Kindergarten Cop:
Dynamic Nursery Resizing for GHC

Vladimir Janjic, Kevin Hammond (University of St Andrews)
Henrique Ferreiro, Laura Castro (University of A Coruna)

E: kevin@kevinhammond.net
T: @khstandrews, @rephrase_eu

ParaPhrase Project: Parallel Patterns for Heterogeneous Multicore Systems (ICT-288570), 2011-2015, €4.2M budget

13 Partners, 8 European countries

UK, Italy, Germany, Austria, Ireland, Hungary, Poland, Israel

Coordinated by Kevin Hammond St Andrews

RePhrase Project: Refactoring Parallel Heterogeneous Software – a Software Engineering Approach
(ICT-644235), 2015-2018, €3.6M budget

8 Partners, 6 European countries
UK, Spain, Italy, Austria, Hungary, Israel

Coordinated by Kevin Hammond St Andrews

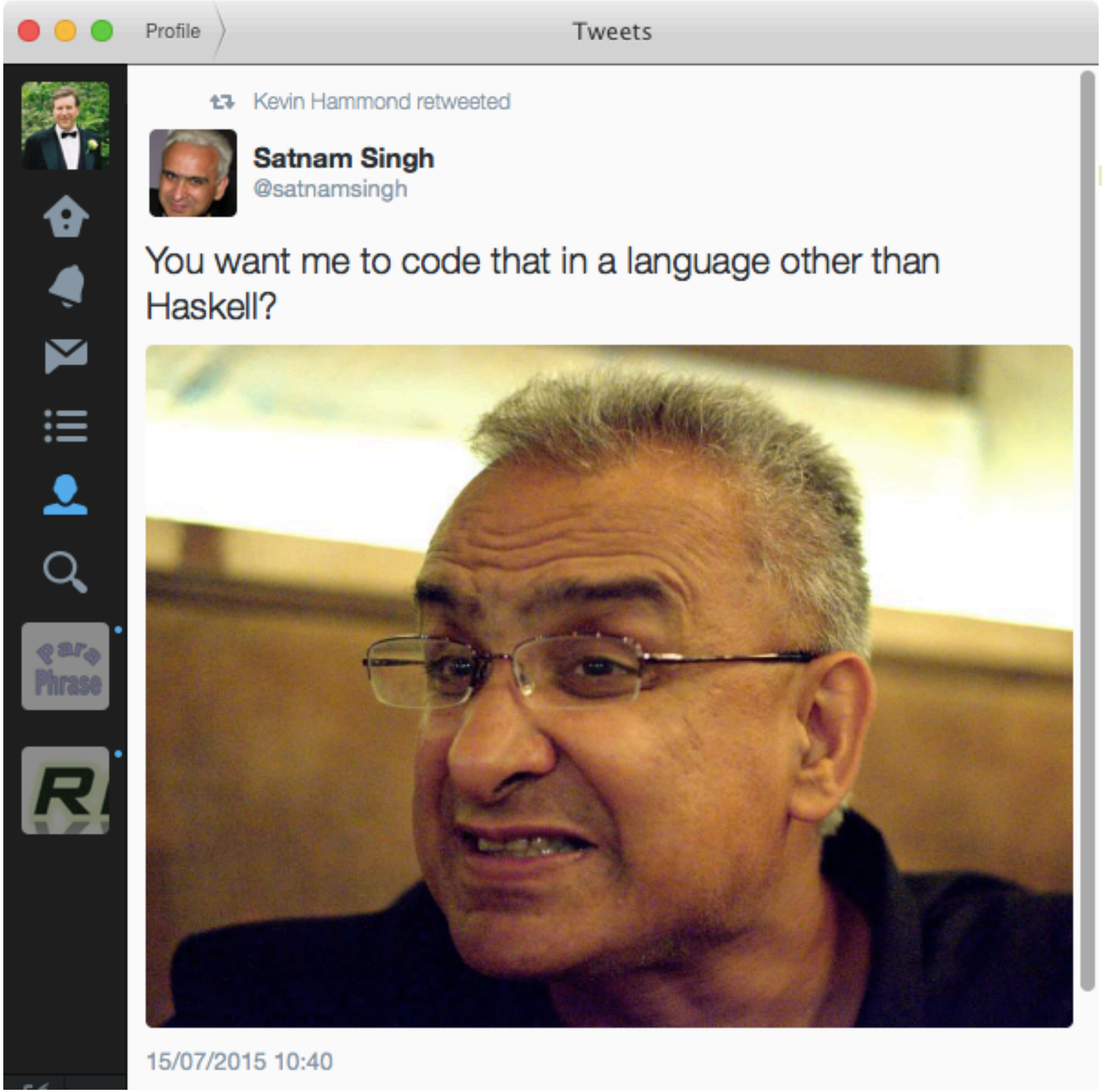
The Glorious Haskell Compiler (GHC)

University of St Andrews

- The de-facto standard for Haskell
  - the non-strict functional language
- Originally, the Glasgow Haskell Compiler
  - now maintained at Microsoft Research

Haskell 98 Language and Libraries, the Revised Report
Simon Peyton Jones (ed.) ... Kevin Hammond ..
Cambridge University Press, 2003

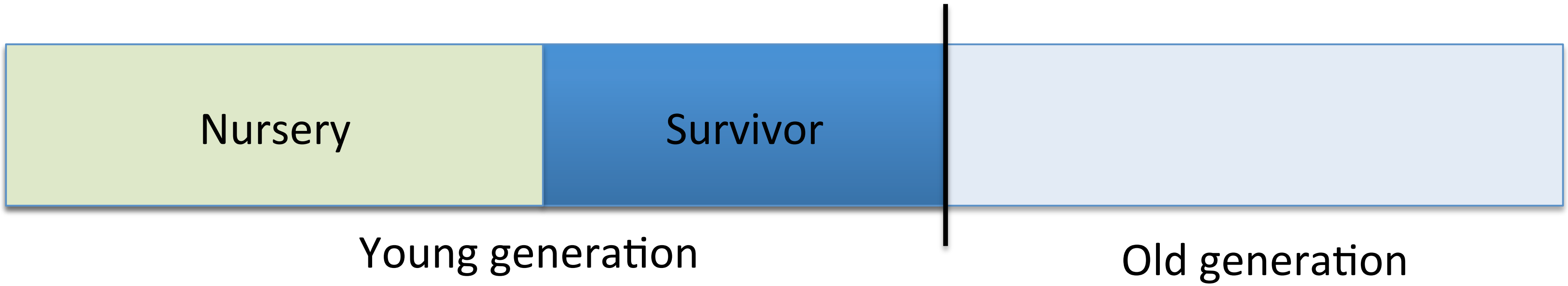
The Glasgow Haskell Compiler
`http://www.haskell.org/ghc`



Satnam Singh
Google

Generational garbage collection

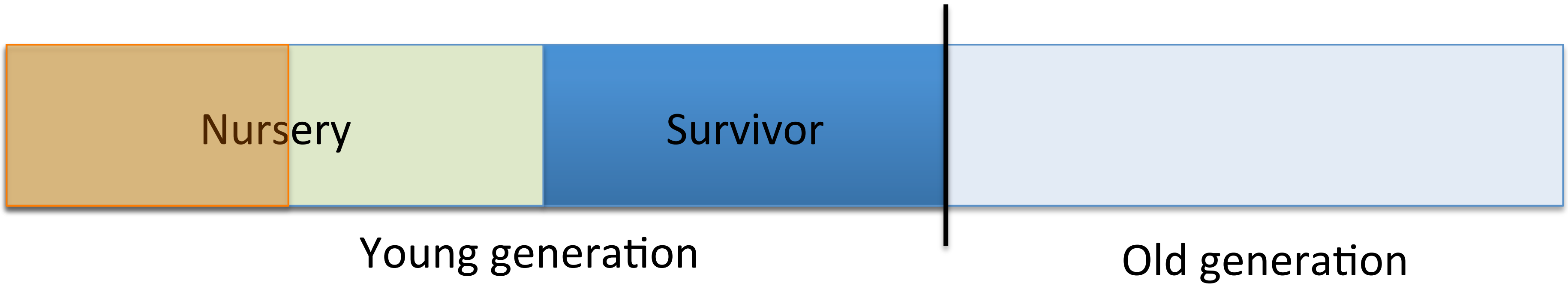
- GHC uses Appel-style generational garbage collection
  - Assumption: most of the allocated objects die young
- Heap divided into a number of generations
  - Usually two generations: young and old
  - Young generation divided into the nursery and the survivor area



A. W. Appel. Simple Generational Garbage Collection and Fast Allocation, Software: Practice and Experience,19:2, p. 171-182, 1989.

