Profile-guided heterogeneous-aware scheduling for cloud workloads

Daniel Araújo de Medeiros

[1] Department of Computer Science
Federal University of Bahia
Salvador, Brazil

Status on
September 2018

Table of Contents

Premises

The proposed scheme, Hurry-Up, is based on the following premises:

- **Heavy vs Light requests**: There is significant difference in computing demands in the user requests (heavy and light ones).
- **Hot functions**: The application has functions showing high computing demands (hot functions), accounting for the increase of the tail latency.

Empirical Observations

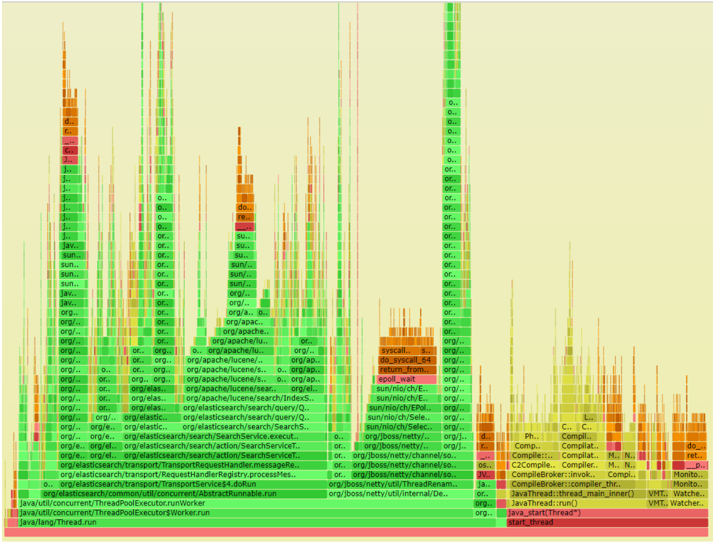
Empirical observations noticed in the algorithm design:

- **Threshold**: To minimize unnecessary up and down migrations, a threshold should be defined when monitoring hot function execution. This is used to better distinguish between heavy and light loads.
- **Priority**: When multiple requests compete for a limited number of cores, there's a need to manage in the task mapping decisions.

Profiling phase - How to identify a hot function?

- Hot functions are determined empirically via standard software profiling.
- The chosen function should not have a very large number of calls because it may lead to many unnecessary migrations.
- The chosen function should be related to the tail latency experienced by the application because the algorithm is designed to improve the service time and efficiency.

Elasticsearch's Flame Graph (perf tool)



Elasticsearch’s call tree (YourKit profiler)

