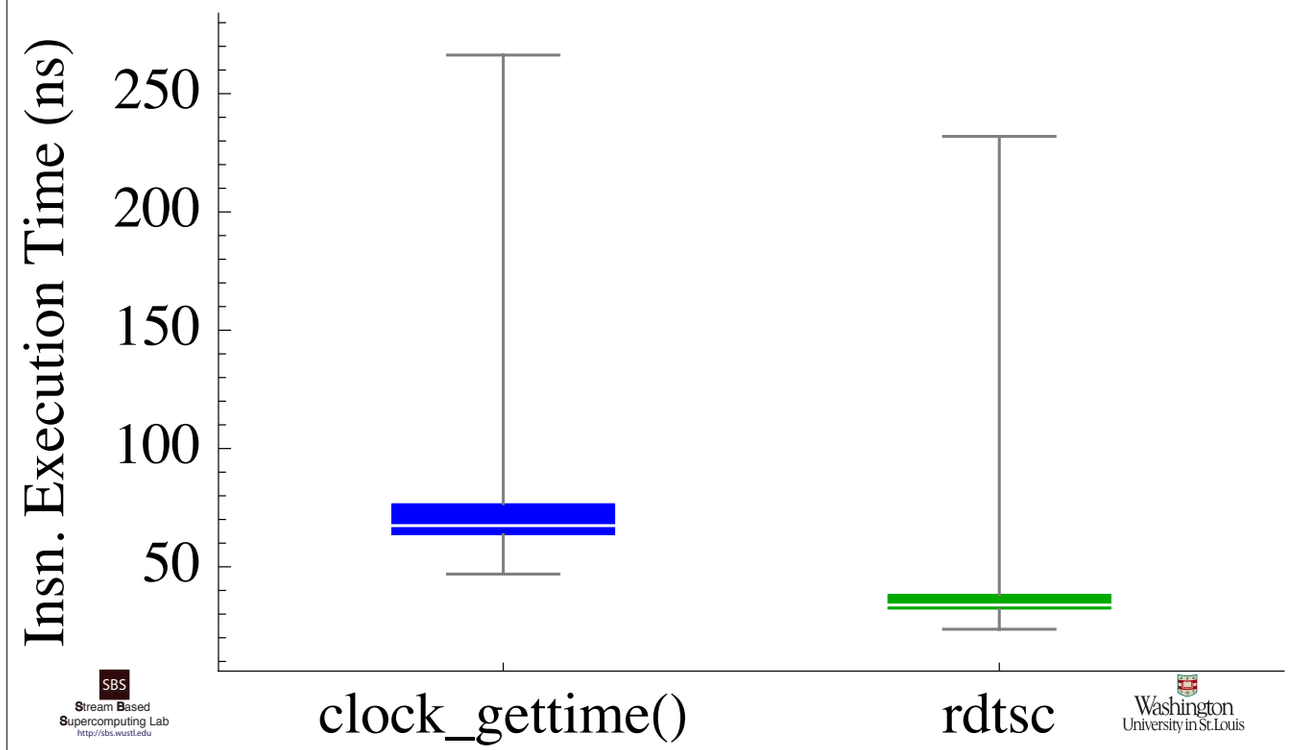


SBS
Stream Based
Supercomputing Lab
http://sbs.wustl.edu

Washington
University in St.Louis

So how probable are these segments (red from previous slide) that are needed to determine the service rate online? Well, not to likely. As the service rate increases, the probability decreases. As the sampling frame increases, the probability decreases. So we need small sampling frames to increase the likelihood that we'll see the segments in red.