Generational garbage collection (2)

- New objects are (almost) always allocated in the nursery



Generational garbage collection (3)

- When the nursery becomes full, *minor collection* is triggered

- Live data is copied into the survivor area

- Data that survives a number of collections is promoted to the old generation
- When the old generation becomes full, *major collection* is triggered (whole heap is collected)

Generational garbage collection (4)

- Generational collectors are designed to do minor collections most of the time

- Performance heavily depends on the size of the nursery!
  - Smaller nursery size => better cache behaviour
  - Larger nursery size => fewer collections, collections less expensive
    - Cost of the garbage collection depends on the amount of live data, not garbage
  - Imperative languages: make nursery as large as possible!
  - Lazy functional languages: small nurseries

GHC Garbage Collection

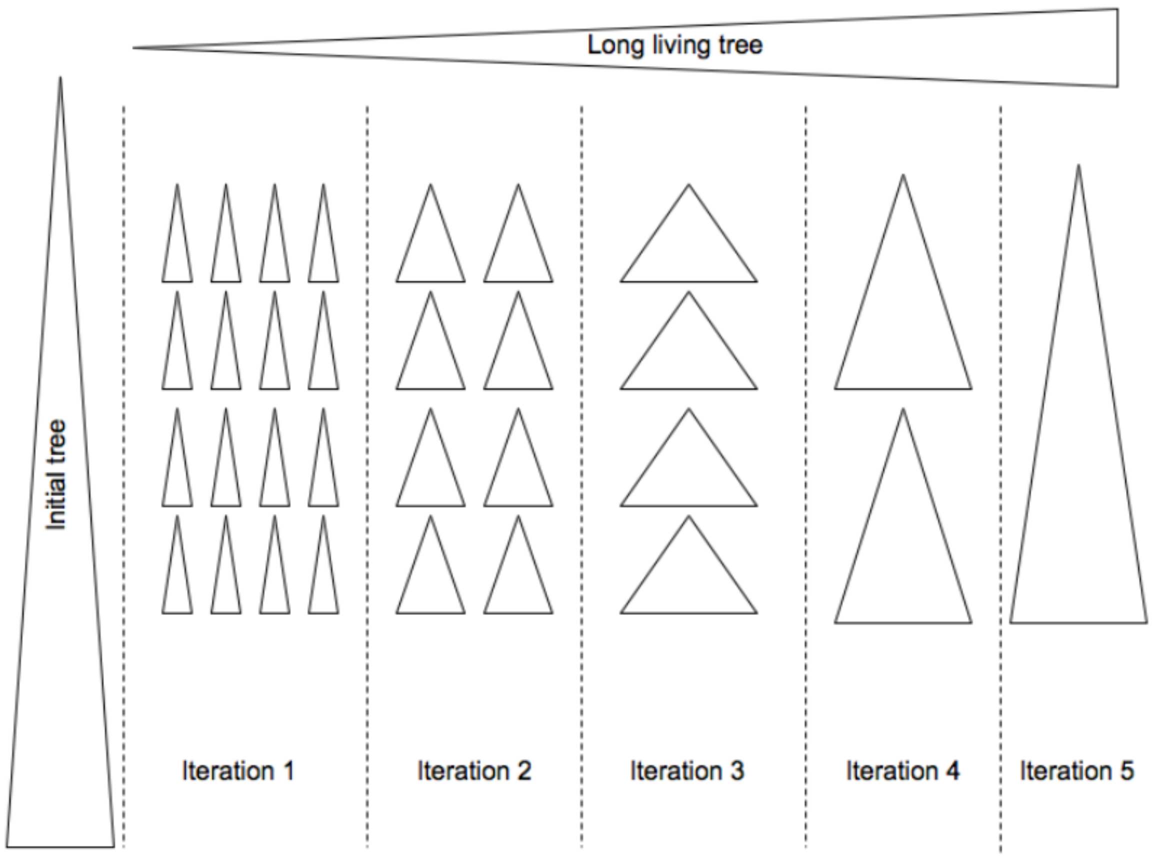
- Generational garbage collection, two generations
- Size of the nursery can be set to a constant (-A <size>)
  - or RTS can *dynamically* change nursery size after each collection (-H)
- Dynamic nursery resizing algorithm sets the nursery to have the largest "reasonable" size that is possible
  - After each garbage collection, the nursery size is set to be

