Analyzing Queueing Networks with Multiclass Fork-Join Constructs

Joel Choo (cyc15@ic.ac.uk)

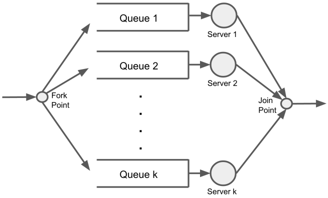
Imperial College London

September 2, 2016

What is a fork-join queue?

Definition

A fork-join queue is a queue where incoming jobs are split on arrival for service by numerous servers and joined before departure



Objective

We want to come up with an approximation method for multiclass fork-join queueing networks with an implementation that is:

- Accurate
- Efficient
- Universal

Overview

Two important building blocks:

- Decay Rate Approximation (DRA)
- Fork-Join Approximate Mean Value Analysis (FJ-AMVA)

Decay Rate Approximation

- Method to approximate multiclass queueing networks
- Does not work with fork-join
- Iteratively solves for the queue lengths and throughputs
- Our method uses the same high level approach/ideas as DRA

Decay Rate Approximation

- Method to approximate multiclass queueing networks
- Does not work with fork-join
- Iteratively solves for the queue lengths and throughputs
- Our method uses the same high level approach/ideas as DRA

Decay Rate Approximation

- Method to approximate multiclass queueing networks
- Does not work with fork-join
- Iteratively solves for the queue lengths and throughputs
- Our method uses the same high level approach/ideas as DRA

Decay Rate Approximation

- Method to approximate multiclass queueing networks
- Does not work with fork-join
- Iteratively solves for the queue lengths and throughputs
- Our method uses the same high level approach/ideas as DRA

Decay Rate Approximation

Obtain initial estimate of throughput, X_i
Optimize for min $f(X)$ locally around X_i (using fmincon)
def $f(X)$:

- Compute utilization, $\vec{\rho}$, for each queue $\rho_{q,c} = X_c \cdot \theta_{q,c}$
- Compute arrival MMAP[k] into each queue as superposition of departure processes of feeding queues scaled by $\vec{\rho}$
- Compute decay rate of each queue when treated as a single MMAP[k]/PH[k]/1 queue
- Use decay rates with product form solver to obtain new estimates for utilization $\vec{\rho'}$ and queue lengths
- Return $|\vec{\rho} - \vec{\rho'}|$

Decay Rate Approximation

Obtain initial estimate of throughput, X_i
Optimize for min $f(X)$ locally around X_i (using fmincon)
def $f(X)$:

- Compute utilization, $\vec{\rho}$, for each queue $\rho_{q,c} = X_c \cdot \theta_{q,c}$
- Compute arrival MMAP[k] into each queue as superposition of departure processes of feeding queues scaled by $\vec{\rho}$
- Compute decay rate of each queue when treated as a single MMAP[k]/PH[k]/1 queue
- Use decay rates with product form solver to obtain new estimates for utilization $\vec{\rho'}$ and queue lengths
- Return $|\vec{\rho} - \vec{\rho'}|$

Decay Rate Approximation

Obtain initial estimate of throughput, X_i
Optimize for min $f(X)$ locally around X_i (using fmincon)
def $f(X)$:

- Compute utilization, $\vec{\rho}$, for each queue $\rho_{q,c} = X_c \cdot \theta_{q,c}$
- Compute arrival MMAP[k] into each queue as superposition of departure processes of feeding queues scaled by $\vec{\rho}$
- Compute decay rate of each queue when treated as a single MMAP[k]/PH[k]/1 queue
- Use decay rates with product form solver to obtain new estimates for utilization $\vec{\rho'}$ and queue lengths
- Return $|\vec{\rho} - \vec{\rho'}|$

Decay Rate Approximation

Obtain initial estimate of throughput, X_i
Optimize for min $f(X)$ locally around X_i (using fmincon)
def $f(X)$:

- Compute utilization, $\vec{\rho}$, for each queue $\rho_{q,c} = X_c \cdot \theta_{q,c}$
- Compute arrival MMAP[k] into each queue as superposition of departure processes of feeding queues scaled by $\vec{\rho}$
- Compute decay rate of each queue when treated as a single MMAP[k]/PH[k]/1 queue
- Use decay rates with product form solver to obtain new estimates for utilization $\vec{\rho'}$ and queue lengths
- Return $|\vec{\rho} - \vec{\rho'}|$

