Group 16 - Adaptive Executor Service

Table of contributions

	Buster Major – bmaj406	**Karan Nellisserry - knel697**	**Timothy Finucane – tfin440**
Research	80%	10%	10%
Presentation	33%	33%	33%
Implementation	35%	20%	45%
Report	33%	33%	33%

1 Introduction

An executor service is a programming interface provided by the Java programming language allowing a client to submit tasks that may be run in parallel and await their completion in an abstract manner without direct reference to threading or multiprocessing [1]. The Java Standard Library provides multiple implementations of this interface.

This report investigates the viability and performance of an adaptive implementation of an Executor Service that continuously adjusts itself to the system's runtime load. Areas such as thread pool size and queue usage are considered for potential variation in the Executor Service. Various techniques are examined, with some implemented and evaluated.

Secondly, A benchmarking suite is created to compare various implementations and techniques against each other and against Java standard implementations. This benchmarking suite considers many different metrics to gain a system-wide view on the performance, efficiency, and usefulness of the Executor Service. Measurements such as system load and responsiveness of separate, consecutively running threads are gathered to get a system-wide perspective, whilst other measurements such as mean and max task completion time are used for comparison of performance.

2 Metrics

Metrics allow us to objectively compare our executor service implementation to existing Java implementations. Important things to measure include the time taken to execute individual tasks, the resources used by the service and the impact on other applications running on the system. These metrics, based on how they are gathered, are divided into two classes: local and global metrics.

2.1 Local Metrics

Local metrics refer to metrics gathered on individual tasks. They include the amount of time a task spends after submittal and before starting (idle time), from submittal to completion (completion time) and from starting to completion (processing time). Task processing times indicate the service's general efficiency in running tasks, idle time can indicate 'fairness' of the service in processing tasks in order, and processing time relates to how many system resources there are devoted to each task, i.e. how many tasks are running simultaneously (indicating the service's level of parallelism). From a user's point of view, local metrics gain insight into how 'fast' the service is at doing the work it is given.

2.2 Global Metrics

Global metrics refer to metrics gathered on a system wide level. Resource-based metrics are CPU usage, how much processing power the service draws, and the number of threads, which gives an idea of memory and CPU time footprint. Analysing these can tell us how efficiently and effectively our service uses the resources at its disposal. To gather information on the impact of a service on the wider system, ‘responsiveness’ can be simulated via measuring the amount of (fixed-size) work done by an auxiliary application and observing the change in this measurement as the service faces heavier loads. Responsiveness indicates the executor service’s impact on real-time applications, such as GUIs.

3 Techniques

There are three main variables we can alter in an executor service, each of which has a different impact on a different aspect of performance. These are thread count, task queue count and scheduling policy.

3.1 Thread Count

The number of threads in an executor service impacts performance greatly. The more threads a service contains, the more work can be performed concurrently in that service. A low thread count leaves processors unused, especially if some tasks are IO bound. A high thread count results in overheads from context switching, additional memory usage, and more contention of shared resources. For an Executor Service handling dynamic loads, the number of threads should be proportional to load, to reduce potential unnecessary resource usage when load is low, and to exploit available parallelism when load is high.

3.1.1 Watermark Model

This model constrains the number of active threads between a *value low* and *value high* count and is otherwise equal to the number of tasks left to process. Thread creation overhead is reduced through the *value low* watermark level, guaranteeing there will be at least *n* threads in the service, meaning load spikes likely result in less thread creation. Furthermore, overhead and resource issues stemming from having too many threads are negated by the *value high* watermark acting as a maximal thread count [1, 2].

3.1.2 Exponential Moving Average for Thread Prediction

It would be useful to be able to create threads before they are needed. E.g. as load begins to spike, the executor service should pre-emptively increase the thread count. This reduces the overhead of thread creation on task submittal and decreases the waiting time for a task to begin work. One way of doing this is using a history of previous loads experienced by the system to predict load in the immediate future. This can be implemented in a computation called an exponential moving average (EMA), described in Figure 2. However, as this computation only considers values in the present or past, it will lag behind current changes to the load. To rectify this imbalance, a term for the current change in load can be introduced to predict a future value as in Figure 3 [1, 2]. Figure 1 shows that regular EMA underestimates thread count and the modified EMA produces much closer values.