$$\frac{H - N}{1 + p}$$

H – heap size, N – 2x size of the live data,

p – percentage of data copied from the nursery in the last collection

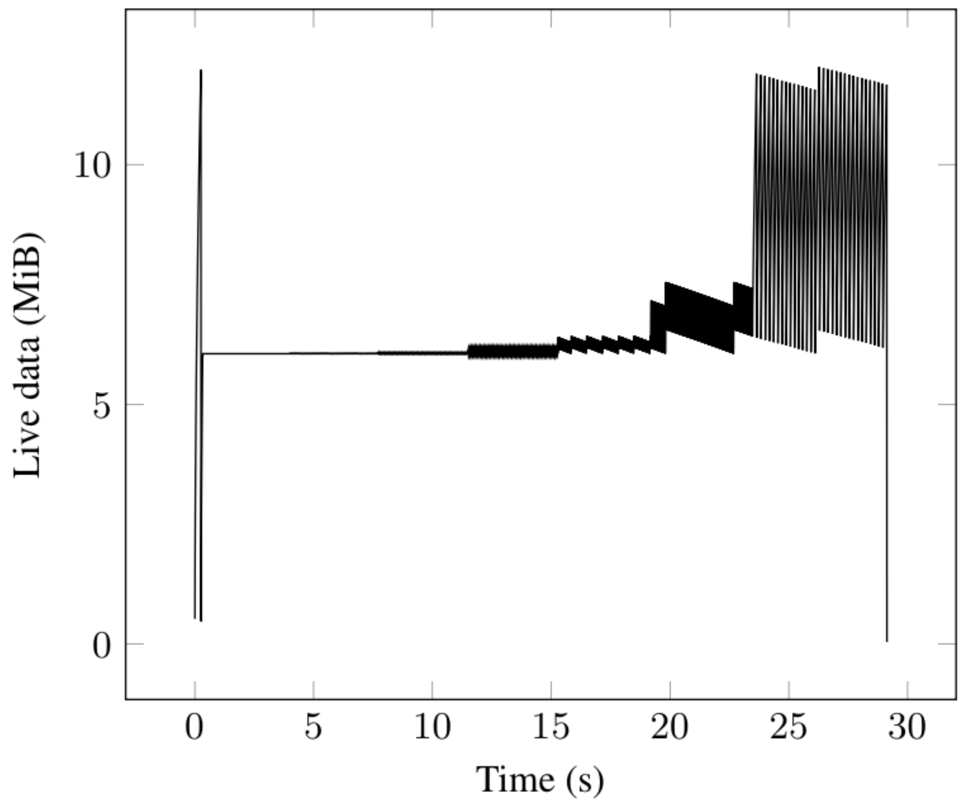
Binary-trees benchmark



Allocation size increases throughout program execution

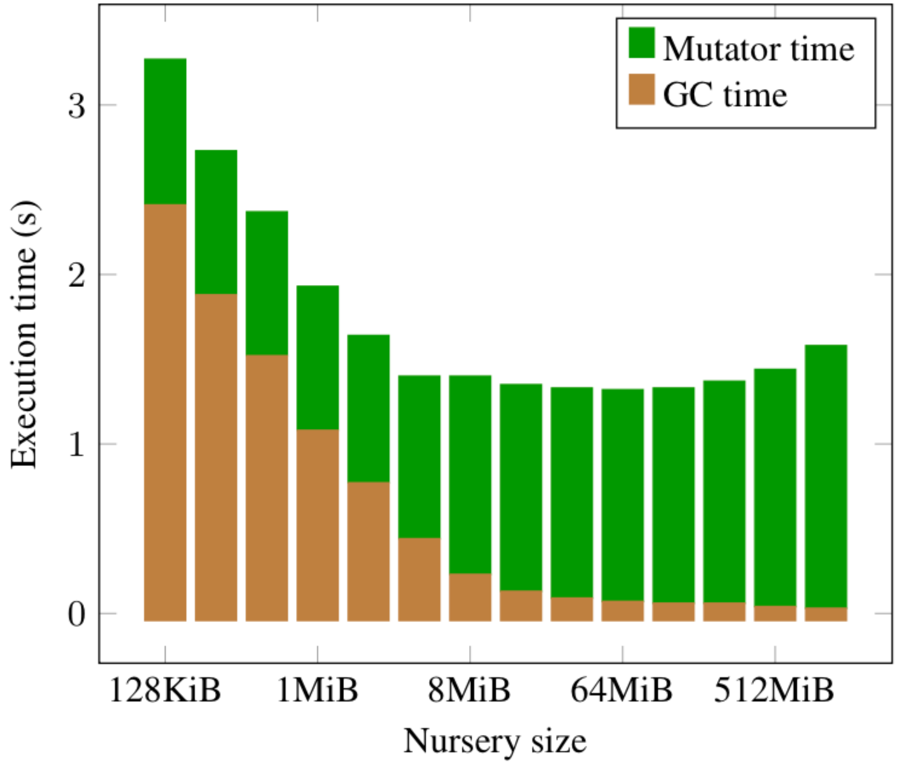
Binary-trees memory behaviour

University of St Andrews



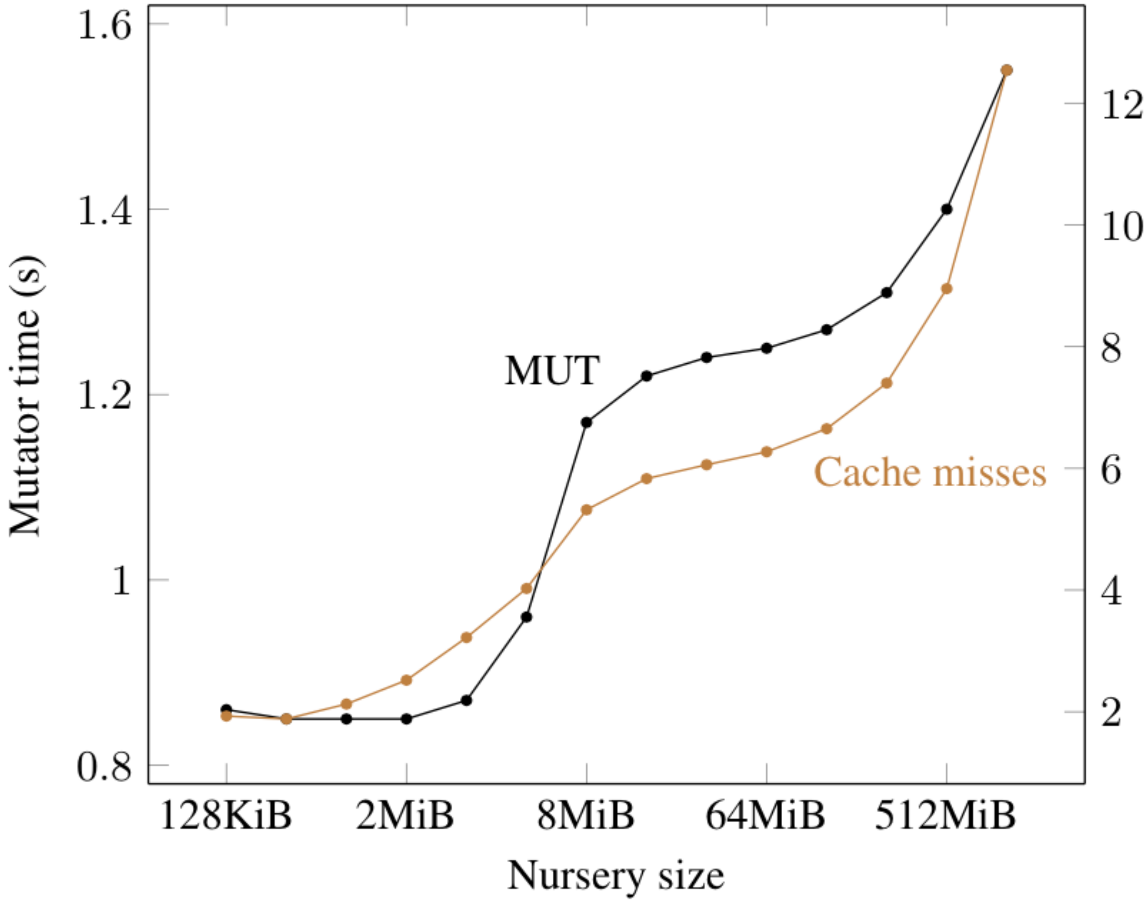
2.4GHz Intel i7 processor, 4Mb L2 Cache, 4GB RAM

Binary-trees with Constant-sized nurseries

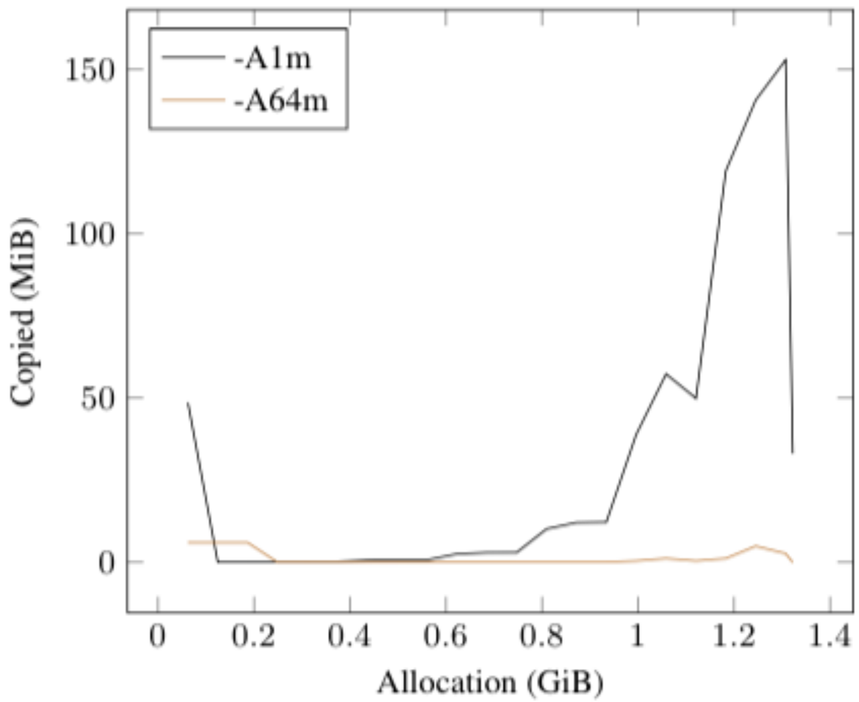


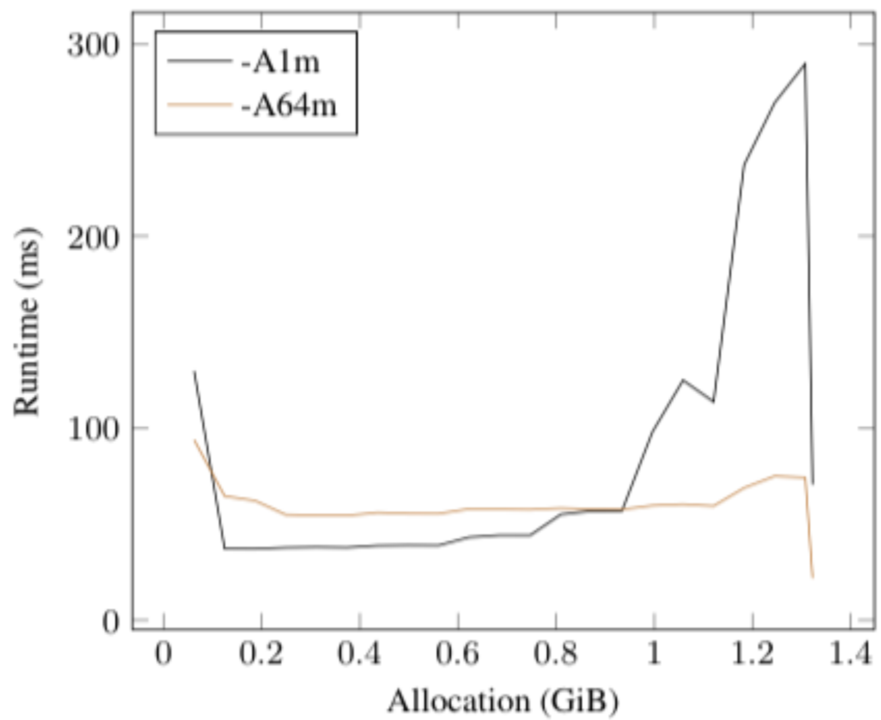
- The bigger the nursery , the less time is spent in garbage collection
  - however, evaluation (mutator) time is also increased

Why does mutator time increase?



Binary-trees phase analysis





(Nursery) Size Matters!

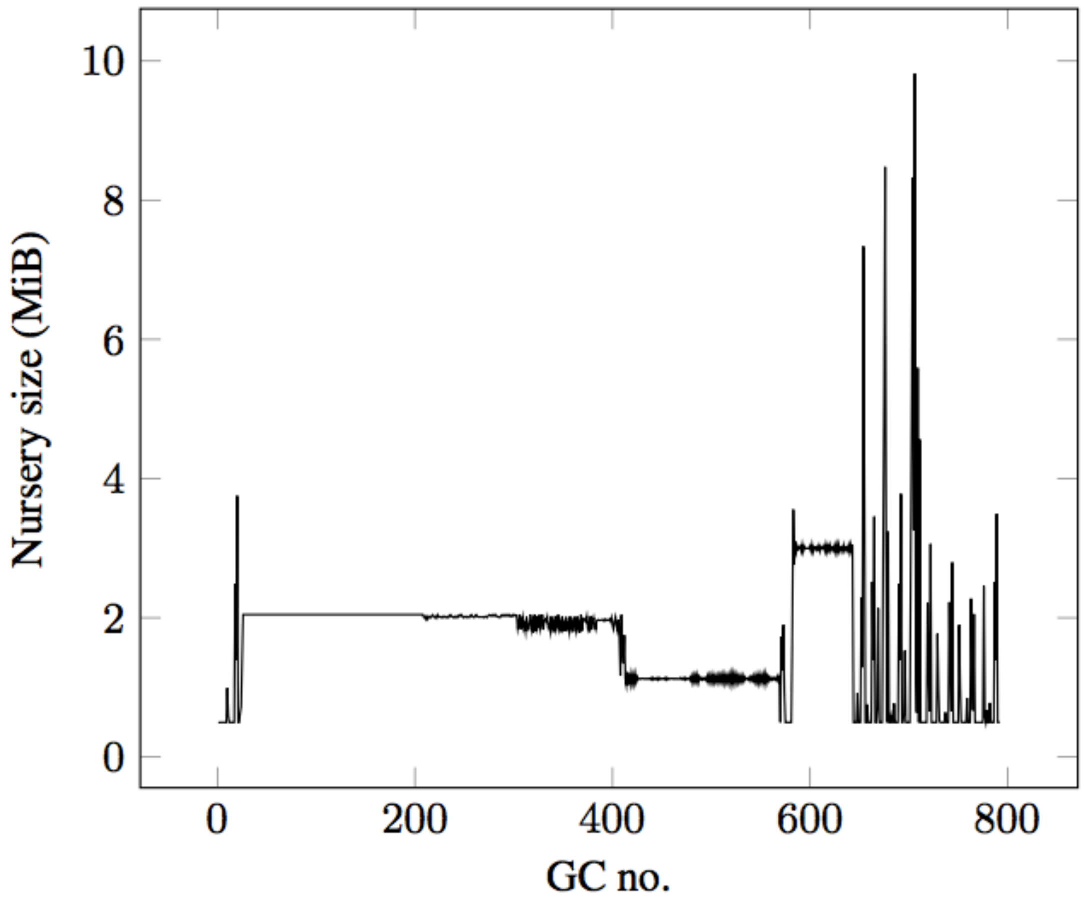
GC configuration	Speedup
-A2m	1.44
-A8m	1.69
-A64m	1.78

Speedup against default of 0.5MB fixed nursery (-A500k)

What can we conclude?