Timer Latency

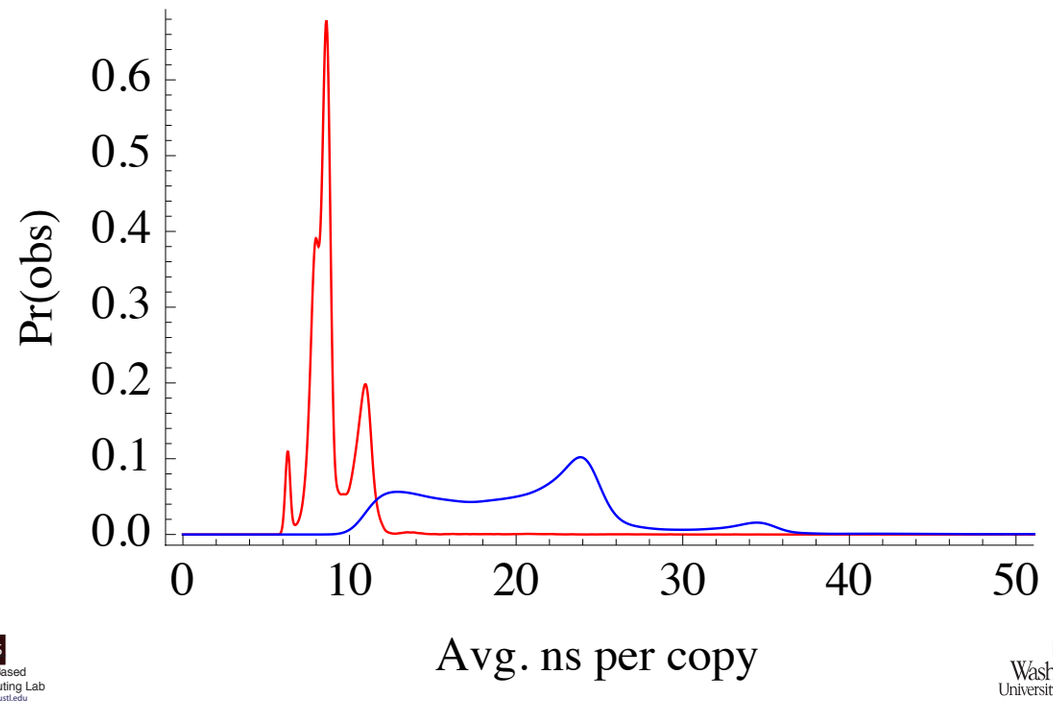


SBS
Stream Based
Supercomputing Lab
http://sbs.wustl.edu

Washington
University in St.Louis

So we need accurate timing. How accurate, well, we want as accurate as possible. One measure that we're interested in is back to back execution, over millions of executions (averaged) the rdtsc instruction has far less latency than the standard clock_gettime which is no huge surprise.

