

Gedae Runtime Kernel Performance Characterization

Kerry B. Barnes,
Gedae, Inc.
18000 Horizon Way, Suite 200
Mount Laurel, NJ 08054

Abstract

The objective of this work is to characterize the execution speed of the Gedae runtime kernel with the goal of determining how the kernel speed can be improved. The kernel was instrumented to measure the time of various functions including firing function boxes, changing schedule state, and managing dynamic queues. A set of benchmark graphs was chosen to exercise the runtime kernel. The results of the measurements made by running these graphs are presented. While preliminary, the results suggest several promising areas for performance enhancements.

Introduction

The objective of this work is to characterize the execution speed of the Gedae runtime kernel with the goal of determining how the kernel speed can be improved. The control functions implemented by the runtime kernel are not unique to Gedae and need to be implemented for most applications – often in an ad hoc manner. Because the Gedae runtime kernel abstracts these functions into a relatively small amount of code, Gedae provides the opportunity to focus on improving the efficiency of this code.

The runtime kernel controls the running of an application by scheduling function box execution, managing dynamic queues and propagating segment boundaries. The scheduling function can be divided into static scheduling and dynamic scheduling. In static scheduling the function boxes are fired in a predetermined order. Dynamic scheduling involves choosing which static schedule to fire based on availability of data in dynamic queues. All of these control activities are overhead to the main work of the graph, which is firing the function boxes that implement the graphs algorithm. A characterization method was created that provides a means for determining what

areas of the kernel have the highest overhead for a given application. The kernel characterization method has a number of benefits.

1. It focuses attention on where efficiency improvements are most needed.
2. It can be used to evaluate the effectiveness of kernel modifications.
3. Kernel execution times measured with the characterization method can be fed back into the Gedae performance simulator – GSIM – to provide accurate simulations of kernel overhead for different processor types.
4. End users of Gedae can use the characterization method to learn about the impact of various implementations of their graphs and also to provide feedback to the Gedae development team about needed performance improvements.

Method

The following method was used to characterize the runtime kernel:

1. A set of timer functions were created to allow accurately measuring the elapse time of different sections of the code.

Table1: Timer Calibration Measurements (usec)

Processor	Time	Mean	Stdv	Min	Max
Linux	Tpt	1.484	0.500	1.000	2.000
	Tt	0.801	3.957	0.000	125.0
NT	Tpt	3.616	0.391	3.352	9.499
	Tt	1.756	1.776	1.676	57.55
Solaris	Tpt	1.418	0.683	0.000	3.000
	Tt	0.473	0.499	0.000	1.000
AltiVec1	Tpt	1.440	0.017	1.399	1.962
	Tt	0.560	0.005	0.520	0.719
AltiVec2	Tpt	1.852	0.157	1.740	2.640
	Tt	0.746	0.127	0.660	1.500
AltiVec3	Tpt	1.316	0.026	1.295	1.680
	Tt	0.586	0.021	0.575	0.816

AltiVec1, AltiVec2, and AltiVec3 represent 3 target processors based on the AltiVec. Since these results are preliminary, we avoided naming the actual target processor type. Of the six processor types, AltiVec1 has the best uniformity for the timer times. The three AltiVec processor BSPs also have much more accurate implementations of the embWallclock function – all three have clock cycle accuracy, as opposed to the Unix processors that have 1usec accuracy, and the NT processor, which has an accuracy of 0.279 usec. For these reasons, the target processors provide better time estimates. In particular, AltiVec1 is used as the main source for conclusions in this paper.

Nested calls of the timer subtract the times from the nested calls from the higher level calls. This subtraction avoids double counting of times and allows the Gedae developers to focus on those sections of code that will provide the greatest performance improvement. The figure below shows how elapse times are calculated when there are nested time intervals. Note particularly that time T1 does not include the times T2 and T3 nor the time required to call the two nested timer routines.