- Unlike imperative programming, a bigger nursery does not necessarily mean better performance
- In programs with irregular memory behaviour, nursery size plays a crucial role in the overall performance
- Having the same nursery size for the whole execution may be suboptimal
- In the phases of the program execution where not much data is copied, a smaller nursery size is better
- In the phases where a large amount of data is copied, go for a bigger nursery
  - **In this case garbage collection, rather than cache behaviour, is the main performance bottleneck**

Varying the Nursery Size (GHC –H)



Effect of Varying the Nursery Size

University of St Andrews

GC configuration	Speedup
-A2m	1.44
-A8m	1.69
-A64m	1.78
-H	1.38

Speedup against default of 0.5MB fixed nursery (-A500k)

TAA Dynamic Resizing Algorithm

University of St Andrews

- $TPBM_n$ – Time per Byte Metric
  - time taken by the n^{th} garbage collection divided by the nursery size S_n for that collection
- Target: reduce TPBM as much as possible
- Set the initial nursery size, S_1, to be the size of L2 cache
- After each garbage collection, calculate a new size S_n
  - Fast method: Half the nursery size, i.e. set $S_n = S_{n-1} / 2$
  - If fast method gives worse TPBM, i.e. if $TPBM_n > TPBM_{n-1}$, use the slow method instead
  - Slow method: Good nursery size is between S_{n-2} and S_{n-1} or between S_{n-1} and S_n – do a binary search to find the optimal value

T.A. Anderson. Optimizations in a Private Nursery-based Garbage Collector. *Proc. 2010 International Symposium on Memory Management, ISMM '10,* pages 21–30.

TAA Dynamic Resizing Algorithm (2)

University of St Andrews

fun `fast_update()`

$S_{n-2} = S_{n-1}$

$S_{n-1} = S_n$

$TPBM_n = GCTime_n / S_n$

if $TPBM_n < TPBM_{n-1}$ **then**

$S_n = S_n / 2$

else

$S_n =$ `slow_update`(S_{n-2}, S_{n-1}, S_n)

end

return S_n

end

HORIZON 2020

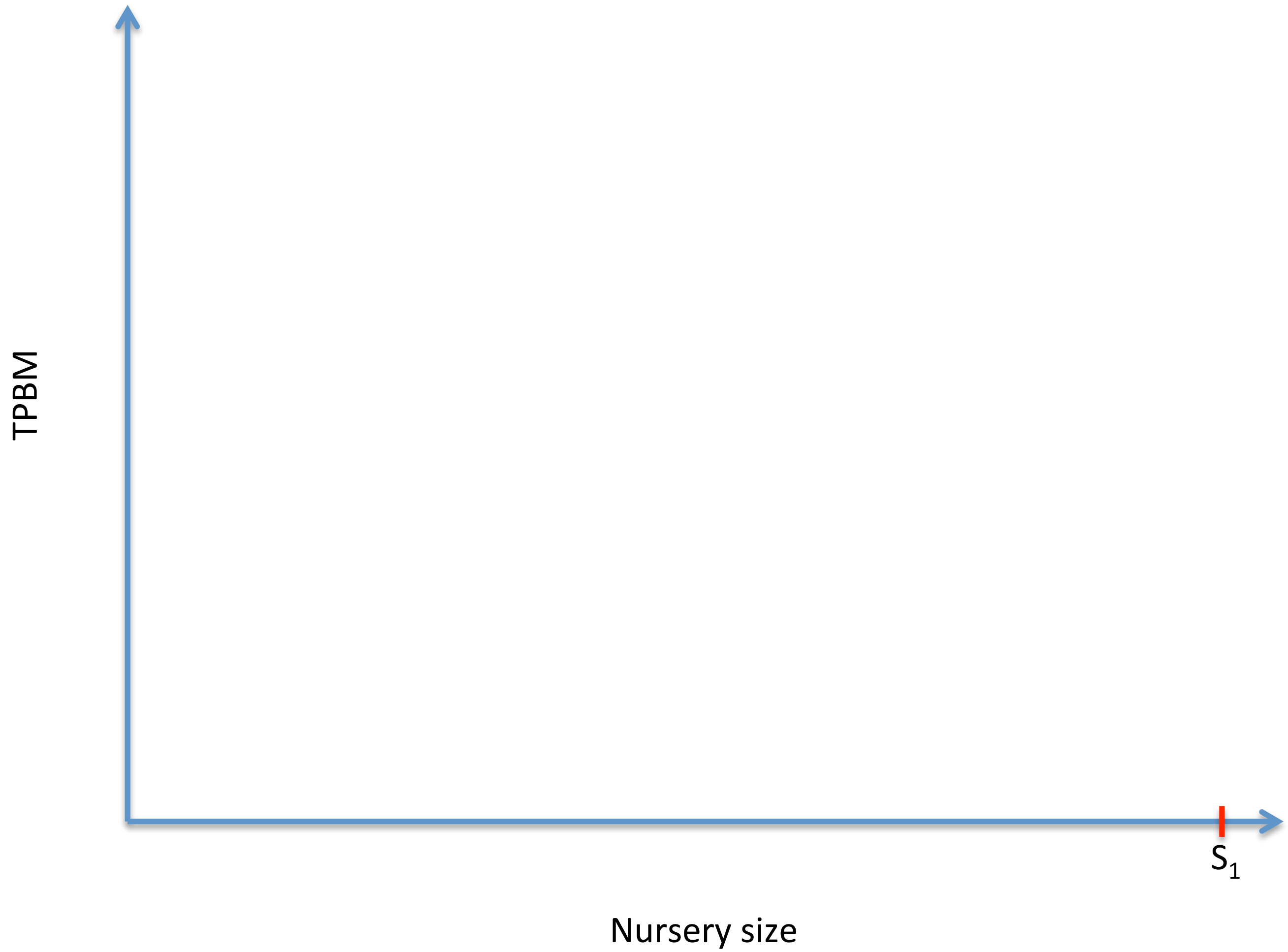
REPHRASE

TAA Dynamic Resizing Algorithm (3)

```
fun slow_update(S_{n-2}, S_{n-1}, S_n)
    if abs(S_n - S_{n-2}) < threshold then
        return S_{n-1}
    end
    S_x = (S_{n-1} + S_{n-2})/2
    [...execution with nursery size S_x ...]
    TPBM_x = GCTime_x/S_x
    if TPBM_x < TPBM_{n-1} then
        return slow_update(S_{n-2}, S_x, S_{n-1})
    else
        S_y = (S_n + S_{n-1})/2
        [...execution with nursery size S_y ...]
        TPBM_y = GCTime_y/S_y
        if TPBM_y < TPBM_{n-1} then
            return slow_update(S_{n-1}, S_y, S_n)
        else
            return slow_update(S_x, S_{n-1}, S_y)
        end
    end
```