Decay Rate Approximation

Obtain initial estimate of throughput, X_i
Optimize for min $f(X)$ locally around X_i (using fmincon)
def $f(X)$:

- Compute utilization, $\vec{\rho}$, for each queue $\rho_{q,c} = X_c \cdot \theta_{q,c}$
- Compute arrival MMAP[k] into each queue as superposition of departure processes of feeding queues scaled by $\vec{\rho}$
- Compute decay rate of each queue when treated as a single MMAP[k]/PH[k]/1 queue
- Use decay rates with product form solver to obtain new estimates for utilization $\vec{\rho'}$ and queue lengths
- Return $|\vec{\rho} - \vec{\rho'}|$

Decay Rate Approximation

Obtain initial estimate of throughput, X_i
Optimize for min $f(X)$ locally around X_i (using fmincon)
def $f(X)$:

- Compute utilization, $\vec{\rho}$, for each queue $\rho_{q,c} = X_c \cdot \theta_{q,c}$
- Compute arrival MMAP[k] into each queue as superposition of departure processes of feeding queues scaled by $\vec{\rho}$
- Compute decay rate of each queue when treated as a single MMAP[k]/PH[k]/1 queue
- Use decay rates with product form solver to obtain new estimates for utilization $\vec{\rho'}$ and queue lengths
- Return $|\vec{\rho} - \vec{\rho'}|$

Fork-Join AMVA

- Fork-Join Approximate Mean Value Analysis
- Approximation method for multiclass fork-join queueing networks
- Iteratively solve for the mean queue lengths and throughputs
- Best approximation method we know of
- Compare our results against this
- Used in our method to initialize the queue lengths and throughputs

Fork-Join AMVA

- Fork-Join Approximate Mean Value Analysis
- Approximation method for multiclass fork-join queueing networks
- Iteratively solve for the mean queue lengths and throughputs
- Best approximation method we know of
- Compare our results against this
- Used in our method to initialize the queue lengths and throughputs

Fork-Join AMVA

- Fork-Join Approximate Mean Value Analysis
- Approximation method for multiclass fork-join queueing networks
- Iteratively solve for the mean queue lengths and throughputs
- Best approximation method we know of
- Compare our results against this
- Used in our method to initialize the queue lengths and throughputs

Fork-Join AMVA

- Fork-Join Approximate Mean Value Analysis
- Approximation method for multiclass fork-join queueing networks
- Iteratively solve for the mean queue lengths and throughputs
- Best approximation method we know of
- Compare our results against this
- Used in our method to initialize the queue lengths and throughputs

Fork-Join AMVA

- Fork-Join Approximate Mean Value Analysis
- Approximation method for multiclass fork-join queueing networks
- Iteratively solve for the mean queue lengths and throughputs
- Best approximation method we know of
- Compare our results against this
- Used in our method to initialize the queue lengths and throughputs

FJ-AMVA Algorithm

In each iteration:

- Compute residence time for all queues as $R'_i(n) = s_i \cdot (1 + \bar{n}_i(n-1))$
- Re-number queues such that $R'_1(n) \geq R'_2(n) \geq \cdots \geq R'_K(n)$
- Compute residence time for FJ construct as $\sum_{k=1}^{K} \frac{1}{k} R'_k(n)$
- Compute throughput as $X(n) = n / \sum_{k=1}^{K} \frac{1}{k} R'_k(n)$
- Compute new mean queue lengths as $\bar{n}_i(n) = X(n) \cdot R'_i(n)$

Terminate loop when the difference in successive mean queue lengths is less than some ϵ

FJ-AMVA Algorithm

In each iteration:

- Compute residence time for all queues as $R'_i(n) = s_i \cdot (1 + \bar{n}_i(n-1))$
- Re-number queues such that $R'_1(n) \geq R'_2(n) \geq \cdots \geq R'_K(n)$
- Compute residence time for FJ construct as $\sum_{k=1}^{K} \frac{1}{k} R'_k(n)$
- Compute throughput as $X(n) = n / \sum_{k=1}^{K} \frac{1}{k} R'_k(n)$
- Compute new mean queue lengths as $\bar{n}_i(n) = X(n) \cdot R'_i(n)$