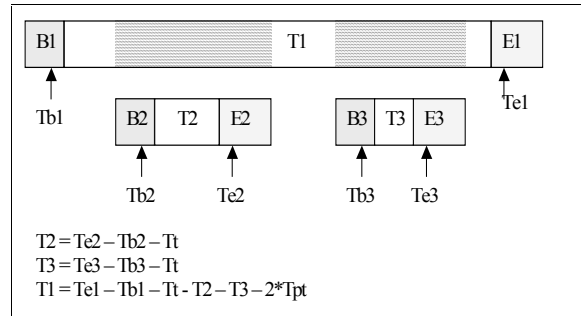


Figure 3: Example elapse time calculation

Instrumenting Code

The Gedae runtime kernel is constructed with natural places to add instrumentation to the code for measuring the time required for various state changes. State diagrams of the Static Scheduler, Dynamic Scheduler and Segment State Changes are shown below.

Static Scheduler Instrumentation

The static scheduler and some associated schedule functions are illustrated below. The run-schedule function, which runs the static schedule Apply methods, is the heart of the static scheduler. The function is implemented as a loop, with each pass of the loop calling the Apply method. If “Not-Done”, then another pass is made through the loop. The time to run-sched-pass is all the time required for one pass through the loop not counting the Apply method. The run-sched-pass time turns out to be one of the biggest contributors to the kernel overhead time.

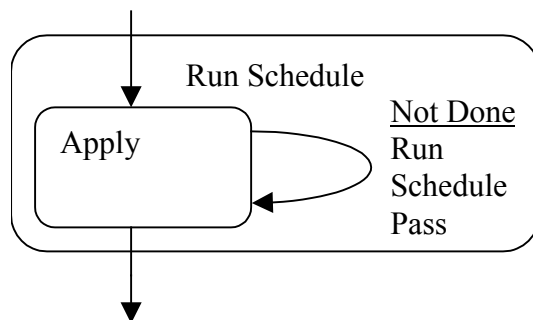


Figure 4: Static Scheduler

The following timers were added to the static scheduler:

1. Apply – measures the time to call the Apply methods of the graph
2. run-schedule-pass – the time to prepare to call an Apply method once
3. run-schedule – the time to enter and exit a static schedule

Dynamic Scheduler Instrumentation

The state diagram for the dynamic scheduler is illustrated below.

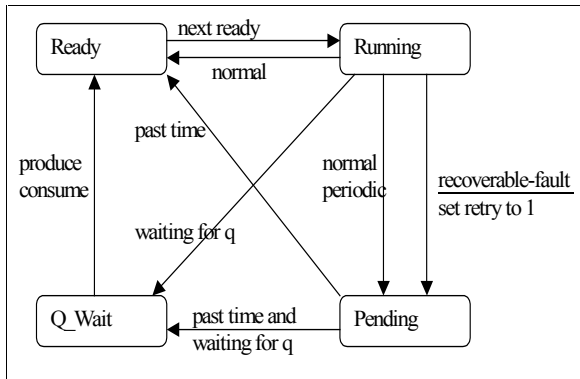


Figure 5: Dynamic Scheduler State Diagram

The following timers were added to measure the state transition times for the dynamic scheduler:

1. enter-ready-state
2. exit-ready-state
3. enter-running-state
4. exit-running-state
5. enter-pending-periodic
6. enter-pending-retry
7. exit-pending

Notably absent is the time to enter and exit the queue-wait state. Entering the queue-wait state merely involves setting the schedule state flag to queue-wait. Exiting

the queue-wait state is equivalent to entering the ready state.

Schedule state changes occur above and below the static scheduler. Above the static scheduler is the dynamic scheduler, which determines which static schedule to run next. Below the static scheduler are the produce and consume functions called from the Apply methods invoked by the static scheduler. The timers associated with these levels are described in the next two sections.

Dynamic Scheduler

The dynamic scheduler is the top-level function run in non-static Gedae applications. The function is illustrated below.