Reading Timer Latency

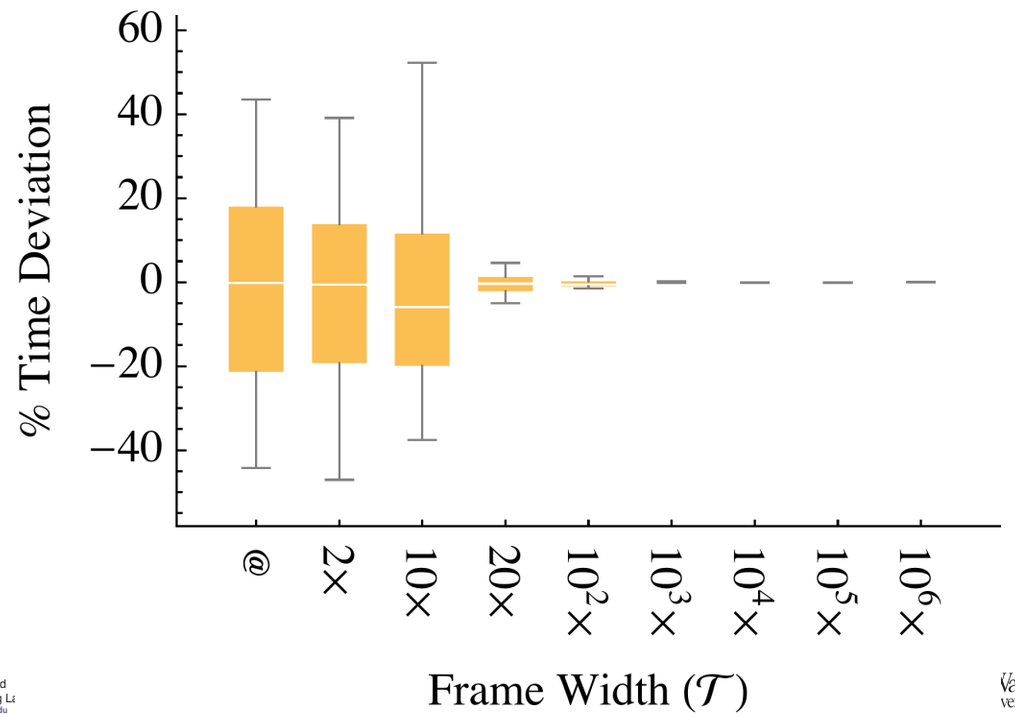


SBS
Stream Based
Supercomputing Lab
http://sbs.wustl.edu

Washington
University in St.Louis

With the timer thread executing on a single core, the latency to access the updated timer values on other cores differ (especially when going to another socket). Every update on the local core invalidates the value on the remote core forcing a QPI access of the newly updated value. Prefetching seems the likely solution, however it doesn't quite fix things so we allocate memory on the other core's NUMA node and prefetch it so that there is no aliasing and the most up to date values are more likely to be in cache speeding access. This gets us closer to the 10nanosecond access time that we see on the local core.

Timer Precision

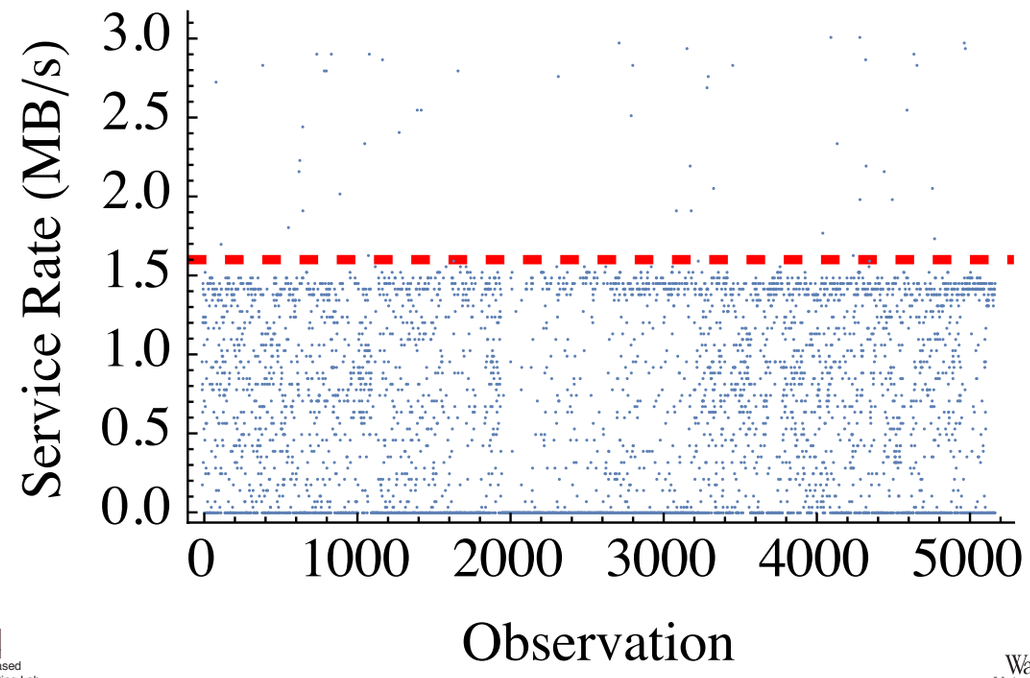


SBS
Stream Based
Supercomputing La
http://sbs.wustl.edu

Washington
versity in St.Louis

The @ symbol in the bottom left hand corner is the minimal resolution of back to back timer calls (in this case RDTSC) averaged across cores. We need a stable time frame so that means that we need to go to the right (on the x-axis) towards larger multiples of the system timer. So, now we have an issue. We have to find the smallest time frame possible, but also the most stable time frame possible. This is done the first time RaftLib starts up, and a profile is written. At startup this profile is quickly verified (may change depending on the dynamic environment), and then the profile is used if it is acceptable (otherwise we search for the time again).