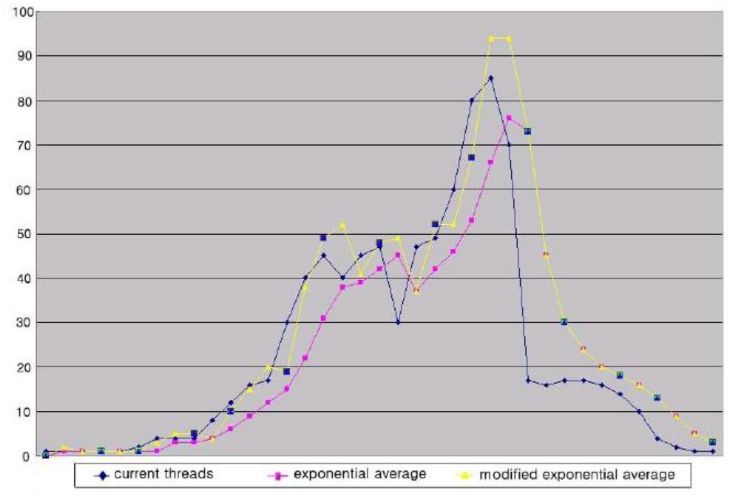

Figure 1 - Comparison of thread count, EMA, and modified EMA [4]

$$S_n = \begin{cases} t_1, & \text{if } n = 1 \\ \alpha \cdot t_n + (1-\alpha) \cdot S_{n-1}, & \text{if } n > 1 \end{cases}$$

Figure 2 - EMA calculation

$$S'_{n+1} = \begin{cases} t_n + (t_n - S_{n+1}), & \text{if } S_{n+1} > S_n \\ S_{n+1}, & \text{otherwise} \end{cases}$$

Figure 3 - EMA with predictive

3.2 Task Queue Count

The number of task queues in an Executor Service has a significant impact on performance. There are two ends of the spectrum of Executor Service queue architecture.

A centralized queue architecture means a service will have one task queue shared between all running threads. Due to concurrent accesses, this queue must be thread safe, introducing contention. However, the load is perfectly balanced between threads. This approach is effective for task streams with high size variance (need for load balancing) and fewer tasks (less queue access contention).

Conversely, a completely distributed queue architecture means a service will have a one-to-one mapping between threads and task queues. Queues do not need to be thread safe as there is no contention; however, load balancing is poor as some queues can run out of tasks before others, resulting in thread idling. This approach is effective for task streams with low size variance (no need for load balancing) and many tasks (many queue accesses).

A third approach, of having a number of queues greater than 1 but less than the number of threads, allows for a medium between the two architectures which can potentially gain benefits from both. In this case there will be a mapping where multiple threads map to one queue each.

3.2.1 Work Stealing

Work stealing is a modification to the distributed queue architecture which can exploit the benefits of load balancing and low queue contention. Each thread requests a task from its mapped queue, if the queue is empty the thread will then search other queues for tasks. This behaviour of 'stealing work' provides the load balancing benefits which distributed queue architectures lack. Although these distributed queues also need to be thread safe because of potential contention, there is low likelihood of multiple threads accessing the same queue at the same time.

3.3 Scheduling Policy

Task scheduling refers to the decision making when assigning incoming tasks to various task queues. The main scheduling policies considered are round-robin scheduling, random queue scheduling and queue with least tasks scheduling (an approximation of earliest start time scheduling using only number of tasks).

4 Implementation

The implementation consists of a series of techniques that can be 'plugged in' to a generic Executor Service, and a Benchmarking suite capable of gathering metrics for these Executor Services as well as Java Standard Library Executor Services.

4.1 Benchmarking

The benchmarking system works by running executor service implementations against certain profile types (simulations of workload) and gathering metrics based on this running. A profile generates a sequence of tasks, that may be placed over time, which are then submitted to the service. For each task, the time of submitting (when *ExecutorService.submit* is called), starting, and finishing are recorded.