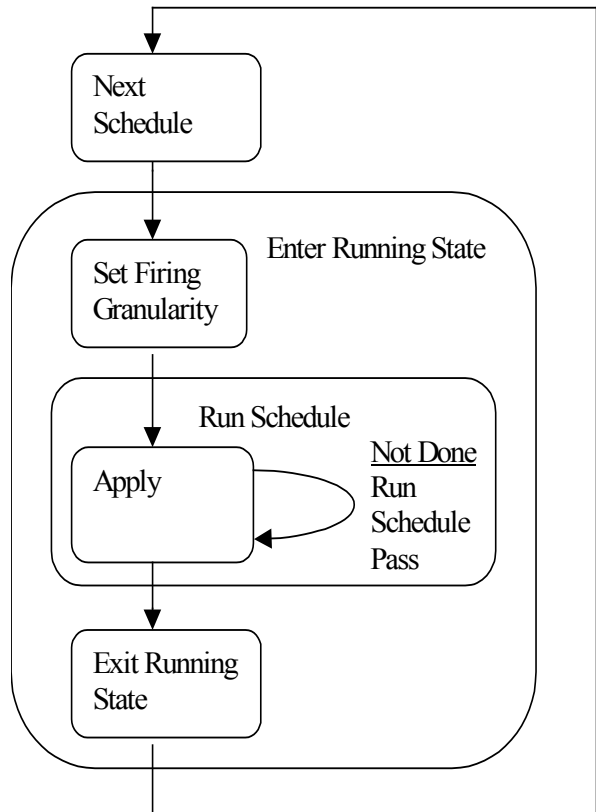


Figure 1: Dynamic Scheduler Execution

The following calling tree shows the timed dynamic scheduler events.

1. next-schedule – moves schedules from pending to ready and then gets next schedule off the ready queue
 - a. exit-pending
 - i. enter-ready
 - b. enter-ready
 - c. exit-ready
2. set-firing-granularity – the time required to change the firing granularity of a static schedule to handle the available queue data
3. run-schedule – the static scheduler as described above
4. exit-running-state
 - a. enter-pending-periodic
 - b. enter-ready
 - c. enter-pending-retry

Dynamic Queue Management

The Dynamic Scheduler is driven by data produced and consumed from function box Apply methods. Function boxes with dynamic queues must also prepare the data for use with the read-amount and write-amount functions. Thus, characterizing the queue management functions is an important part of characterizing the runtime kernel. In addition to affecting schedule state these functions also manage the data in the queues, therefore, they can involve some copying of data. For some applications, this copying can become a dominating time of the function overhead.

The events timed on dynamic queues are:

1. produce – produce data on a queue array – one queue for each destination queue
 - a. produce1 – produce data for one queue in the array
 - i. update-scheduler-on-produce – inform

- dynamic scheduler data was produced for schedule
 1. enter-ready-state – as described above
 - b. produce-copy – for queue arrays with more than one element copy data from first queue to remaining queues
2. write-dqs – called by dynenq box to move data from static schedule memory to dynamic queue array
 - a. ready-to-write – check if queue is ready to write
 - b. write-dq – handle single queue
 - i. write-dq-copy1 – if contiguous block is copied
 - ii. write-dq-copy2 – if block is split between beginning and end of queue
 - iii. produce1 – as described above
3. consume – consume data from a queue
 - a. update-scheduler-on-consume – inform dynamic scheduler data was consumed from static schedule output
 - i. enter-ready-state – as described above
4. read-dq – called by dyndeq box to move data from dynamic queue to static schedule memory
 - a. ready-to-read – check if queue is ready to read
 - b. read-dq-copy1 – if contiguous block is copied
 - c. read-dq-copy2 – if block is split between beginning and end of queue
 - d. consume – as described above
5. write-amount

- a. ready-to-write – as described above
- b. write-amount1 – prepare queue to write to
 - i. write-amount-copy – move data in queue to consolidate space
- 6. read-amount
 - a. ready-to-read – as described above
 - b. read-amount-copy – move data in queue to consolidate tokens

Segmentation

Segmentation is driven by segment-begin and segment-end events invoked from Apply methods on dynamic output queues of the box and also from the end-of-segment procedure of a schedule. The DynamicQueue state diagram for segmentation is illustrated below.