Raw Observations



SBS
Stream Based
Supercomputing Lab
http://sbs.wustl.edu

Washington
University in St.Louis

These are the raw values of reads that our instrumentation thread views. At first glance it appears that there is a nice front right where our expected non-blocking service rate is (red dashed line). The key is understanding that there are still hundreds of values above the red line. A quantile based approach is the obvious one, and it is the one we ultimately took. The issue with quantiles is that we can't take quantiles of an arbitrary distribution without saving lots of state. We only want to save a few values, so we need another approach. Observing that each one of these points is in fact a sum of non-blocked reads (although realizing that due to the fact that we use no locking or atomic accesses during the gathering of this data that there are many potential outliers within the data representing something less or greater than the non-blocking rates that we're searching for). Sums of observations of a random variable tend towards a Gaussian, and we can use this.

Solutions

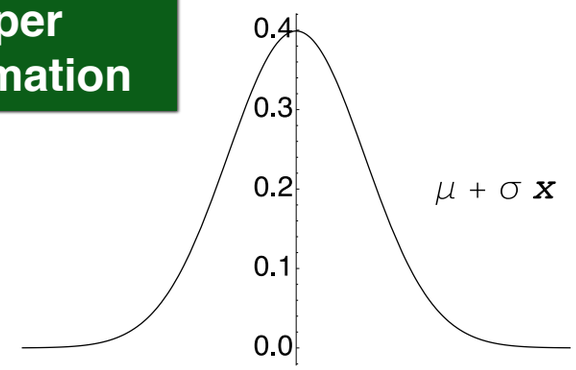
Calculate Quantile of Service Rate Observations

Using a closed form solution for the continuous Gaussian gives a solution to our data saving problem (we don't want to transmit continuously and we want something that will fit in a typical L1 or L2 cache line so that our instrumentation can be quick). But our data is still noisy.

Solutions

~~Calculate Quantile of Service Rate Observations~~

Cheaper Approximation



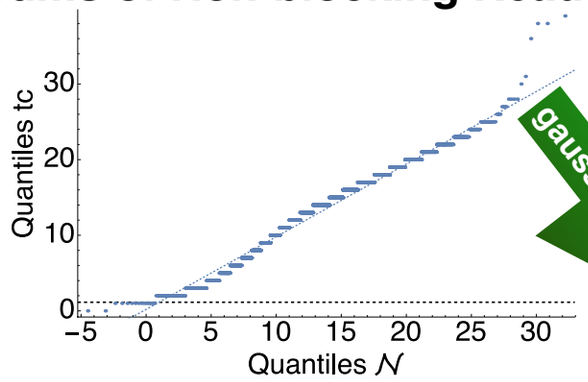
SBS
Stream Based Supercomputing Lab
http://sbs.wustl.edu

Washington University in St. Louis

Using a closed form solution for the continuous Gaussian gives a solution to our data saving problem (we don't want to transmit continuously and we want something that will fit in a typical L1 or L2 cache line so that our instrumentation can be quick). But our data is still noisy.