Example of TAA

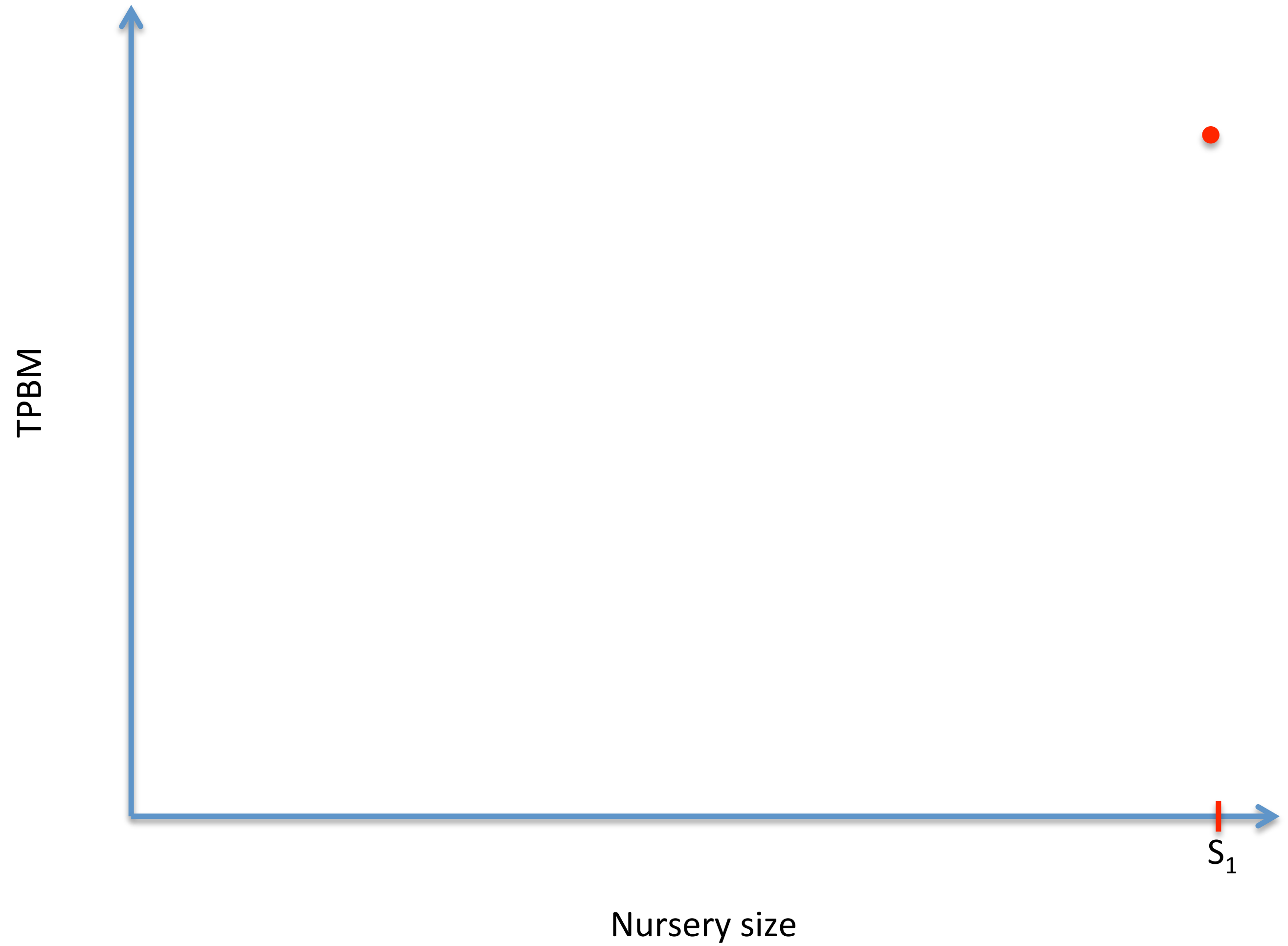
University of St Andrews



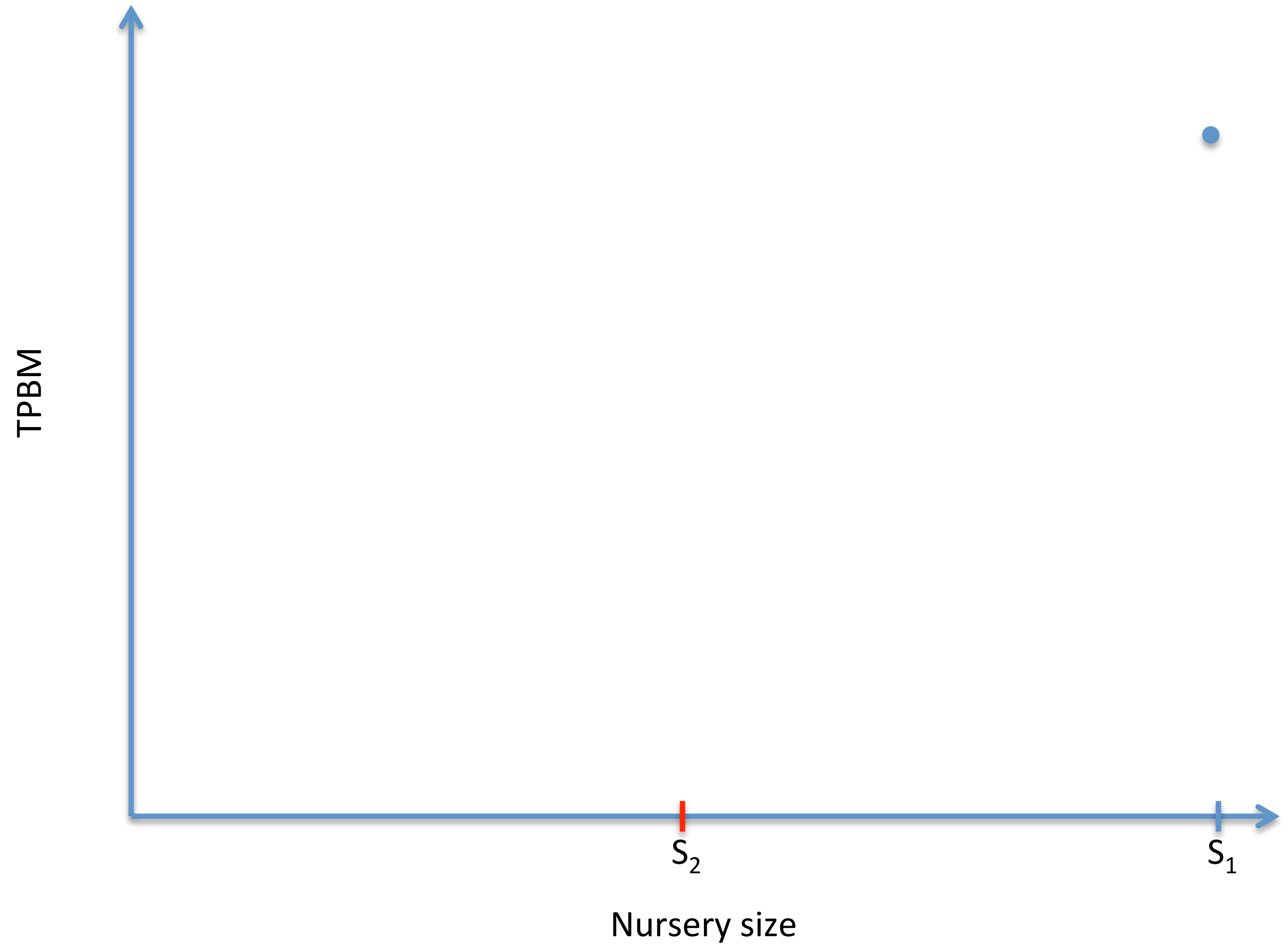
HORIZON 2020

REPHRASE

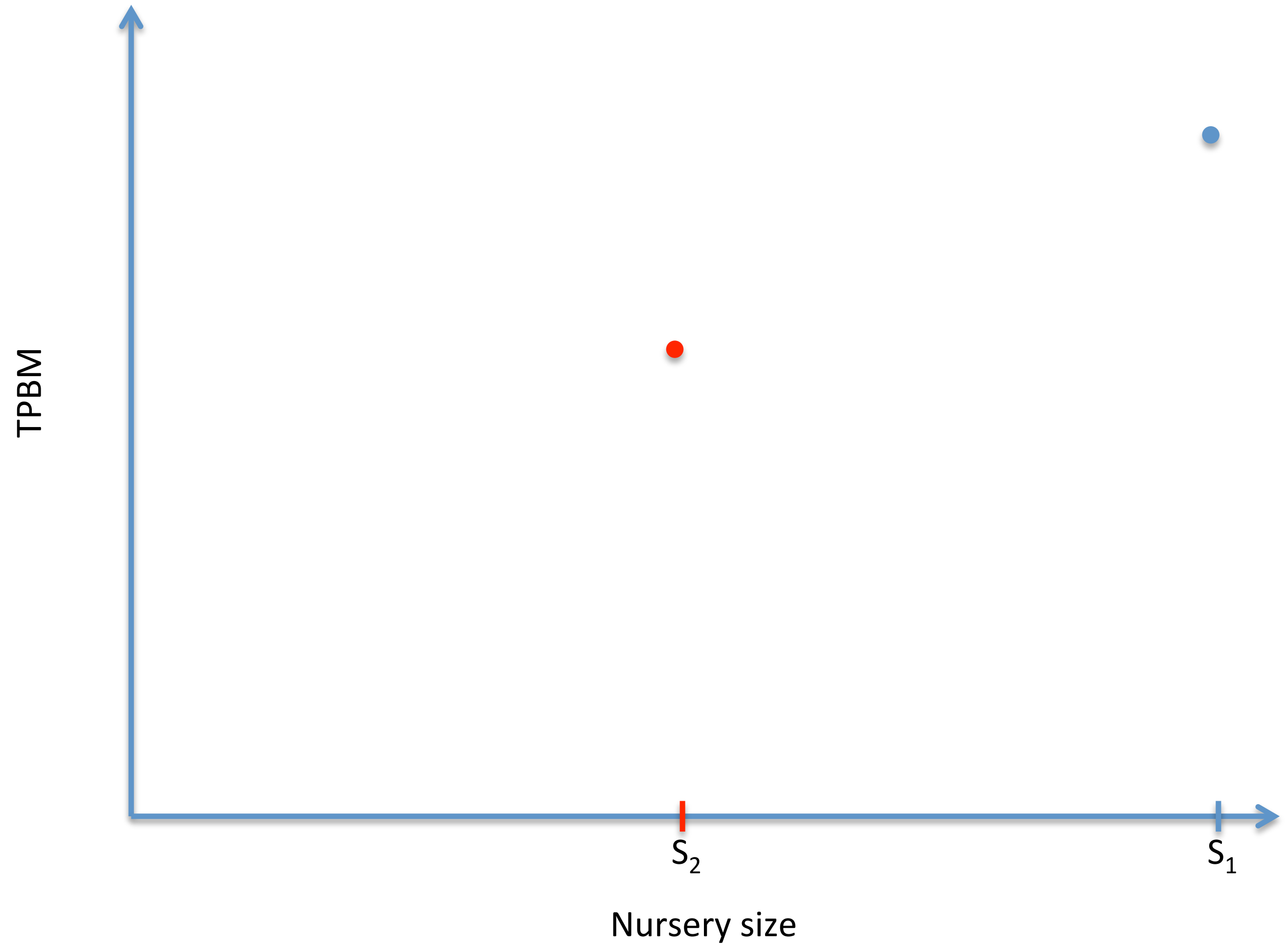