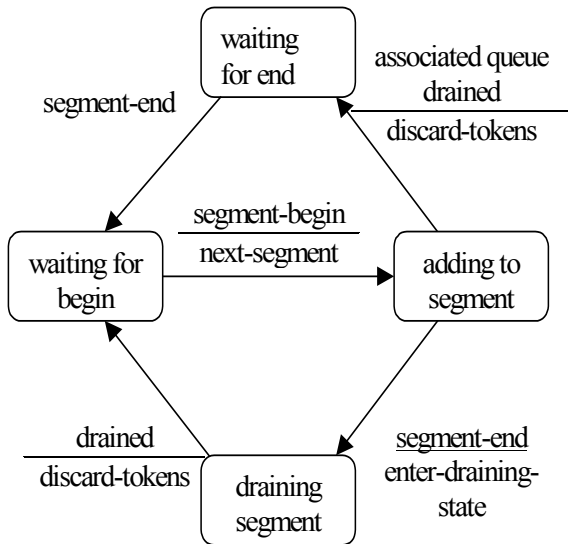


Figure 2: Dynamic Queue Segment State Diagram

As with schedule state changes, segment state changes can be caused by events above and below the static scheduler. Above the static scheduler is the modified enter-running-state function for

segmentation illustrated below. The modified functions contain the new blocks reset-segment – called at the beginning of segment processing – and end-of-segment, which is called when one of the input queues has reached the end-of-segment.

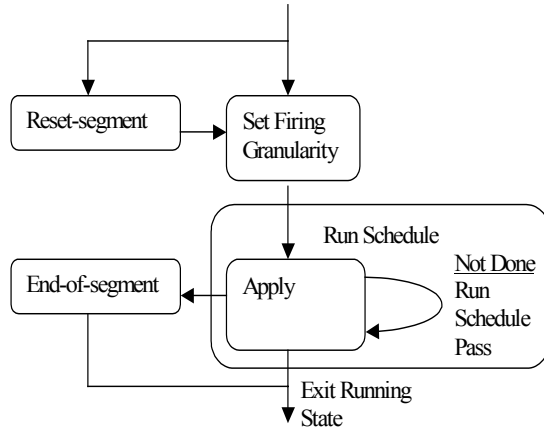


Figure 3: Enter Running State for Segmented Schedules

The new events that are timed are:

1. reset-segment – the time to run the segment reset procedure
 - a. reset-method – time to call a reset method
2. end-of-segment – the time to run the end-of-segment procedure
 - a. eos-method – time to call an eos method
 - b. destroy-method – time to call a destroy method
 - c. forward-eos
 - i. segment-begin
 - ii. segment-end
 - iii. forward-eos
 - d. discard-tokens
 - i. consume
 - e. next-segment
 - i. update-scheduler-on-produce
 - f. enter-draining-state
 - i. update-scheduler-on-produce

Below the static scheduler are the segment-begin and segment-end functions called from function box apply methods. These functions invoke the following timed state changes

1. segment-begin
 - a. next-segment
 - i. update-scheduler-on-produce
2. segment-end
 - a. enter-draining-state
 - i. update-scheduler-on-produce

Benchmark Graphs

Three graphs were chosen to benchmark the runtime kernel. Each graph exercises different aspects of the runtime kernel. The graphs are:

1. demo/stap/rt_stap_easy – this graph is statically scheduled, and therefore, provides a good measurement of the critical run-sched-pass code. The graph consists of 66 primitives
2. demo/comm/e_comm – this graph requires dynamic scheduling. It consists of 19 primitives broken into 4 static schedules. The static schedules are connected by 3 dynamic queues
3. training/tutorial/noise_removal – this graph requires dynamic scheduling with segmentation. The graph consists of 55 primitives, one of which implements an external state variable. There are 5 static schedules connected together with 5 dynamic queues. A distributor with an exclusive segmented output drives 3 exclusive branches of 16 boxes each. Each exclusive branch shares the single state variable.

Thus, the rt_stap_easy graph exercises the static scheduler, the e_comm graph exercises the static and dynamic schedulers,

and the noise_removal graph exercises static and dynamic schedulers and segmentation. The three graphs together provide a fairly representative sample of the types of applications run by Gedae. However, it is by no means a complete representation, and different graphs will undoubtedly show different areas.