Simultaneously, a watcher thread gathers other metrics, such as thread count and CPU usage. System responsiveness is simulated by running simple tasks in an additional thread and counting how many tasks are completed per time step. A drastic change in this value indicates the thread has been paused for a significant amount of time due to high system usage by the Executor Service. As such, responsiveness (including graph on 8.2.2) is measured in variance of this task count over time.

On completion of all the tasks generated by the profile, all metrics are serialized into JSON and visualized via python scripts.

4.1.1 Profiles

Four profiles are implemented to simulate four different use-cases of an Executor Service. Each type can simulate a static load (all tasks submitted at once and then executed) or a dynamic load (tasks are dispatched at specific times). Computational tasks are simulated by iteratively calculating the value of Pi, for a given number of iterations. IO tasks sleep the thread for a short period of time, to simulate non-CPU bound behaviour.

The benchmarking functionality uses random distributions for features like task size generation and task submittal time generation to prevent the system from exploiting any regularity. It is important that the benchmarking software is repeatable, so the *org.apache.commons* package, with the ability to generate values in a distribution from a separate random generator, is used to ensure that two separate Executor Services can be tested with the exact same set of tasks [2].

The Uniform Profile simulates a very regular load, with task sizes following a tight normal distribution. This size variance prevents the Java runtime from optimizing away repeated runs of the task. It may create a static or dynamic load and produces only computational tasks. This profile is useful for simulating a constant load of work on an executor service, which is important because it can help us understand how well our executor service optimizes queue contention and work stealing.

The irregular profile produces computational tasks with sizes that lie within a bimodal distribution, where each peak is an order of magnitude apart. This can be performed either statically or dynamically. An Irregular Profile is used to simulate an application with a highly variable of task sizes. It tests an executor service's quality of load balancing. In other words, if a series of irregular sized tasks are dispatched, ideally an executor service should be able to ensure task queues stay relatively balanced. This balance is measured by task slack time or thread idle/contention time.

The dynamic load profile dispatches tasks dynamically at a changing rate. A number of 'peaks' are specified, which result in areas of more frequent and less frequent task submittal concentration. These peaks simulate a web service, receiving requests at a varying rate. The profile tests the executor service's adaptability to changes in load. This adaption could come in the form of changing thread numbers or altering queue structure, and is measured by thread count, and maximum task completion time.

The IO profile dispatches both computational and IO tasks, at a given ratio. Like the Uniform Profile, these tasks are regular with a tight normal distribution. This simulates applications which perform IO work, such as web services that access a database, GUI applications reading and writing to the filesystem, etc. Importantly, IO-bound tasks will consume a thread despite not using the CPU, and as such a perfect Executor Service will increase the number of threads to ensure the CPU is fully utilised, else suffer from large wait times.

4.1.2 Challenges

While building and testing the benchmarking system we encountered unusual behaviour. Earlier in development, we found that the time taken for *any* service to complete an individual randomly sized dispatched task would follow the expected curve for several seconds and then instantly drop to a near-zero amount of time for the remainder of the services execution (see Figure 5 and Figure 4). At the time our profiles would utilise computational tasks which calculate the value of Pi to some degree of accuracy, however their values would be immediately discarded. We realized that it was this behaviour causing the steep drop in processing 'effort' by the Java runtime. We hypothesize that the Java runtime would run the tasks on the service for a period, and by warming up it would 'realize' after several seconds that results from the tasks were not being used nor were the tasks writing to other areas of memory, so it would simply stop doing the tasks in an attempt to optimize performance. We believe this is evidence of a feature of JIT compilation optimization, where code the JVM can remove segments of code it deems 'unnecessary' [3]. To work around this, we recorded the output of every calculation task and printed an average of all values upon completion.