Terminate loop when the difference in successive mean queue lengths is less than some ϵ

FJ-AMVA Algorithm

In each iteration:

- Compute residence time for all queues as $R_i'(n) = s_i \cdot (1 + \bar{n}_i(n-1))$
- Re-number queues such that $R_1'(n) \geq R_2'(n) \geq \cdots \geq R_K'(n)$
- Compute residence time for FJ construct as $\sum_{k=1}^{K} \frac{1}{k} R_k'(n)$
- Compute throughput as $X(n) = n / \sum_{k=1}^{K} \frac{1}{k} R_k'(n)$
- Compute new mean queue lengths as $\bar{n}_i(n) = X(n) \cdot R_i'(n)$

Terminate loop when the difference in successive mean queue lengths is less than some ϵ

FJ-AMVA Algorithm

In each iteration:

- Compute residence time for all queues as $R'_i(n) = s_i \cdot (1 + \bar{n}_i(n-1))$
- Re-number queues such that $R'_1(n) \geq R'_2(n) \geq \cdots \geq R'_K(n)$
- Compute residence time for FJ construct as $\sum_{k=1}^{K} \frac{1}{k} R'_k(n)$
- Compute throughput as $X(n) = n / \sum_{k=1}^{K} \frac{1}{k} R'_k(n)$
- Compute new mean queue lengths as $\bar{n}_i(n) = X(n) \cdot R'_i(n)$

Terminate loop when the difference in successive mean queue lengths is less than some ϵ

FJ-AMVA Algorithm

In each iteration:

- Compute residence time for all queues as $R'_i(n) = s_i \cdot (1 + \bar{n}_i(n-1))$
- Re-number queues such that $R'_1(n) \geq R'_2(n) \geq \cdots \geq R'_K(n)$
- Compute residence time for FJ construct as $\sum_{k=1}^{K} \frac{1}{k} R'_k(n)$
- Compute throughput as $X(n) = n / \sum_{k=1}^{K} \frac{1}{k} R'_k(n)$
- Compute new mean queue lengths as $\bar{n}_i(n) = X(n) \cdot R'_i(n)$

Terminate loop when the difference in successive mean queue lengths is less than some ϵ

FJ-AMVA Algorithm

In each iteration:

- Compute residence time for all queues as $R'_i(n) = s_i \cdot (1 + \bar{n}_i(n-1))$
- Re-number queues such that $R'_1(n) \geq R'_2(n) \geq \cdots \geq R'_K(n)$
- Compute residence time for FJ construct as $\sum_{k=1}^{K} \frac{1}{k} R'_k(n)$
- Compute throughput as $X(n) = n / \sum_{k=1}^{K} \frac{1}{k} R'_k(n)$
- Compute new mean queue lengths as $\bar{n}_i(n) = X(n) \cdot R'_i(n)$

Terminate loop when the difference in successive mean queue lengths is less than some ϵ

Fork-Join DRA

- Extends the DRA method to work for fork-join queueing networks
- Same high level idea as DRA, but we modify how some steps are performed

Fork-Join DRA

- Extends the DRA method to work for fork-join queueing networks
- Same high level idea as DRA, but we modify how some steps are performed

Recap on DRA

Recall the DRA method:

Obtain initial estimate of throughput, X_i
Optimize for min $f(X)$ locally around X_i (using fmincon) def $f(X)$:

- Compute utilization, $\vec{\rho}$, for each queue $\rho_{q,c} = X_c \cdot \theta_{q,c}$
- Compute arrival MMAP[k] into each queue as superposition of departure processes of feeding queues scaled by $\vec{\rho}$
- Compute decay rate of each queue when treated as a single MMAP[k]/PH[k]/1 queue
- Use decay rates with product form solver to obtain new estimates for utilization $\vec{\rho'}$ and queue lengths
- Return $|\vec{\rho} - \vec{\rho'}|$

Recap on DRA

Recall the DRA method:

Obtain initial estimate of throughput, X_i
Optimize for min $f(X)$ locally around X_i (using fmincon) def $f(X)$:

- Compute utilization, $\vec{\rho}$, for each queue $\rho_{q,c} = X_c \cdot \theta_{q,c}$
- Compute arrival MMAP[k] into each queue as superposition of departure processes of feeding queues scaled by $\vec{\rho}$
- Compute decay rate of each queue when treated as a single MMAP[k]/PH[k]/1 queue
- Use decay rates with product form solver to obtain new estimates for utilization $\vec{\rho'}$ and queue lengths
- Return $|\vec{\rho} - \vec{\rho'}|$

Initialization

Two possible methods of initializing the throughput:

- AMVA-FCFS (same as DRA)
- Fork-Join AMVA

Initialization

- DRA uses a simple iterative solver, AMVA-FCFS
- Requires the mean service demand, $\theta_{ir} = v_{ir} \cdot s_{ir}$
- s_{ir} : mean service time per visit
- $s_{ir} = \frac{1}{\mu_{ir}}$
- v_{ir} : mean number of visits of class r jobs to queue i
- $v_{ir} = \sum_{j=1}^{M} P_{ji} \cdot v_{jr}$
- P_{ij} : routing probability from queue i to queue j
- Visits to queue i is the sum of the visits to queues feeding into queue i

Initialization

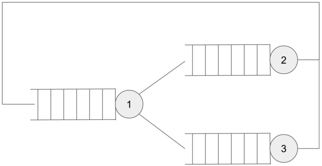
- DRA uses a simple iterative solver, AMVA-FCFS
- Requires the mean service demand, $\theta_{ir} = v_{ir} \cdot s_{ir}$
- s_{ir} : mean service time per visit
- $s_{ir} = \frac{1}{\mu_{ir}}$
- v_{ir} : mean number of visits of class r jobs to queue i
- $v_{ir} = \sum_{j=1}^{M} P_{ji} \cdot v_{jr}$
- P_{ij} : routing probability from queue i to queue j
- Visits to queue i is the sum of the visits to queues feeding into queue i

Initialization

- For FJ queueing networks, the sum of visits of feeding queues will overcount for queues after the join point
- To deal with that, we need to use the sum of visits of feeding queues into the fork point instead

Initialization - Comparison

- We compared the accuracy of the approximations obtained when initializing with both methods
- We refer to the mean queue length error across all queues and classes
- error $= \frac{1}{2K} \sum_{i=1}^{M} \sum_{r=1}^{R} |Q_{i,r} - \hat{Q}_{i,r}|$
- Tested on the queueing network below



Initialization - Comparison

$\lambda_{1,1}$	$\lambda_{1,2}$	$\lambda_{2,1}$	$\lambda_{2,2}$	$\lambda_{3,1}$	$\lambda_{3,2}$	K_1	K_2	Error (FCFS)	Error (FJ)
2	4	2	4	2	4	1	1	0.1082	0.1082
2	4	2	4	2	4	2	2	0.1196	0.1196
2	4	2	4	2	4	3	3	0.1248	0.1247
2	4	2	4	2	4	4	4	0.1289	0.1289
2	4	2	2	4	8	2	3	0.0476	0.0476
2	4	2	4	3	6	2	3	0.0862	0.0862
2	4	2	4	4	8	2	3	0.0607	0.0607
2	4	2	4	6	12	2	3	0.0467	0.0467

Table: Error for Different Initialization Methods

Initialization - Comparison

- Both methods are $O(IMK)$ where I is the number of iterations
- Runtime of both methods are orders of magnitude smaller than the overall solver
- FJ-AMVA produces initial estimations that are much closer to the actual values
- FJ-AMVA is easier to use, no extra work required to prepare input
- No discernible difference in accuracy or performance
- We chose FJ-AMVA as the initialization method for ease of use

Initialization - Comparison

- Both methods are $O(IMK)$ where I is the number of iterations
- Runtime of both methods are orders of magnitude smaller than the overall solver
- FJ-AMVA produces initial estimations that are much closer to the actual values
- FJ-AMVA is easier to use, no extra work required to prepare input
- No discernible difference in accuracy or performance
- We chose FJ-AMVA as the initialization method for ease of use