Results

The table below shows the summary results for all 3 graphs running on the 6 different processor types

Table 1: See Attachment 1

Some things to note:

1. The Linux times for noise_removal are showing comparatively low efficiency. This is accounted for by the vx_rect2pol primitive, which is comparatively much faster on the Linux processor than on the other processor types where it is a major part of the execution time. Thus, the efficiency of the Linux processor is less than the Solaris and Nt processors, even though its firing overhead is lower.
2. More efficient doesn't necessarily mean better – for example, the 99.1% efficiency of the Linux processor on the rt_stap_easy problem is more reflective of the fact that the e_functions called from the Apply methods were not vectorized as was the case for the AltiVec processors. As a result, the Apply methods take proportionately more time, and the efficiency is higher.

Detailed results are presented below for AltiVec1. The 3 non-target processors all have multi-tasking OS's and poor clock resolution making their results less accurate. AltiVec1 showed consistently better standard deviations of the times (though any of the three AltiVec processors would also have been a good choice). The results are shown in the tables below.

The Max values are consistently much higher than the Min and Mean values and may skew the results. However, a histogram of the measured elapse times for each timer determined that the Max time events are rare enough not to be statistically important.

Tables 3-5: See Attachment 1

Conclusion

From the three benchmark graphs characterized, the following appear to be promising areas of performance improvement

1. run-sched-pass – the code required to fire an Apply method
2. write-dq-copy2, read-dq-copy1 – copy functions to manage data in the dynamic queues are a significant source of overhead in the e_comm and noise_removal graphs
3. the five dynamic schedule state change functions listed below are of comparable importance:
 - a. enter-running-state
 - b. exit-running-state
 - c. next-schedule
 - d. enter-ready-state
 - e. exit-ready-state
4. the queue management functions update-scheduler-on-produce and update-scheduler-on-consume are the interface between the queues and the schedule state

These 4 categories account for the majority of the overhead in all three graphs. The percentage of the overhead used by each category is:

Table 2: Percentage of Overhead Accounted for by Different Functions

Function	rt_stap _easy	e_comm	noise_removal
run-sched-pass	90%	20%	18%
copy functions		18%	9%
schedule state		36%	26%
queue management		10%	15%
Total	90%	84%	68%

Three ideas for improving the runtime kernel performance are:

1. Code generating multiple Apply methods into a single Apply method would reduce the cost of the run-sched-pass function by reducing the number of times it needs to be called.
2. Rather than copying data from dynamic queues into schedule memory as is done now, it is possible to directly manipulate the pointers in the Apply method state vectors to point directly into the dynamic queues.
3. The schedule state change functions all use the schedule-heap management functions. The two functions for managing the schedule-heaps, insertInSchedHeap and extractFromSchedHeap are candidates for optimization and may be added to the e_library for direct support in assembly code for the different BSPs.

The new method of characterizing the Gedae runtime kernel has been tested on three representative graphs. Large max times on all processors, and especially, Altivec 3 indicated further statistics – such as elapse time histograms – should be obtained with the goal of enhancing the method. Once the method is enhanced, more graphs should be added to the

benchmark suite to gain confidence in choosing the areas for performance enhancements. The method can easily be applied to other graphs and will be made generally available to users to do so. Users will be encouraged to share results of this characterization with the Gedae development team.

Attachment 1

Table 3: Summary Efficiency and Overhead Results

Graph: Proc	rt_stap_easy		e_comm		noise_removal	
	Efficiency	Overhead (sec/apply)	Efficiency	Overhead (sec/apply)	Efficiency	Overhead (sec/apply)
Linux	99.08%	1.454e-06	72.62%	4.717e-06	67.67%	9.270e-06
Solaris	97.63%	1.438e-05	79.43%	3.211e-05	89.73%	2.171e-05
Nt	99.46%	1.568e-06	79.43%	6.041e-006	89.15%	1.608e-05
Altivec1	96.70%	1.692e-06	90.50%	2.731e-06	95.14%	1.780e-06
Altivec2	93.40%	1.138e-06	92.16%	2.838e-06	91.83%	2.033e-06
Altivec3	99.76%	0.421e-06	96.17%	1.873e-06	89.6%	2.014e-06