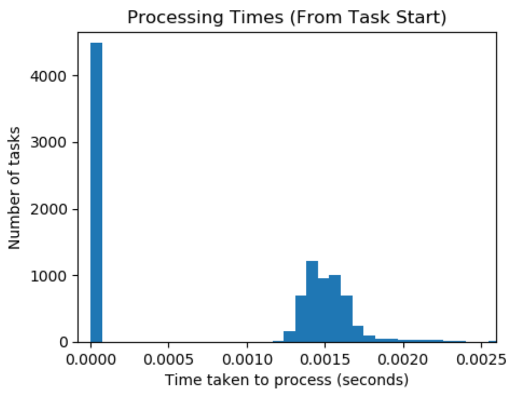

Figure 4 - ~10 seconds into dynamic load dispatch with 5 intended peaks, JIT optimizes tasks

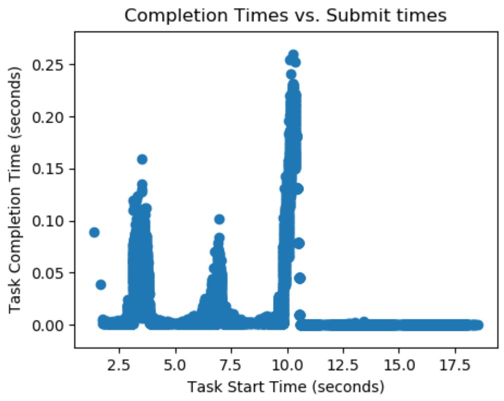

Figure 5 - JIT compilation resulting in ~45% of tasks being completed in almost zero time

4.2 Executor Service

The Executor Service consists of a modular core with two replaceable parts: a *ThreadManager*, which determines when new threads are created and when existing threads are deleted, and a *TaskManager*, which opaquely stores tasks and allows threads to retrieve them. Each manager has two implementations, for each of the techniques listed above.

The *WatermarkPredictor* is straightforward, and simply ensures the thread count is between two given values. In our implementation we have a minimum at the number of cores, and a maximum at twice that, which ensures a good range with minimal contention. The second thread manager implements the EMA prediction via a watcher thread that keeps track of how many tasks are in existence at a certain sampling rate. Rather than ignoring the current rate of change in tasks if the change is negative, our implementation uses a minimum value (like the watermark predictor), also equal to the number of CPU cores.

Our two task managers are a simple shared queue implementation, that stores and retrieves all tasks from a single queue, and a fixed queue implementation, which uses a fixed number of queues, determined on initialization and usually set to the number of cores (to give a one-to-one mapping when threads are at their minimum). Queues are evenly mapped to threads. A fixed number of queues was chosen over a fixed ratio or other style for the sake of efficiency and ease of implementation. As the number of queues doesn't change at runtime, no locking needs to take place for a thread when retrieving a task from its queue. Finally, tasks are added in round robin style. Round robin is the fastest out of the three methods discussed, whilst still being relatively fair if there are many tasks or if tasks are regular.

Due to time constraints, the *shutdownNow* method of this Executor Service was not implemented. This has no impact on the benchmarks as the benchmarking application does not make use of this method, and the cost to service efficiency of implementing this method would be negligible. Options for configuring the executor service are simple and provided via static method constructors. More consideration could be put into this interface in future work.

5 Results

Executor Services were run on 5 Profiles. Those executor services are: Java's Fixed Thread Pool (with no. of threads equal to number of CPU logical cores), Java's Fork Join Pool, and Java's Work Stealing Pool (each with no additional settings), and every combination of this project's Executor Service, with Watermark or EMA thread prediction and shared or fixed queues[1]. The 5 profiles are:

- UNIFORM: A uniform and dynamic load, with few tasks of medium size. Not stressful.
- PEAKED: A dynamic load profile with a large number of tasks, to stress task management
- IO: An IO Profile with few tasks, but 20% are IO-Bound. Tests adaptation to non-CPU-bound tasks.
- IRREGULAR: An irregular profile with few tasks of medium-large size, to test balancing.
- STATIC: A uniform static load, medium number of tasks that are very large. Tests system stress.