Example of TAA



Example of TAA

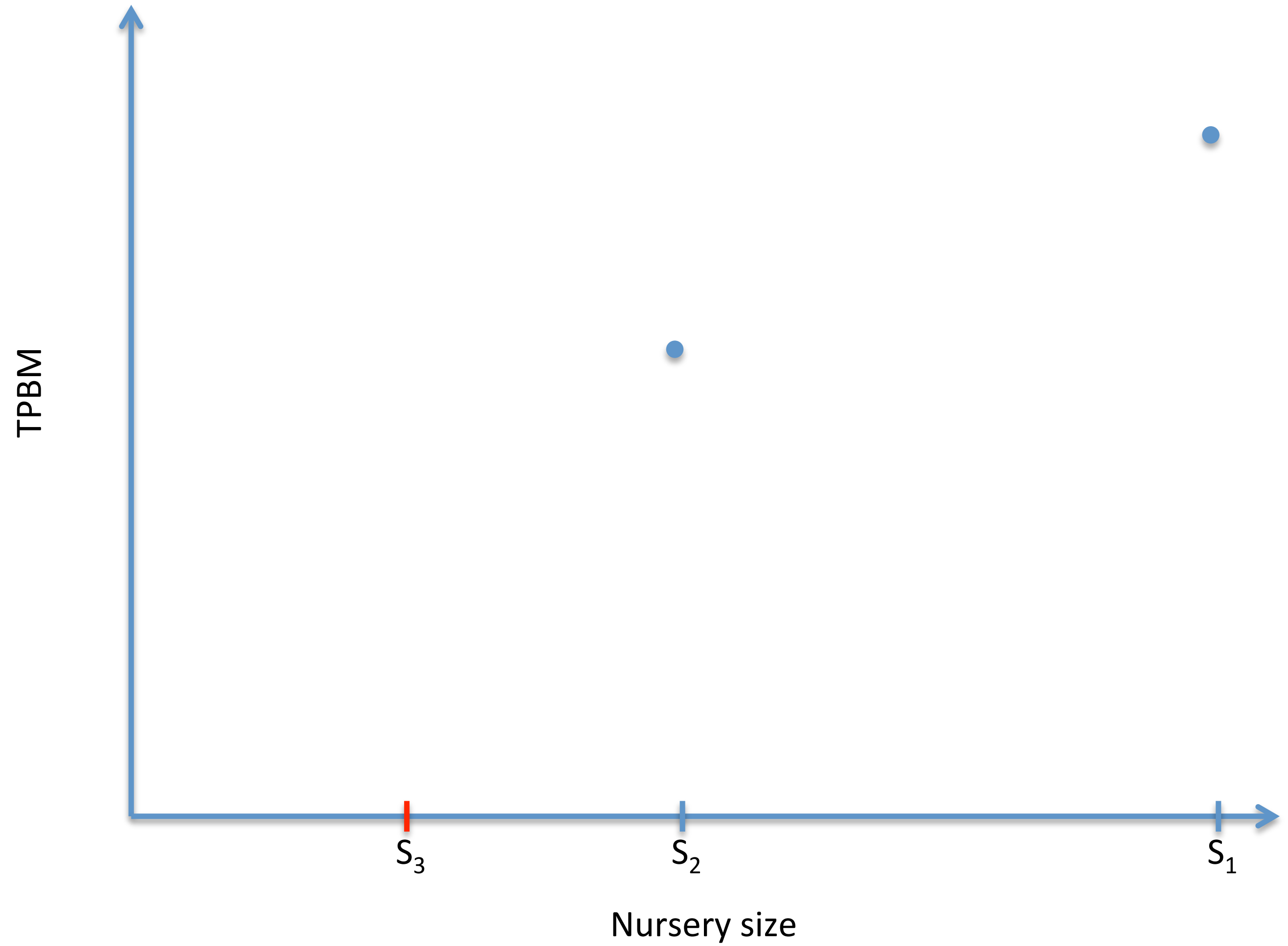


Example of TAA

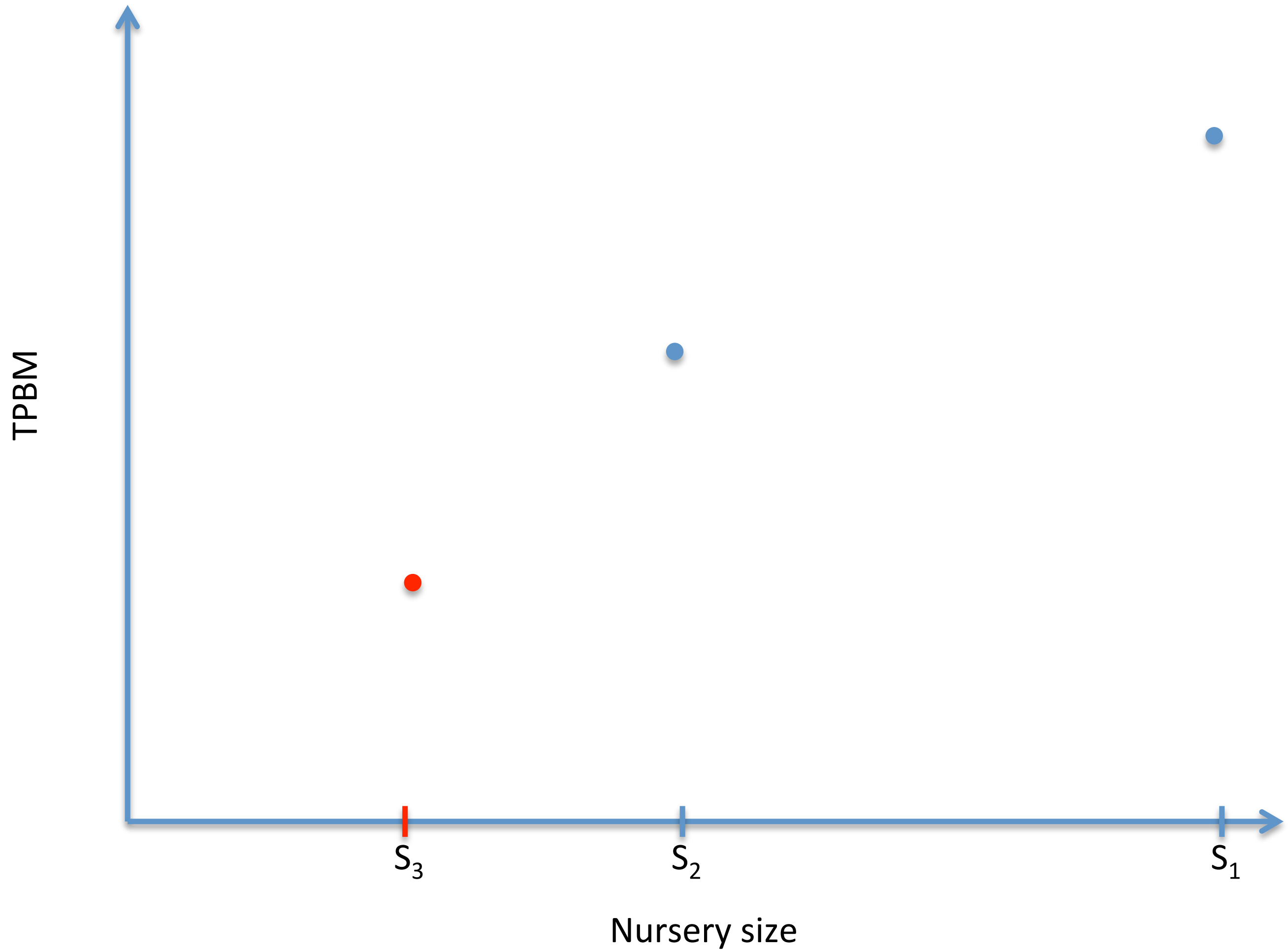


Example of TAA