Table 4: rt_stap_easy Characterization on Altivec 1

Timer	Called	Total	Mean	Stdv	Min	Max	Perc	PerFire
apply	7592362	3.761e+02	4.954e-05	7.821e-03	3.600e-07	2.428e+00	96.70%	4.954e-05
run-sched-pass	7592361	1.156e+01	1.523e-06	8.573e-07	8.000e-08	2.205e-04	2.97%	1.523e-06
run-schedule	7346338	1.283e+00	1.747e-07	6.751e-07	1.200e-07	1.950e-04	0.33%	1.690e-07

Table 5: e_comm Characterization on Altivec 1

Timer	Called	Total	Mean	Stdv	Min	Max	Perc	PerFire
apply	667628	1.737e+01	2.601e-05	2.177e-03	1.590e-07	7.901e-01	90.50%	2.601e-05
run-sched-pass	821288	3.649e-01	4.443e-07	1.167e-06	3.900e-08	1.502e-04	1.90%	5.465e-07
read-dq-copy1	102408	3.392e-01	3.313e-06	5.552e-05	1.990e-07	1.294e-03	1.77%	5.081e-07
exit-running-state	153660	1.693e-01	1.102e-06	1.207e-06	5.990e-07	1.312e-04	0.88%	2.535e-07
next-schedule	153660	1.669e-01	1.086e-06	1.184e-06	8.800e-07	1.316e-04	0.87%	2.500e-07
enter-running-state	153660	1.285e-01	8.365e-07	1.144e-06	5.610e-07	1.320e-04	0.67%	1.925e-07
exit-ready-state	153660	1.026e-01	6.676e-07	3.410e-07	3.990e-07	1.123e-04	0.53%	1.536e-07
update-on-consume	102408	9.840e-02	9.609e-07	2.084e-07	6.390e-07	3.504e-05	0.51%	1.474e-07
update-on-produce	102589	9.652e-02	9.408e-07	1.157e-06	5.200e-07	1.173e-04	0.50%	1.446e-07
enter-ready-state	153661	8.908e-02	5.797e-07	5.875e-07	3.990e-07	1.153e-04	0.46%	1.334e-07
run-schedule	153660	4.083e-02	2.657e-07	1.420e-06	1.200e-07	1.270e-04	0.21%	6.116e-08
produce 1	102589	3.427e-02	3.341e-07	1.014e-06	2.390e-07	1.241e-04	0.18%	5.133e-08
write-amount	102609	3.357e-02	3.272e-07	8.316e-07	2.390e-07	1.320e-04	0.17%	5.028e-08
exit-pending-state	307320	3.192e-02	1.039e-07	5.119e-07	7.900e-08	1.140e-04	0.17%	4.781e-08
consume	102408	3.122e-02	3.048e-07	5.114e-07	2.390e-07	1.042e-04	0.16%	4.676e-08
read-dq	102408	2.087e-02	2.038e-07	3.995e-07	8.000e-08	1.088e-04	0.11%	3.126e-08
write-amount 1	102609	1.934e-02	1.885e-07	1.016e-07	3.900e-08	1.228e-05	0.10%	2.896e-08
ready-to-write-req	102609	1.838e-02	1.791e-07	6.027e-07	1.590e-07	1.133e-04	0.10%	2.753e-08
produce	102589	1.546e-02	1.507e-07	4.925e-07	7.900e-08	1.157e-04	0.08%	2.316e-08
ready-to-read-req	102408	1.386e-02	1.353e-07	5.614e-08	1.190e-07	4.160e-06	0.07%	2.076e-08
set-granularity	153660	6.498e-03	4.229e-08	4.060e-07	3.900e-08	1.127e-04	0.03%	9.734e-09
write-amount-copy	570	1.530e-03	2.684e-06	7.003e-06	5.210e-07	1.212e-04	0.01%	2.292e-09