Call Tree	Time (ms)		Count
<All threads>	10,780,512	100 %	
<...> java.lang.**Thread.run**()	9,575,828	89 %	45
<2 calls> ThreadPoolExecutor.java:1149 org.elasticsearch.common.util.concurrent.**AbstractRunnable.run**()	8,362,682	78 %	1,410
AbstractRunnable.java:37 org.elasticsearch.transport.**TransportService$4.doRun**()	8,360,366	78 %	1,138
TransportService.java:378 org.elasticsearch.transport.**RequestHandlerRegistry.processMessageReceived**(TransportRequest, TransportChannel)	8,360,366	78 %	1,138
RequestHandlerRegistry.java:77 org.elasticsearch.transport.**TransportRequestHandler.messageReceived**(TransportRequest, TransportChannel, Ta	8,360,273	78 %	1,098
TransportRequestHandler.java:33 org.elasticsearch.search.action.**SearchServiceTransportAction$SearchQueryTransportHandler.messageR**	8,296,286	77 %	544
SearchServiceTransportAction.java:365 org.elasticsearch.search.action.**SearchServiceTransportAction$SearchQueryTransportHandler.m**	8,296,286	77 %	544
SearchServiceTransportAction.java:368 org.elasticsearch.search.**SearchService.executeQueryPhase**(ShardSearchRequest)	8,295,466	77 %	544
SearchService.java:385 org.elasticsearch.search.**SearchService.loadOrExecuteQueryPhase**(ShardSearchRequest, SearchContext, Qu	8,292,433	77 %	544
SearchService.java:372 org.elasticsearch.search.query.**QueryPhase.execute**(SearchContext)	8,292,421	77 %	544
QueryPhase.java:113 org.elasticsearch.search.query.**QueryPhase.execute**(SearchContext, IndexSearcher)	8,292,415	77 %	544
QueryPhase.java:384 org.apache.lucene.search.**IndexSearcher.search**(Query, Collector)	8,292,168	77 %	544
IndexSearcher.java:535 org.apache.lucene.search.**IndexSearcher.search**(List, Weight, Collector)	8,228,665	76 %	544
IndexSearcher.java:821 org.apache.lucene.search.**BulkScorer.score**(LeafCollector, Bits)	8,209,549	76 %	16,613
BulkScorer.java:39 org.apache.lucene.search.**BooleanScorer.score**(LeafCollector, Bits, int, int)	8,106,592	75 %	13,310
BooleanScorer.java:335 org.apache.lucene.search.**BooleanScorer.scoreWindow**(BooleanScorer$BulkScorerAndI	8,095,698	75 %	500,945
BooleanScorer.java:311 org.apache.lucene.search.**BooleanScorer.scoreWindowMultipleScorers**(LeafCollec	8,057,975	75 %	495,631
BooleanScorer.java:266 org.apache.lucene.search.**BooleanScorer.scoreWindowIntoBitSetAndReplay**(I	8,043,423	75 %	495,631
BooleanScorer.java:219 org.apache.lucene.search.**BooleanScorer$BulkScorerAndDoc.score**(LeafColl	7,012,186	65 %	1,957,032
BooleanScorer.java:61 org.apache.lucene.search.**BooleanScorer.score**(LeafCollector, Bits, int, int)	4,980,101	46 %	241,847
BooleanScorer.java:335 org.apache.lucene.search.**BooleanScorer.scoreWindow**(BooleanScorer	4,978,243	46 %	241,847
BooleanScorer.java:311 org.apache.lucene.search.**BooleanScorer.scoreWindowMultipleSc**	4,974,738	46 %	241,654
BooleanScorer.java:266 org.apache.lucene.search.**BooleanScorer.scoreWindowIntoBit!**	4,970,490	46 %	241,654
BooleanScorer.java:219 org.apache.lucene.search.**BooleanScorer$BulkScorerAndDc**	4,065,679	38 %	483,305
BooleanScorer.java:61 org.apache.lucene.search.**Weight$DefaultBulkScorer.scoi**	4,064,482	38 %	483,305
Weight.java:183 org.apache.lucene.search.**Weight$DefaultBulkScorer.scorel**	4,061,427	38 %	483,305
Weight.java:196 org.apache.lucene.search.**BooleanScorer$OrCollector.col**	3,125,437	29 %	266,489,083
BooleanScorer.java:143 org.apache.lucene.search.**TermScorer.score**()	2,561,114	24 %	266,489,083
TermScorer.java:66 org.apache.lucene.search.similarities.**TFIDFSimila**	2,010,255	19 %	266,489,083
TFIDFSimilarity.java:715 org.apache.lucene.codecs.lucene53.**Lucei**	879,723	8 %	266,489,082
Lucene53NormsProducer.java:139 org.apache.lucene.store.**Buffer**	159,392	1 %	120,489,916
Lucene53NormsProducer.java:139 org.apache.lucene.store.**ByteB**	150,249	1 %	145,999,166
TFIDFSimilarity.java:713 org.apache.lucene.search.similarities.**Classic**	277,372	3 %	266,489,083
Weight.java:198 <...> org.apache.lucene.codecs.lucene50.**Lucene50Postin**	360,752	3 %	266,489,081
Weight.java:171 org.apache.lucene.search.**TermScorer.docID**()	567	0 %	483,305
Weight.java:175 org.apache.lucene.search.**TermScorer.docID**()	463	0 %	483,305
BooleanScorer.java:222 <...> org.apache.lucene.search.**BooleanScorer.scoreMatche**	903,610	8 %	241,651
BooleanScorer.java:223 java.util.**Arrays.fill**(long[], long)	≥ 2	0 %	≥ 3,849
BooleanScorer.java:271 <...> org.apache.lucene.util.**PriorityQueue.insertWithOverflow**(Ol	2,712	0 %	483,300
BooleanScorer.java:264 org.apache.lucene.util.**PriorityQueue.clear**()	333	0 %	241,654