Tests on multiple machines produced results in favour of this project's Executor Service, with mean and max task completion time outperforming their Java counterparts. Some expected results are the increased efficiency of EMA on the peaked profile, as the change in tasks as a peak arrives is very easy for EMA to predict, the benefit of Watermark creating many threads for IO tasks in the IO profile, and the superiority of a shared queue in Uniform, where there are very few tasks (so misbalancing can occur easily). *SharedQueue-EMA* is the worst of the Group16 bunch 3 times, likely due to the increased contention EMA introduces on a shared queue by pre-emptively creating threads. An unidentified trend is the superiority of *FixedQueue-Watermark* on the Irregular load, when the initial expectation was that a shared queue could load balance more effectively.

This project's services perform worst in Uniform, where there is low stress. This is likely due to increased overheads in the Group16 services due to the algorithms used and the modular design. CPU Usage is in line with other services, with small enough variations as to not be significant. Responsiveness, however, is worse in the Group16 Executors. This is unfortunate and would need to be investigated further to arrive at a solution.

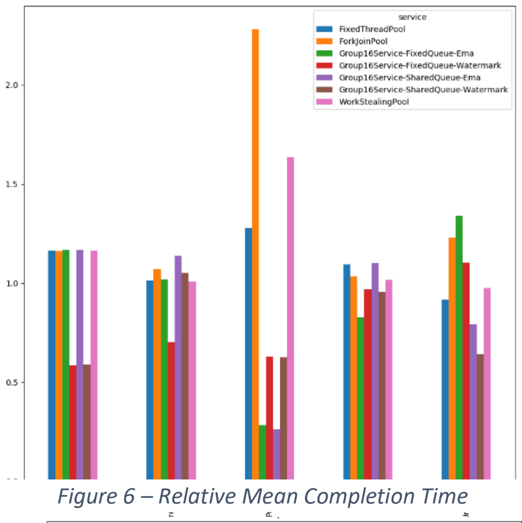

Figure 6 – Relative Mean Completion Time

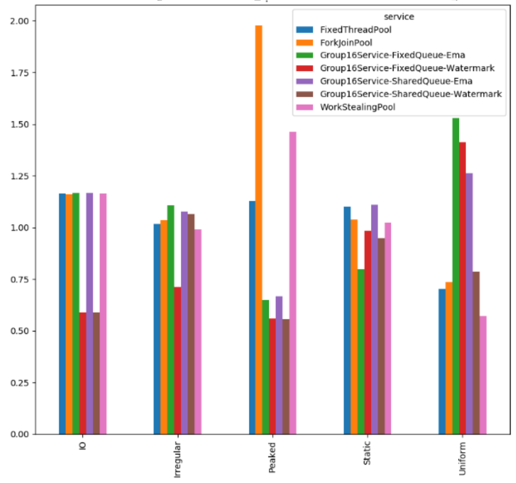

Figure 7 – Relative Maximum Completion Time

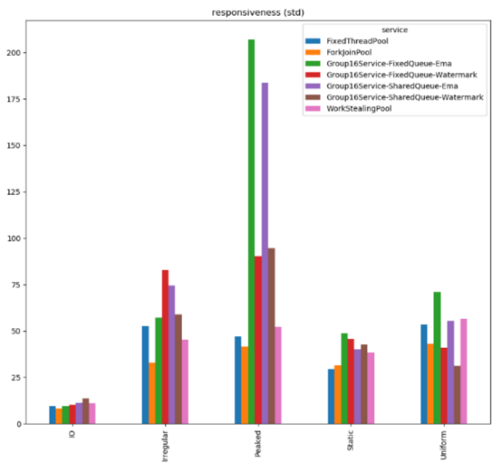

Figure 8 - System Responsiveness

[1] For more information about the exacts of each profile, see 8.1

6 Conclusion

Research was conducted upon thread pool efficiency, specifically in the area of thread management and queue management. Key techniques in these areas were implemented as Java Executor Services. These Executor Services were then compared against various java standard executor service implementations via custom benchmarking software. Metrics for efficiency, system load, and responsiveness were identified and used for comparison.