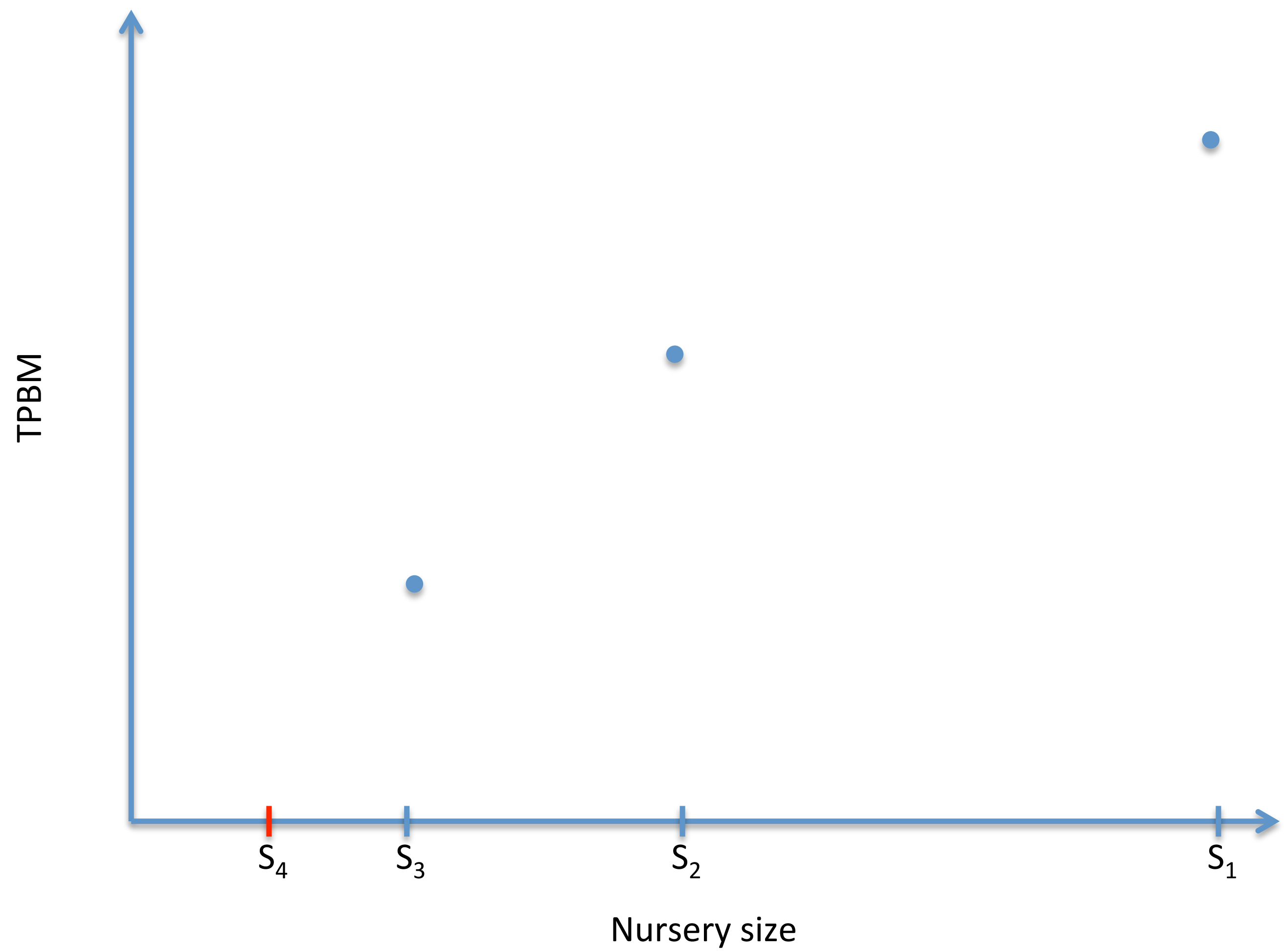
University of St Andrews



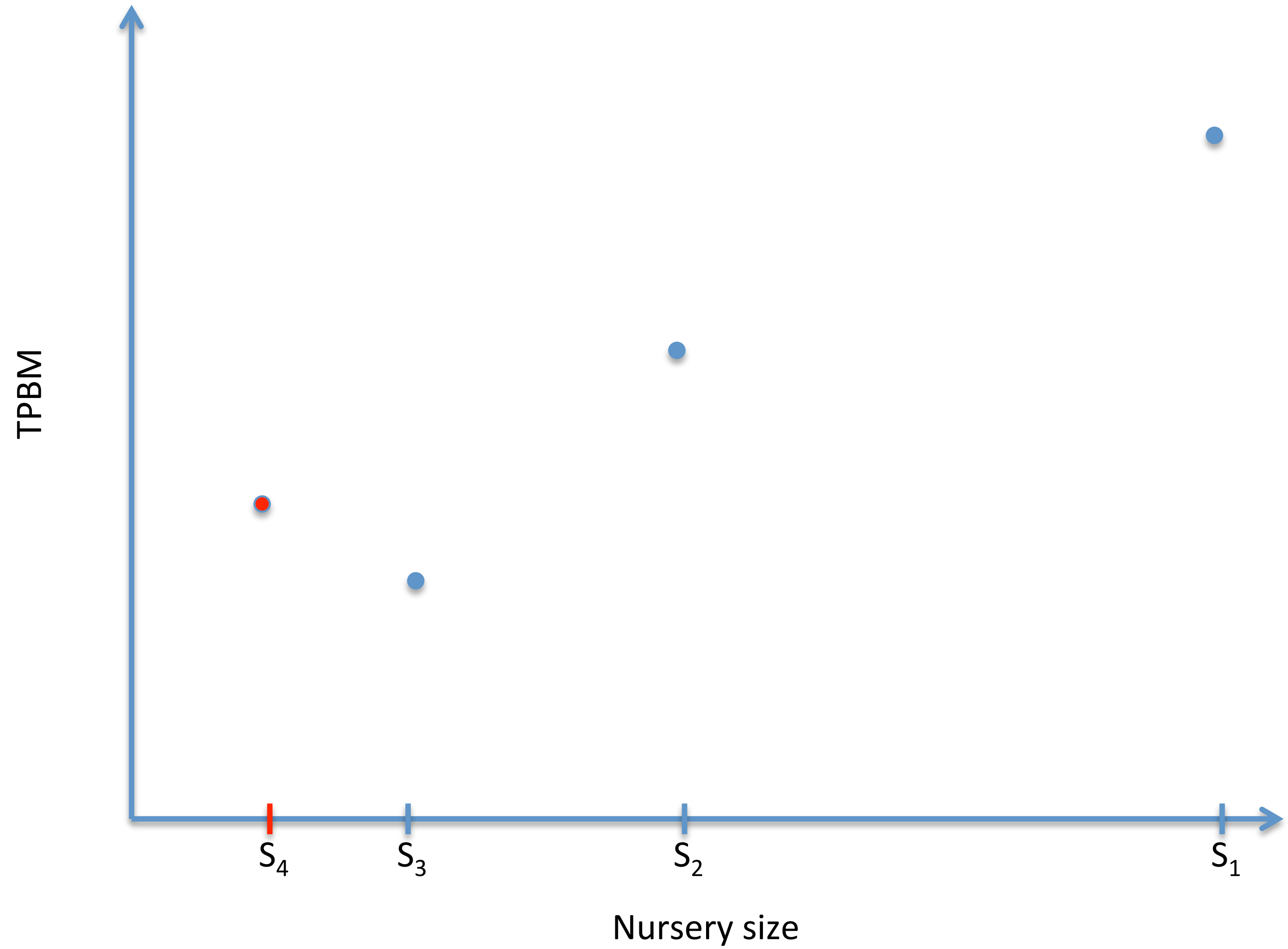
Example of TAA



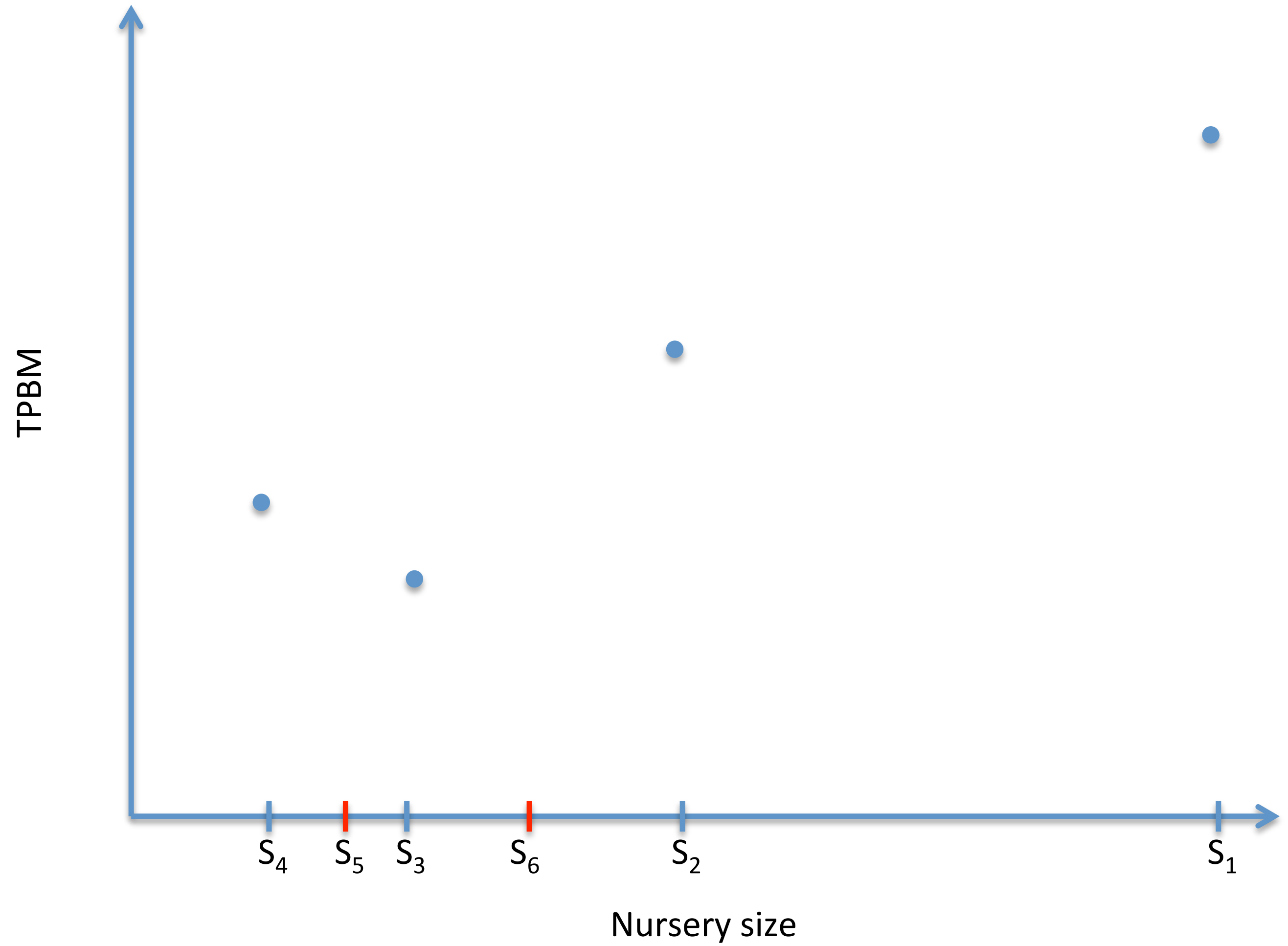
Example of TAA