Initialization - Comparison

- Both methods are $O(IMK)$ where I is the number of iterations
- Runtime of both methods are orders of magnitude smaller than the overall solver
- FJ-AMVA produces initial estimations that are much closer to the actual values
- FJ-AMVA is easier to use, no extra work required to prepare input
- No discernible difference in accuracy or performance
- We chose FJ-AMVA as the initialization method for ease of use

Computing Arrival Processes - Recap

Recall the DRA method:

Obtain initial estimate of throughput, X_i
Optimize for min $f(X)$ locally around X_i (using fmincon)
def $f(X)$:

- Compute utilization, $\vec{\rho}$, for each queue $\rho_{q,c} = X_c \cdot \theta_{q,c}$
- Compute arrival MMAP[k] into each queue as superposition of departure processes of feeding queues scaled by $\vec{\rho}$
- Compute decay rate of each queue when treated as a single MMAP[k]/PH[k]/1 queue
- Use decay rates with product form solver to obtain new estimates for utilization $\vec{\rho'}$ and queue lengths
- Return $|\vec{\rho} - \vec{\rho'}|$

Computing Arrival Processes - Recap

Recall the DRA method:

Obtain initial estimate of throughput, X_i
Optimize for min $f(X)$ locally around X_i (using fmincon)
def $f(X)$:

- Compute utilization, $\vec{\rho}$, for each queue $\rho_{q,c} = X_c \cdot \theta_{q,c}$
- Compute arrival MMAP[k] into each queue as superposition of departure processes of feeding queues scaled by $\vec{\rho}$
- Compute decay rate of each queue when treated as a single MMAP[k]/PH[k]/1 queue
- Use decay rates with product form solver to obtain new estimates for utilization $\vec{\rho'}$ and queue lengths
- Return $|\vec{\rho} - \vec{\rho'}|$

Computing Arrival Processes

- For queues after the join point, we need a new way to compute the arrival process
- For all other queues, arrival process is unaffected and nothing new is required

Synchronizing Fork-Join Queues

Existing method to approximate the departure process from join point by assuming finite length synchronization queues

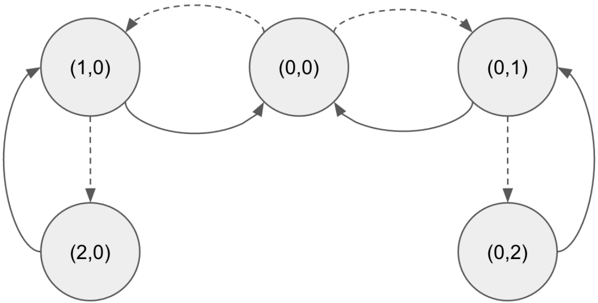


Figure: MAP representing departure process with sync queue length = 2

Synchronizing Fork-Join Queues

- Consider the service process for class r at queue i
- We approximate the departure process as the service process multiplied by $\rho_{i,r}$
- Use the method to generate a new D_0 matrix in exactly the same way as before
- Consider each class one at a time and generate new $D_{1,i}$ matrices
- Set $D_1 = \sum_{i=1}^{M} D_{1,i}$
- Normalize the MMAP, ensuring $D_0 + D_1$ is a valid transition rate matrix

Using Product Form Solver

Recall the DRA method:

Obtain initial estimate of throughput, X_i
Optimize for min $f(X)$ locally around X_i (using fmincon)
def $f(X)$:

- Compute utilization, $\vec{\rho}$, for each queue $\rho_{q,c} = X_c \cdot \theta_{q,c}$
- Compute arrival MMAP[k] into each queue as superposition of departure processes of feeding queues scaled by $\vec{\rho}$
- Compute decay rate of each queue when treated as a single MMAP[k]/PH[k]/1 queue
- Use decay rates with product form solver to obtain new estimates for utilization $\vec{\rho'}$ and queue lengths
- Return $|\vec{\rho} - \vec{\rho'}|$

Using Product Form Solver

Recall the DRA method:

Obtain initial estimate of throughput, X_i
Optimize for min $f(X)$ locally around X_i (using fmincon)
def $f(X)$:

- Compute utilization, $\vec{\rho}$, for each queue $\rho_{q,c} = X_c \cdot \theta_{q,c}$
- Compute arrival MMAP[k] into each queue as superposition of departure processes of feeding queues scaled by $\vec{\rho}$
- Compute decay rate of each queue when treated as a single MMAP[k]/PH[k]/1 queue
- Use decay rates with product form solver to obtain new estimates for utilization $\vec{\rho'}$ and queue lengths
- Return $|\vec{\rho} - \vec{\rho'}|$