Overview of the Scheduler Design

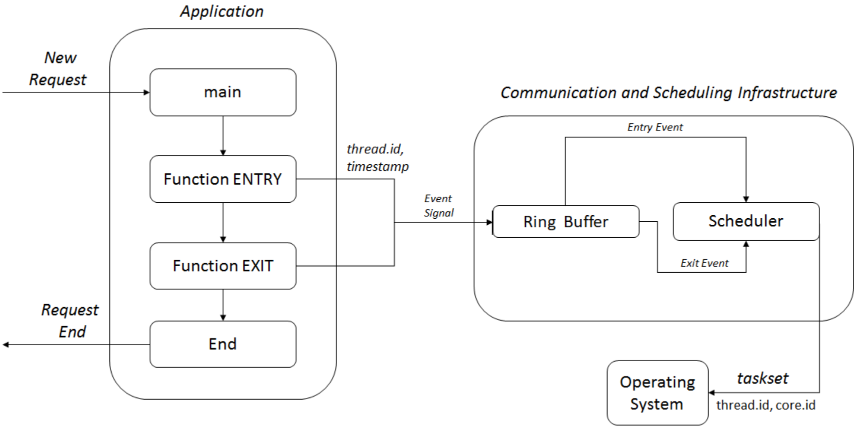


Illustration of the algorithm design

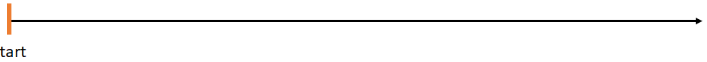
Number of physical cores;
In case of Juno, 6 cores.

Long integer type

0: Not in Function
1: Executing hot function
2: 1 + Over threshold

	core.id	thread.id	timestamp	thread priority
Big cores	0			
	1			
Little cores	2			
	3			
	4			
	5			

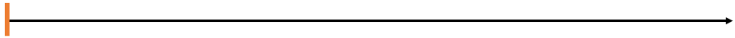
EVENT TIMELINE



Case 1: No Competition and Below Threshold

core.id	*thread.id*	*timestamp*	*thread priority*
0			
1			
2			
3			
4			
5			

EVENT TIMELINE

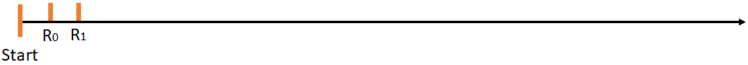


Start

Case 1: No Competition and Below Threshold

core.id	thread.id	timestamp	thread priority
0	900	2	1
1	901	3	1
2			
3			
4			
5			

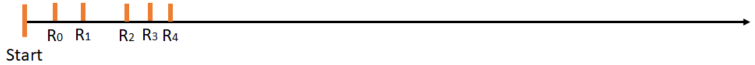
EVENT TIMELINE



Case 1: No Competition and Below Threshold

core.id	*thread.id*	*timestamp*	*thread priority*
0	900	2	1
1	901	3	1
2	902	5	1
3	903	6	1
4	904	7	1
5			

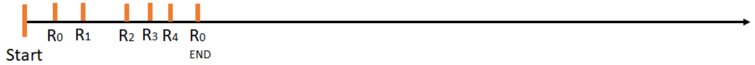
EVENT TIMELINE



Case 1: No Competition and Below Threshold

core.id	thread.id	timestamp	thread priority
0	900	2	0
1	901	3	1
2	902	5	1
3	903	6	1
4	904	7	1
5			

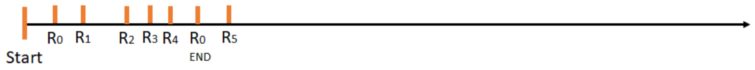
EVENT TIMELINE



Case 1: No Competition and Below Threshold

core.id	*thread.id*	*timestamp*	*thread priority*
0	900	2	0
1	901	3	1
2	902	5	1
3	903	6	1
4	904	7	1
5	905	9	1