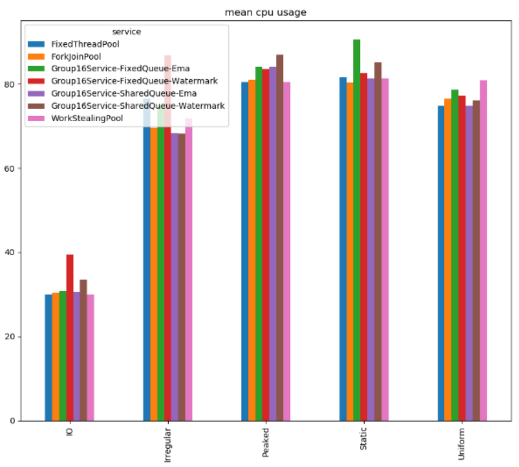

Figure 9 - Mean CPU Usage

Measurements show that the custom Executor Services improve upon Java's offerings in the majority of applications, with minimal or no increase in system load. Specifically, major improvements are made where loads are either highly irregular or vary in throughput. System responsiveness however is worsened, indicating that these services may not be suitable for GUI applications or for personal systems.

7 References

[1] Oracle, “java.util.concurrent Interface ExecutorService,” 2018. [Online]. Available: https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ExecutorService.html.

[2] Apache Commons, “org.apache.commons.math3.random Interface RandomGenerator,” 2014. [Online]. Available: https://commons.apache.org/proper/commons-math/javadocs/api-3.4/org/apache/commons/math3/random/RandomGenerator.html.

[3] Oracle, “Understanding JIT Compilation and Optimizations,” 2016. [Online]. Available: https://docs.oracle.com/cd/E13150_01/jrockit_jvm/jrockit/geninfo/diagnos/underst_jit.html.

[4] D. Kang, S. Han and S. Yoo, “Prediction-Based Dynamic Thread Pool Scheme for Efficient Resource Usage,” in *IEEE 8th International Conference on Computer and Information Technology Workshops*, 2008.

[5] H. Linfeng, G. Yuhai and W. Juyuan, “Design and implementation of high-speed server based on dynamic thread pool,” in *13th IEEE International Conference on Electronic Measurement & Instruments*, 2017.

[6] K.-L. Lee, H. N. Pham, H.-s. Kim, H. Y. Youn and O. Song, “A Novel Predictive and Self -- Adaptive Dynamic Thread Pool Management,” in *IEEE Ninth International Symposium on Parallel and Distributed Processing with Applications*, Busan, 2011.

[7] M. Hogmann and G. Runger, “Towards an adaptive task pool implementation,” in *IEEE International Symposium on Parallel and Distributed Processing*, Miami, 2008.

8 Appendices

8.1 Repository link

All code, including scripts for generating graphs and statistics from gathered metrics is located within https://github.com/TimFinucane/executor-benchmark (private) and available to Oliver Sinnen immediately. Any other people wishing to gain access can email tfin440@aucklanduni.ac.nz to be added.

8.2 Tabled benchmarking results

Benchmarks were run on two computers, with details specified below. The above graphs (figures 6 through 9) were generated solely from Computer 1 results.

Computer 1: Windows 10 x86-64, Intel Core i5-4690 (3.50 GHz, 4 cores), 16GB RAM.
Computer 2: Windows 10 x86-64, Intel Core i7-7500 (2.70 GHz, 4 cores), 8GB RAM.