Using Product Form Solver

- Part of the input to the product form (PF) solver is the job population for each class
- Mean queue lengths from PF solver will be based on those populations
- $c_i = \sum_{q=1}^{M} Q_{q,i}^{(PF)}$
- where c_i is the population of class i
- $Q_{q,i}^{(PF)}$ is the mean queue lengths for class i at queue q obtained from the PF solver
- However, when we have fork-join queues we have:
- $c_i < \sum_{q=1}^{M} Q_{q,i}$
- because one job splits into multiple jobs at the fork point

Using Product Form Solver

- Part of the input to the product form (PF) solver is the job population for each class
- Mean queue lengths from PF solver will be based on those populations
- $c_i = \sum_{q=1}^{M} Q_{q,i}^{(PF)}$
- where c_i is the population of class i
- $Q_{q,i}^{(PF)}$ is the mean queue lengths for class i at queue q obtained from the PF solver
- However, when we have fork-join queues we have:
- $c_i < \sum_{q=1}^{M} Q_{q,i}$
- because one job splits into multiple jobs at the fork point

Using Product Form Solver

- To deal with this, we provide the PF solver a modified set of populations
- $c_i' = \sum_{q=1}^{M} Q_{q,i}^{\text{init}}$
- where $Q_{q,i}^{\text{init}}$ is the initial approximation of mean queue length for queue q and class i
- Doing this scales $Q_{q,i}^{(PF)}$ to be a more accurate approximation

Efficiency

- Profiled code using MATLAB's built in profiler
- Investigate how runtime scales as we increase:
  - Number of fork-join queues
  - Length of synchronization queue

Efficiency - Profiling

Fork-Join Queues	Total Time (s)
2	2.412
3	5.536
4	35.643
5	814.599

Table: Increasing Number of Fork-Join Queues and Runtime

Sync Queue Length	Total Time (s)
1	4.501
2	8.092
3	14.925
4	33.174
5	88.88

Table: Increasing Synchronization Queue Length and Runtime

Efficiency - Resizing MMAP

- Generated MMAPs become extremely large as we increase the two factors
- Resize the MMAP once it becomes larger than a certain size
- We investigated resizing to MMAP with one state

Efficiency - Resizing MMAP

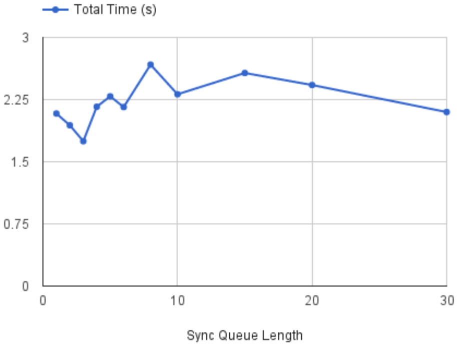
- Generated MMAPs become extremely large as we increase the two factors
- Resize the MMAP once it becomes larger than a certain size
- We investigated resizing to MMAP with one state

Efficiency - Resizing MMAP

- Generated MMAPs become extremely large as we increase the two factors
- Resize the MMAP once it becomes larger than a certain size
- We investigated resizing to MMAP with one state

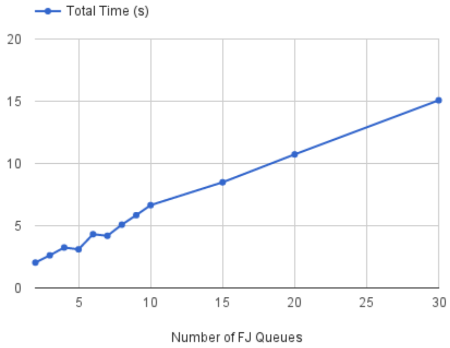
Efficiency - Resizing to single state MMAP

- We compute the rate for each class as $\lambda_c = \text{sum}(\pi \cdot D_{1,c})$
- where π is the equilibrium distribution of the MMAP
- So the new matrices are $D_{1,c} = [\lambda_c]$ and $D_0 = -\sum_{c=1}^{K} D_{1,c}$