EVENT TIMELINE



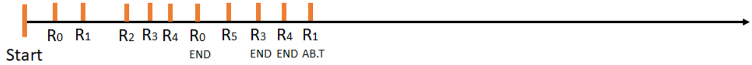
Notes

- In the initialization, the threads are allocated in a round-robin fashion.
- Up migration to a big core in the initialization is not performed, even when the big core is available, so as to reduce the number of unnecessary migrations.
- If the thread is not above the threshold in a little core (priority = 2 and core id between 2 and 5), a migration is not performed (the request was light and little core was already able to handle it).

Case 2: No Competition and Above Threshold

core.id	*thread.id*	*timestamp*	*thread priority*
0	900	2	0
1	901	3	2
2	902	5	1
3	903	6	0
4	904	7	0
5	905	9	1

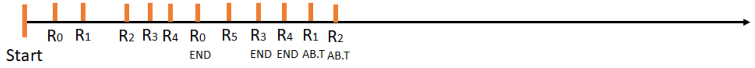
EVENT TIMELINE



Case 2: No Competition and Above Threshold

core.id	thread.id	timestamp	thread priority
0	902	2	2
1	901	3	2
2	900	5	0
3	903	6	0
4	904	7	0
5	905	9	1

EVENT TIMELINE



Notes

- If the request is already on big core and goes to priority 2, we can't do anything about it - already on fastest core.
- In case a heavy request (above threshold) is on a little core and there's a slot available on a big core (priority = 0 or 1), swap it.

Case 3: Competition and Above Threshold

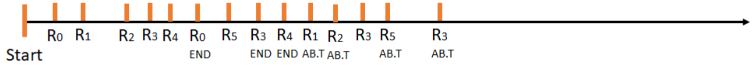
core.id	*thread.id*	*timestamp*	*thread priority*
0	902	2	2
1	901	3	2
2	900	5	0
3	903	20	1
4	904	7	0
5	905	9	1

EVENT TIMELINE

Case 3: Competition and Above Threshold

core.id	*thread.id*	*timestamp*	*thread priority*
0	902	2	2
1	901	3	2
2	900	5	0
3	903	20	2
4	904	7	0
5	905	9	2

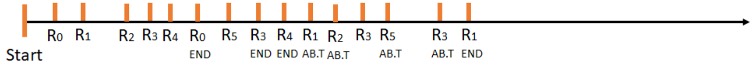
EVENT TIMELINE



Case 3: Competition and Above Threshold

core.id	*thread.id*	*timestamp*	*thread priority*
0	902	2	2
1	905	9	2
2	900	5	0
3	903	20	2
4	904	7	0
5	901	3	0

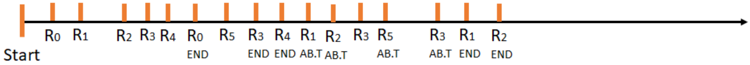
EVENT TIMELINE



Case 3: Competition and Above Threshold

core.id	*thread.id*	*timestamp*	*thread priority*
0	903	20	2
1	905	9	2
2	900	5	0
3	902	2	0
4	904	7	0
5	901	3	0

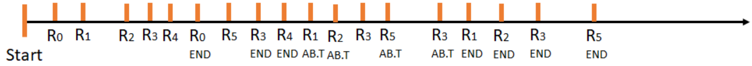
EVENT TIMELINE



Case 3: Competition and Above Threshold

core.id	*thread.id*	*timestamp*	*thread priority*
0	903	20	0
1	905	9	0
2	900	5	0
3	902	2	0
4	904	7	0
5	901	3	0