8.2.1 Normalized task completion time mean/maximum[2]

Profile	Computer	Peaked		IO		Irregular		Uniform		Static		Peaked	
		Mean	Max.	Mean	Max.	Mean	Max.	Mean	Max.	Mean	Max.	Mean	Max.
Fixed Thread Pool	1	1.2769	1.1282	1.1648	1.1641	1.0121	1.0163	0.9184	0.7037	1.0951	1.1014	1.2769	1.1282
	2	1.3486	0.9471	1.1623	1.1616	1.0422	1.0335	1.031	0.828	1.0372	1.031	1.3486	0.9471
Fork Join Pool	1	2.2831	1.977	1.1621	1.1607	1.0706	1.0351	1.2279	0.7344	1.0337	1.0368	2.2831	1.977
	2	2.0734	1.6729	1.1626	1.162	1.0902	1.0726	0.7334	0.6394	1.0641	1.0667	2.0734	1.6729
FixedQueue Ema	1	0.2841	0.6481	1.1664	1.1673	1.0192	1.1054	1.3401	1.5287	0.8264	0.7998	0.2841	0.6481
	2	0.3851	1.0143	1.1699	1.1731	1.0326	1.0248	1.5692	2.1501	1.0231	1.0284	0.3851	1.0143
FixedQueue Watermark	1	0.6307	0.5599	0.5848	0.5877	0.7014	0.712	1.1039	1.4131	0.9697	0.9829	0.6307	0.5599
	2	0.4359	0.7355	0.5947	0.5918	0.7844	0.8319	1.3231	1.2603	0.8844	0.8921	0.4359	0.7355
haredQueue Ema	1	0.2618	0.6672	1.1676	1.1653	1.1393	1.0766	0.7931	1.2625	1.1017	1.1098	0.2618	0.6672
	2	0.2893	0.772	1.1626	1.1636	1.0403	1.0282	0.8182	0.7223	1.0178	1.0163	0.2893	0.772
G16 SharedQueue Watermark	1	0.6274	0.6274	0.5898	0.5898	1.0506	1.0506	0.6397	0.6397	0.9558	0.9558	0.6274	0.6274
	2	0.3912	0.3954	0.5847	0.5854	0.9232	0.9333	0.6418	0.6857	0.9022	0.8976	0.3912	0.3954
Work Stealing Pool	1	1.636	1.4632	1.1644	1.1645	1.0068	0.9889	0.9769	0.5721	1.0176	1.0216	1.636	1.4632
	2	2.0765	1.4629	1.1632	1.1625	1.0871	1.0756	0.8832	0.7142	1.0712	1.0679	2.0765	1.4629

8.2.2 Responsiveness measures[3]

Profile	Computer	Peaked	IO	Irregular	Uniform	Static	Peaked
Fixed Thread Pool	1	47	9.6	52.5	53.3	29.5	47
	2	90.6	22.1	27.5	40	22.3	90.6
Fork Join Pool	1	41.6	8.2	33	43.1	31.5	41.6
	2	98.6	22.7	23.5	34.7	28.8	98.6
G16 FixedQueue Ema	1	207.2	9.5	57.2	71	48.7	207.2
	2	252.8	46.3	22.2	62.6	31.4	252.8
G16 FixedQueue Watermark	1	90.4	10.2	82.9	41.1	45.7	90.4
	2	131.1	59.8	27.3	50.6	20.8	131.1
G16 SharedQueue Ema	1	183.5	11.2	74.5	55.3	40	183.5
	2	259.9	34.8	33.4	44.4	21.8	259.9
G16 SharedQueue Watermark	1	94.5	13.5	58.8	31.1	42.7	94.5
	2	126	45	43.1	36.2	19.6	126
Work Stealing Pool	1	52.3	10.9	45.3	56.4	38.5	52.3
	2	84.3	21.1	37.8	71.5	35.1	84.3

8.2.3 CPU Utilisation[4]

Profile	Computer	Peaked	IO	Irregular	Uniform	Static	Peaked
Fixed Thread Pool (%)	1	80.41	30.02	76.45	74.77	81.6	80.41
Fork Join Pool (%)	1	81.04	30.34	69.62	76.52	80.29	81.04
G16 FixedQueue Ema (%)	1	84.16	30.86	74.69	78.62	90.57	84.16
G16 FixedQueue Watermark (%)	1	83.6	39.49	86.82	77.22	82.51	83.6
G16 SharedQueue Ema (%)	1	84.18	30.52	68.39	74.84	81.31	84.18
G16 SharedQueue Watermark (%)	1	86.96	33.47	68.17	76.08	85.07	86.96
Work Stealing Pool (%)	1	80.46	30.02	71.82	80.85	81.31	80.46

[2] Values are normalized to the relative proportion of the service's average value
[3] See 4.1 for description of metric
[4] Only computer 1 was able to return CPU values using the sun.management package