Table 6: noise_removal Characterization Results on AltiVec 1

Timer	Called	Total	Mean	Stdv	Min	Max	Perc	PerFire
apply	425702	1.484e+01	3.486e-05	5.015e-05	2.800e-07	2.353e-04	95.14%	3.486e-05
run-sched-pass	474822	1.393e-01	2.934e-07	5.873e-07	4.000e-08	3.880e-05	0.89%	3.272e-07
write-dq-copy2	16373	7.289e-02	4.452e-06	4.078e-07	4.080e-06	3.484e-05	0.47%	1.712e-07
read-dq-copy1	32746	6.523e-02	1.992e-06	1.769e-06	2.000e-07	3.468e-05	0.42%	1.532e-07
update-on-consume	82376	6.122e-02	7.432e-07	3.350e-07	4.400e-07	1.676e-05	0.39%	1.438e-07
update-on-produce	49632	5.646e-02	1.138e-06	4.279e-07	5.600e-07	4.243e-06	0.36%	1.326e-07
next-schedule	49120	5.559e-02	1.132e-06	3.442e-07	8.420e-07	3.209e-05	0.36%	1.306e-07
exit-running-state	49120	4.133e-02	8.415e-07	2.710e-07	4.400e-07	2.084e-05	0.26%	9.710e-08
enter-running-state	49120	3.974e-02	8.091e-07	2.565e-07	4.420e-07	3.068e-05	0.25%	9.336e-08
exit-ready-state	49120	3.105e-02	6.322e-07	2.431e-07	3.600e-07	3.112e-05	0.20%	7.295e-08
enter-ready-state	49121	2.959e-02	6.023e-07	2.089e-07	3.600e-07	1.628e-05	0.19%	6.950e-08
consume	82376	2.439e-02	2.960e-07	2.039e-07	1.200e-07	3.060e-05	0.16%	5.728e-08
produce1	49121	2.104e-02	4.284e-07	1.169e-07	2.800e-07	1.548e-05	0.13%	4.943e-08
run-schedule	49120	1.814e-02	3.694e-07	4.208e-07	1.220e-07	3.138e-05	0.12%	4.262e-08
ready-to-read-nreq	81865	1.485e-02	1.814e-07	8.084e-08	1.200e-07	1.904e-05	0.10%	3.489e-08
write-amount	32748	1.457e-02	4.448e-07	2.362e-07	2.420e-07	3.104e-05	0.09%	3.422e-08
write-dqs	16373	1.322e-02	8.075e-07	2.745e-07	6.820e-07	3.140e-05	0.08%	3.106e-08
ready-to-write-nreq	49121	1.121e-02	2.283e-07	5.523e-08	1.600e-07	2.760e-06	0.07%	2.634e-08
read-dq	32746	1.022e-02	3.122e-07	1.851e-07	1.600e-07	3.056e-05	0.07%	2.402e-08
exit-pending-state	98240	1.013e-02	1.031e-07	7.109e-08	8.000e-08	1.536e-05	0.06%	2.379e-08
read-amount	49119	8.092e-03	1.647e-07	1.458e-07	8.000e-08	3.052e-05	0.05%	1.901e-08
write-dq	16373	5.797e-03	3.541e-07	4.449e-08	2.410e-07	2.803e-06	0.04%	1.362e-08
produce	32748	4.724e-03	1.443e-07	1.950e-07	8.000e-08	3.080e-05	0.03%	1.110e-08
set-granularity	49120	2.415e-03	4.916e-08	2.624e-08	4.000e-08	2.281e-06	0.02%	5.672e-09
write-amount1	32748	1.975e-03	6.031e-08	8.709e-08	4.000e-08	1.520e-05	0.01%	4.639e-09
end-of-segment	511	1.187e-03	2.323e-06	6.118e-08	2.162e-06	2.642e-06	0.01%	2.788e-09
next-segment	512	1.033e-03	2.017e-06	6.960e-08	1.881e-06	2.561e-06	0.01%	2.426e-09
reset-segment	514	6.417e-04	1.248e-06	1.358e-07	1.161e-06	3.685e-06	0.00%	1.507e-09
segment-end	511	5.855e-04	1.146e-06	2.947e-08	1.080e-06	1.242e-06	0.00%	1.375e-09
segment-begin	512	5.469e-04	1.068e-06	4.514e-08	9.210e-07	1.362e-06	0.00%	1.285e-09