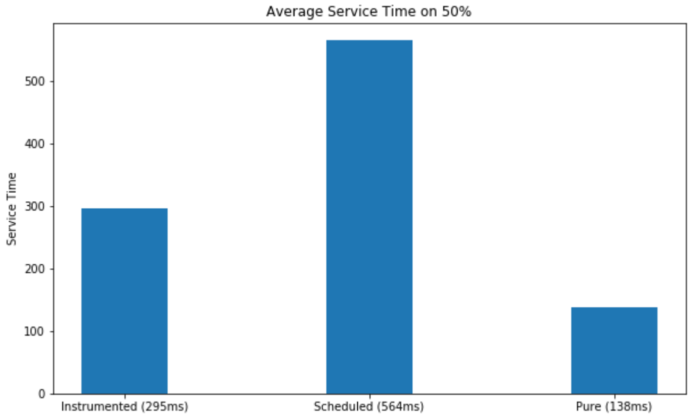
EVENT TIMELINE



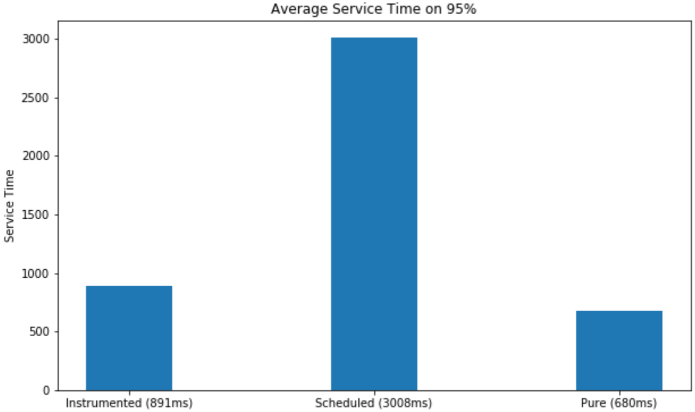
Notes

- Scheduler runs every *t* milliseconds in order to check for new events and perform thread management.
- Threshold should be determined empirically
- Every time a new request arrives, the timestamp on matrix is updated.
- In case there are more requests with priority = 2 than big cores, the oldest ones has priority of execution (based on the timestamp information)

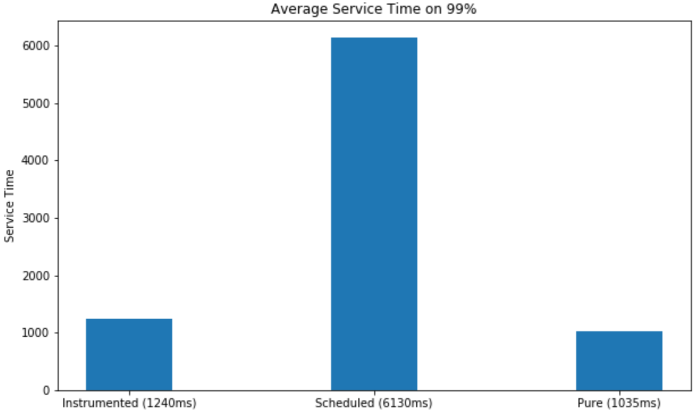
Results


Average Service Time on 50%
Service Time
500
400
300
200
100
0
Instrumented (295ms)
Scheduled (564ms)
Pure (138ms)

Results


Average Service Time on 95%
Service Time
3000
2500
2000
1500
1000
500
0
Instrumented (891ms)
Scheduled (3008ms)
Pure (680ms)

Results


Average Service Time on 99%
Service Time
6000
5000
4000
3000
2000
1000
0
Instrumented (1240ms)
Scheduled (6130ms)
Pure (1035ms)

Future steps

Challenges ahead:

- Algorithm is designed for the situation where the number of cores is equal or less to number of threads.
- Code instrumentation can lead to excessive overhead.
- Initial results show that further refining is necessary in the algorithm implementation.
- Apply the algorithm to other cloud workloads, such as Cassandra.

List of Contributors

- Daniel Mossé (Pitt): Advising Committee;
- Denilson Amorim (UFBA): Instrumentation infrastructure;
- George Lima (UFBA): Advising Committee;
- Paul Carpenter (BSC): Constructive feedback and access to the Juno board at BSC;
- Rajiv Nishtala (NTNU): Constructive feedback and help with experiments on Juno;
- Vinicius Petrucci (UFBA): Advising Committee, Chair.

Thank you!