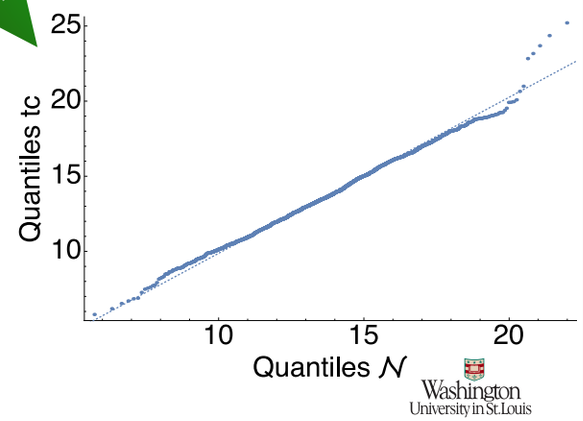
Process

Sums of Non-blocking Reads



gaussian filter

Filtered Sums

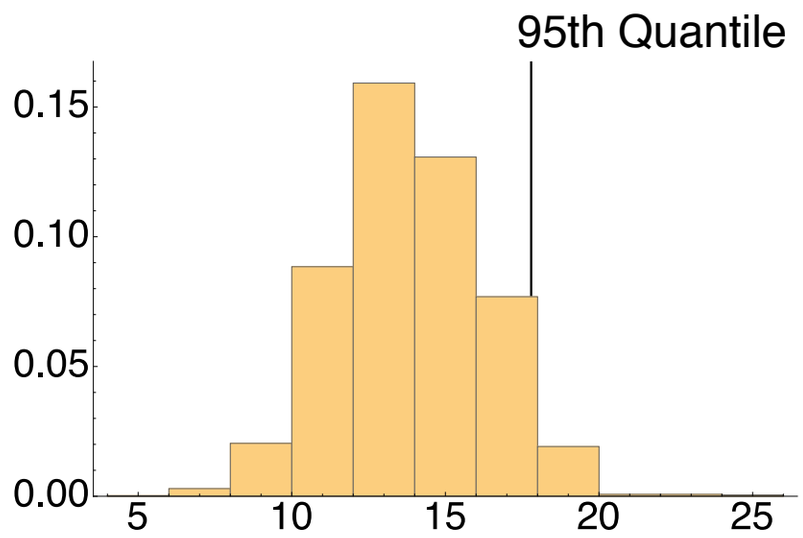


SBS
Stream Based
Supercomputing Lab
http://sbs.wustl.edu

Washington
University in St.Louis

Applying a filter over only the previous 16 values gives a less noisy view (shown on the qq plot before and after)

Process

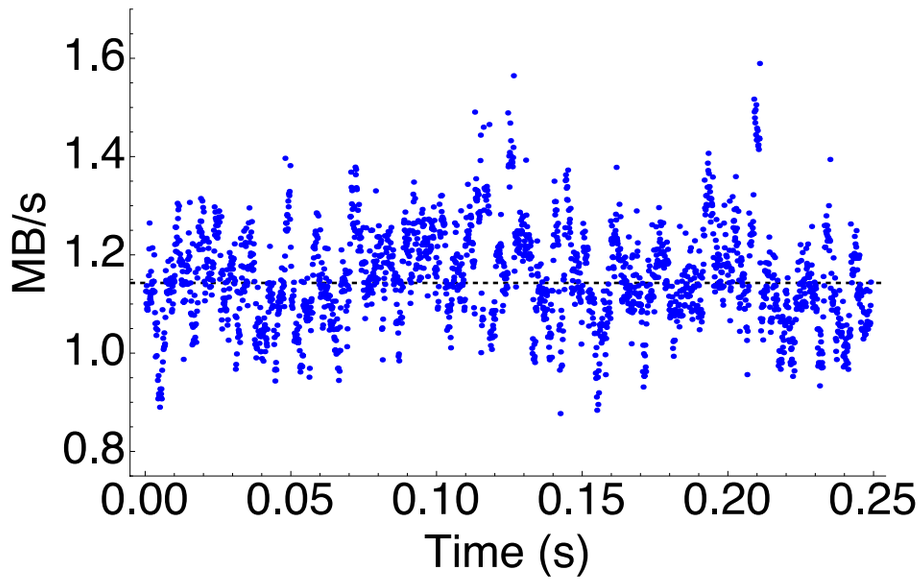


SBS
Stream Based
Supercomputing Lab
http://sbs.wustl.edu

Washington
University in St.Louis

Histogram view of the quantile we want, the rest below that is assumed to be other stuff (side effect of the atomic-less data collection)