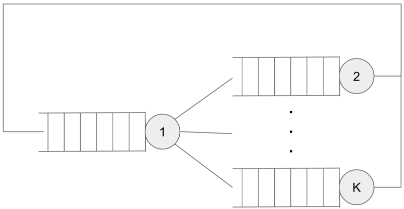
Efficiency - Resizing to single state MMAP

- Scales better - No discernible increase with synchronization queue length and approximately linear increase with number of FJ queues



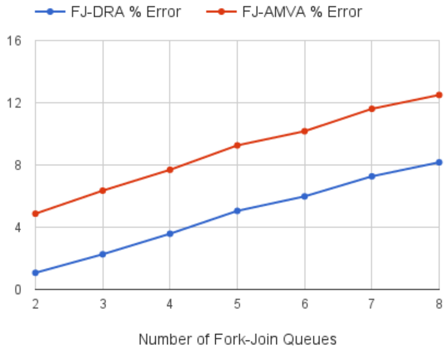
Results - Overview

- Compare our method against the FJ-AMVA method
- Vary the following factors:
  - Number of FJ queues
  - Heterogeneity of FJ queues
  - Complexity of service distributions
- Homogeneous FJ queues with exponential service distribution (unless directly testing that factor)



Results - Number of FJ Queues

- Our FJ-DRA method is more accurate than the FJ-AMVA method for all our tests
- Trend suggests that it will continue to be more accurate even as we increase the number of FJ queues
- Overall error increases as number of FJ queues increases



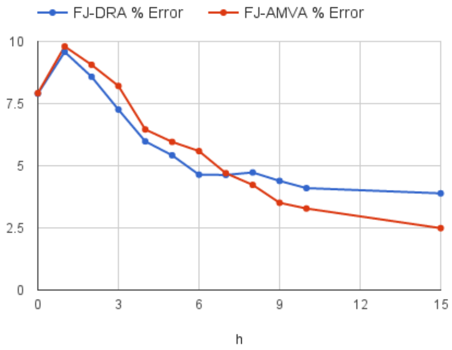
Results - Heterogeneity

We fixed the number of FJ queues at 4 and used the following parameters:

Parameter	Value
c_1	2
c_2	3
$\lambda_{1,1}$	1
$\lambda_{1,2}$	2
$\lambda_{k,1}$ $(\forall k \neq 1)$	1+0.1(k-1)h
$\lambda_{k,2}$ $(\forall k \neq 1)$	2+0.2(k-1)h

Results - Heterogeneity

- The performance of both approximation methods is quite close
- FJ-DRA is better at lower levels of heterogeneity
- Overall, error is trending down



Results - Erlang-2 Service Distribution

- Used Erlang-2 as service distribution for all queues
- Used 2 FJ queues
- Fix some parameters as in the table below:

Parameter	Value
$\lambda_{1,1}$	2
$\lambda_{1,2}$	4
c_1	1
c_2	2

Results - Erlang-2 Service Distribution

- Performance is quite close, FJ-DRA is slightly better in most cases
- Error is less than 5% for all tests
- No clear trend as λ's change

$\lambda_{k,1}$	$\lambda_{k,2}$	FJ-DRA % Error	FJ-AMVA % Error
2	4	4	4
3	4	3.12	3.16
4	4	2.63	4.36
3	5	2.76	3
3	6	2.56	3.11
4	8	3.85	3.39
5	10	4.27	3.68
6	12	4.23	3.72
10	20	3.57	3.18

Results - Summary

- Tested three factors:
  - Number of FJ Queues
  - Heterogeneity of FJ Queues
  - Complexity of service distributions
- In most of our tests, FJ-DRA is at least approximately as accurate as the FJ-AMVA method and there are some cases where FJ-DRA clearly outperforms FJ-AMVA
- Only tests where FJ-AMVA performs better is where FJ queues are very heterogeneous
- Overall, FJ-DRA method had less than 10% error in all our tests

Conclusion

- We presented the FJ-DRA method to approximate multiclass FJ queues
- We investigated how resizing the MMAP to a single state can help improve the efficiency
- We compared FJ-DRA against FJ-AMVA and found that it performs better in most of our tests

Questions?