Raw Quantile Observations

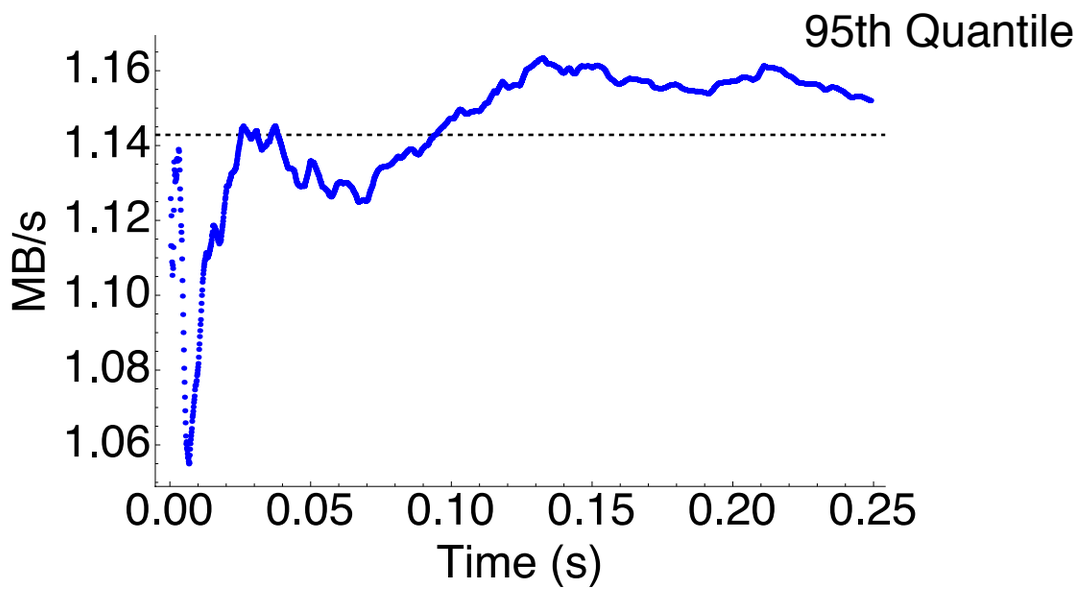


SBS
Stream Based
Supercomputing Lab
http://sbs.wustl.edu

Washington
University in St.Louis

Ok, this gets us almost to where we want. But it’s not so stable

Streaming Estimation

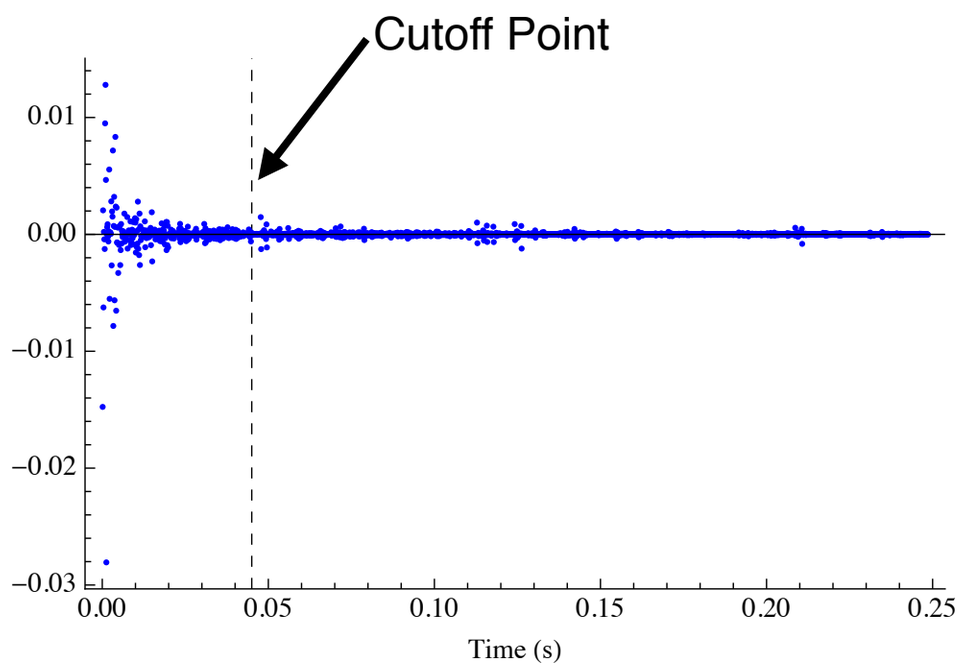


SBS
Stream Based
Supercomputing Lab
http://sbs.wustl.edu

Washington
University in St.Louis

Using a streaming mean gets us to a more stable result

Convergence Cutoff

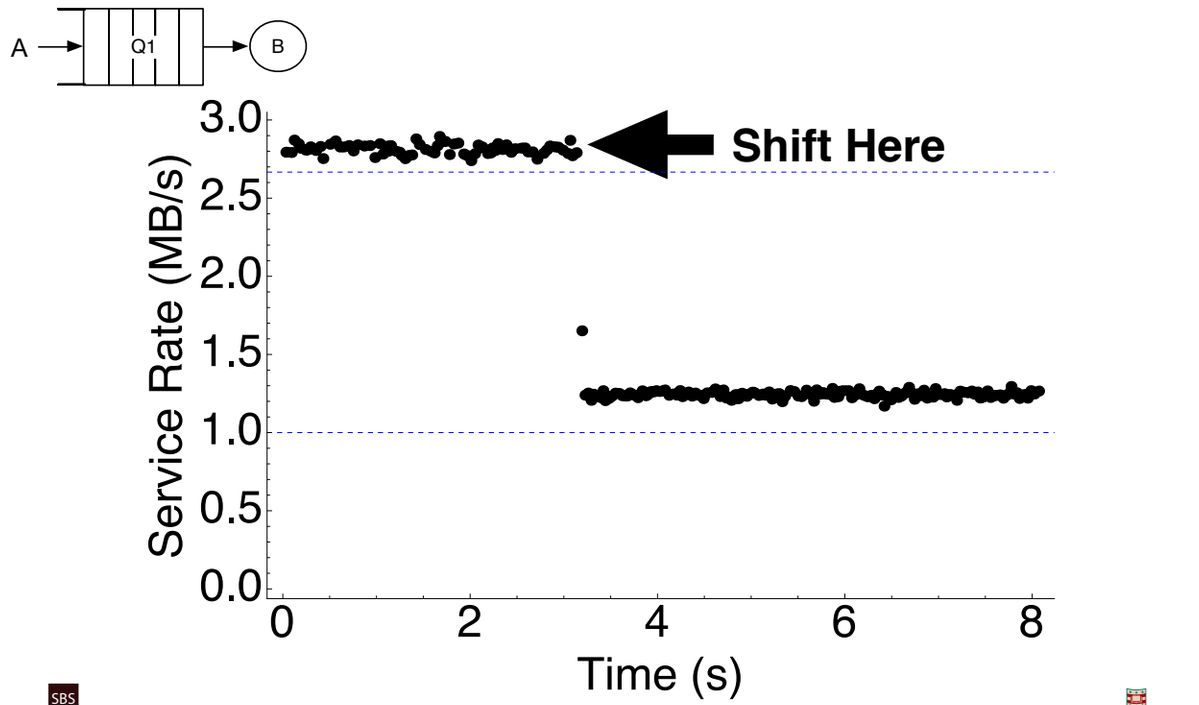


SBS
Stream Based
Supercomputing Lab
http://sbs.wustl.edu

Washington
University in St.Louis

We use a Laplacian Gaussian Filter to filter the standard deviation of the quantile estimate, commonly used in edge detection to tell when a “stable” service rate has been found. The above is a plot of the Laplacian filtered standard deviation over time, the dotted line shows the cut-off point. When we cut-off, we can re-start the instrumentation and find another service rate (which could have changed during execution).

Micro-Benchmark Test

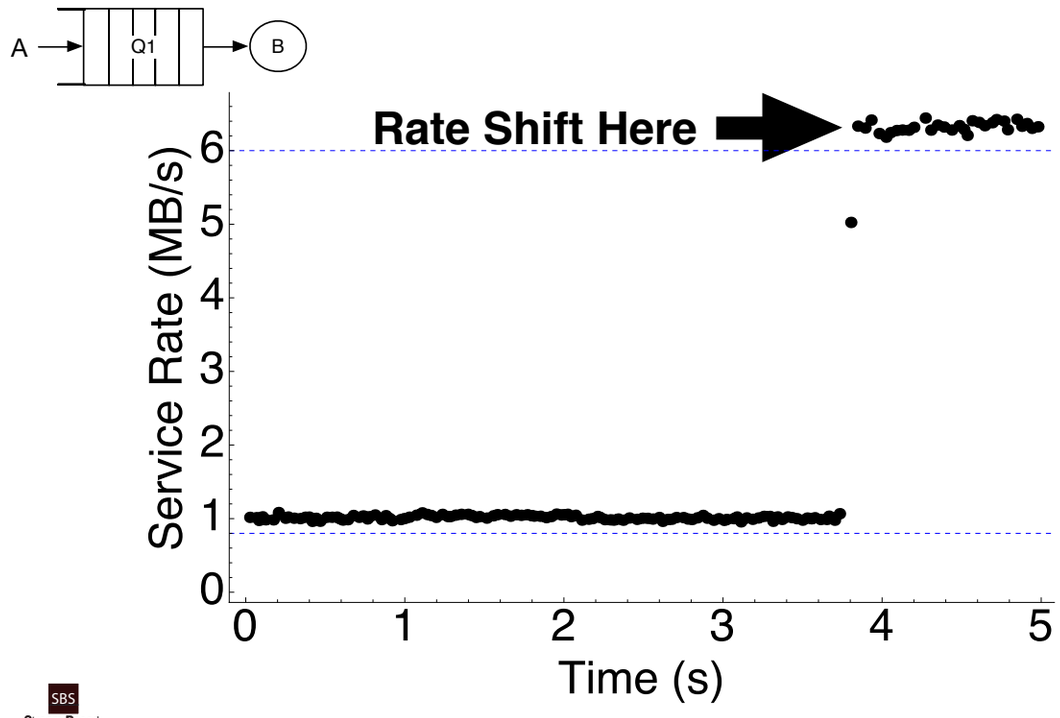


SBS
Stream Based
Supercomputing Lab
http://sbs.wustl.edu

Washington
University in St.Louis

Instrumentation provided service rates from micro-benchmark that shifts the service rate of B halfway through its execution (in elements, not time).

Instrumentation In Action



SBS
Stream Based
Supercomputing Lab
http://sbs.wustl.edu

Washington
University in St.Louis

Going the other way, same benchmark…different shift.

RaftLib

C++ Streaming Template Library

Auto-parallelizes code

Manages resources, buffers, TCP links

GOAL: Automatically Optimized Online

software download: http://raftlib.io

Online Instrumentation

- Queue Occupancy (mean, histogram, etc.)
- Service Rate (non-blocking and actual throughput)
- Process Distribution**
- Less than 1% impact on processor load on average with our implementation
- Execution times not affected by instrumentation by any statistically significant measure
- Can be turned on and off dynamically through queue reallocation process

SBS
Stream Based Supercomputing Lab
http://sbs.wustl.edu

Washington University in St. Louis

**Currently only supported in experimental branch and with method of moments, eventually will migrate once I've explored using kernel methods vs. moments since the moments is a bit expensive to compute still.

Conclusion

- Described a method to approximate non-blocking service rate while executing
- Shown that it works well empirically and if not fails with notification
- Described briefly how it can be used in online optimization processes

RaftLib: http://raftlib.io
My Page: http://jonathanbeard.io
Email: jonathan.beard@arm.com