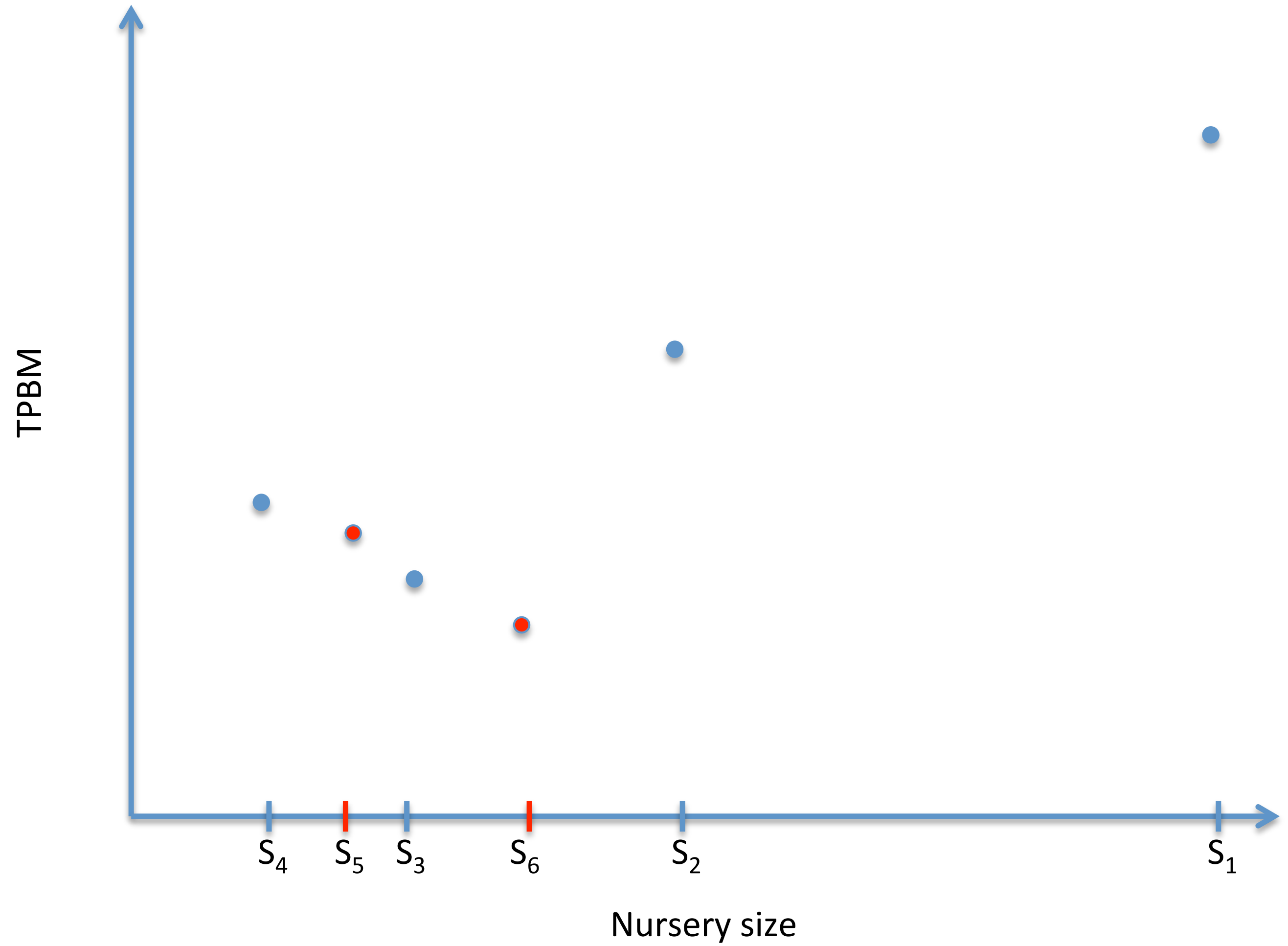
Example of TAA



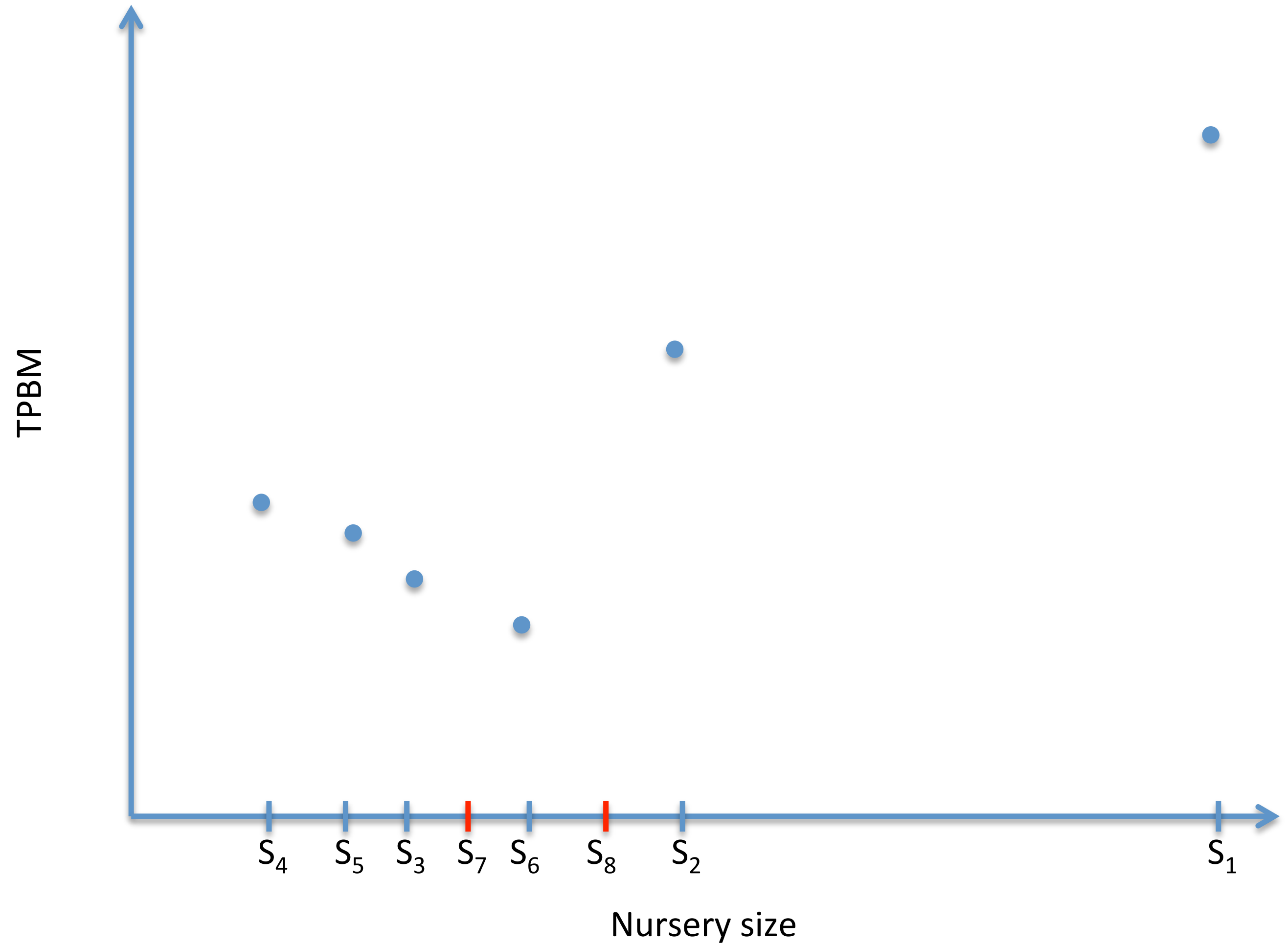
Example of TAA



Example of TAA

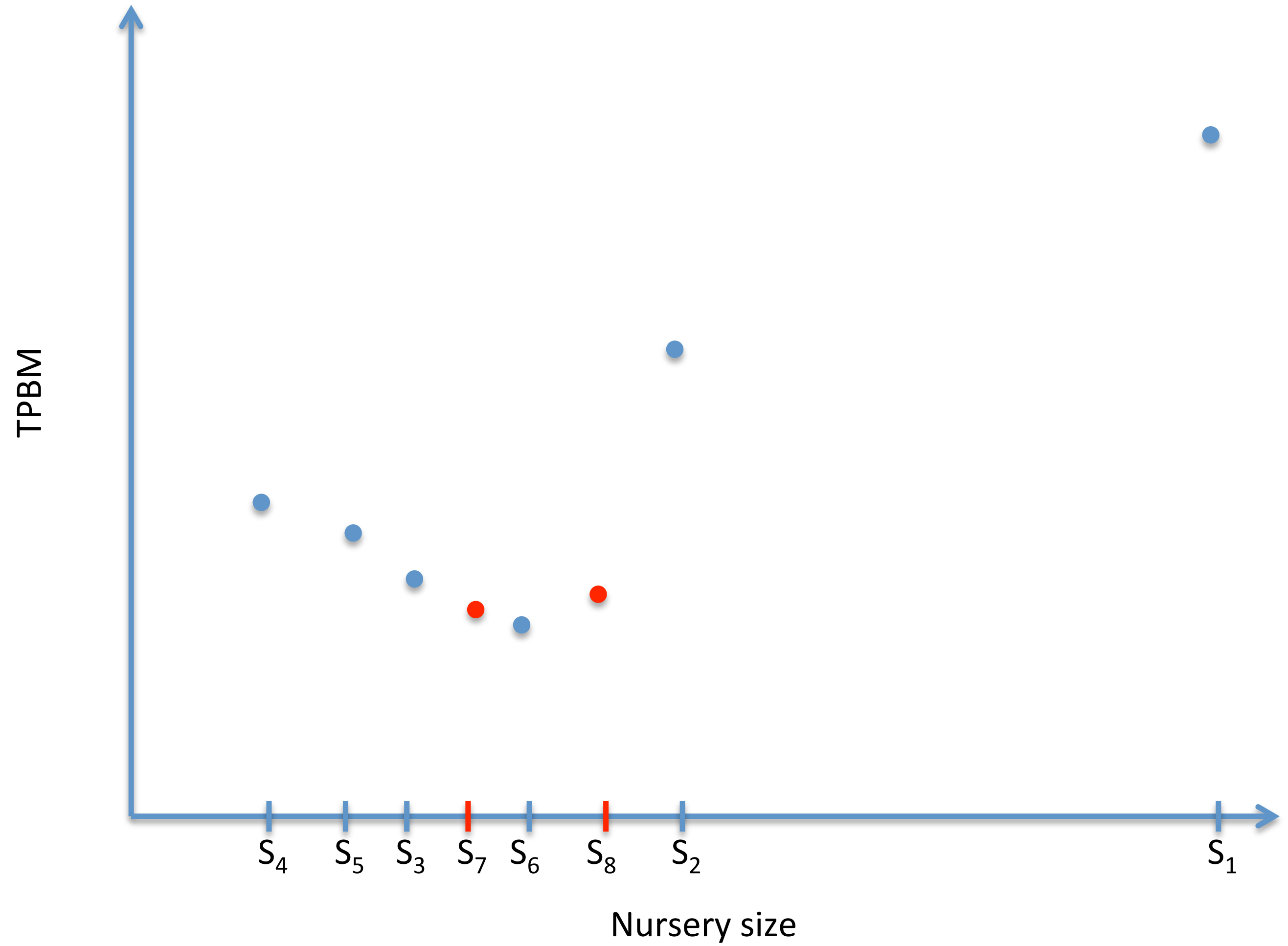


Example of TAA

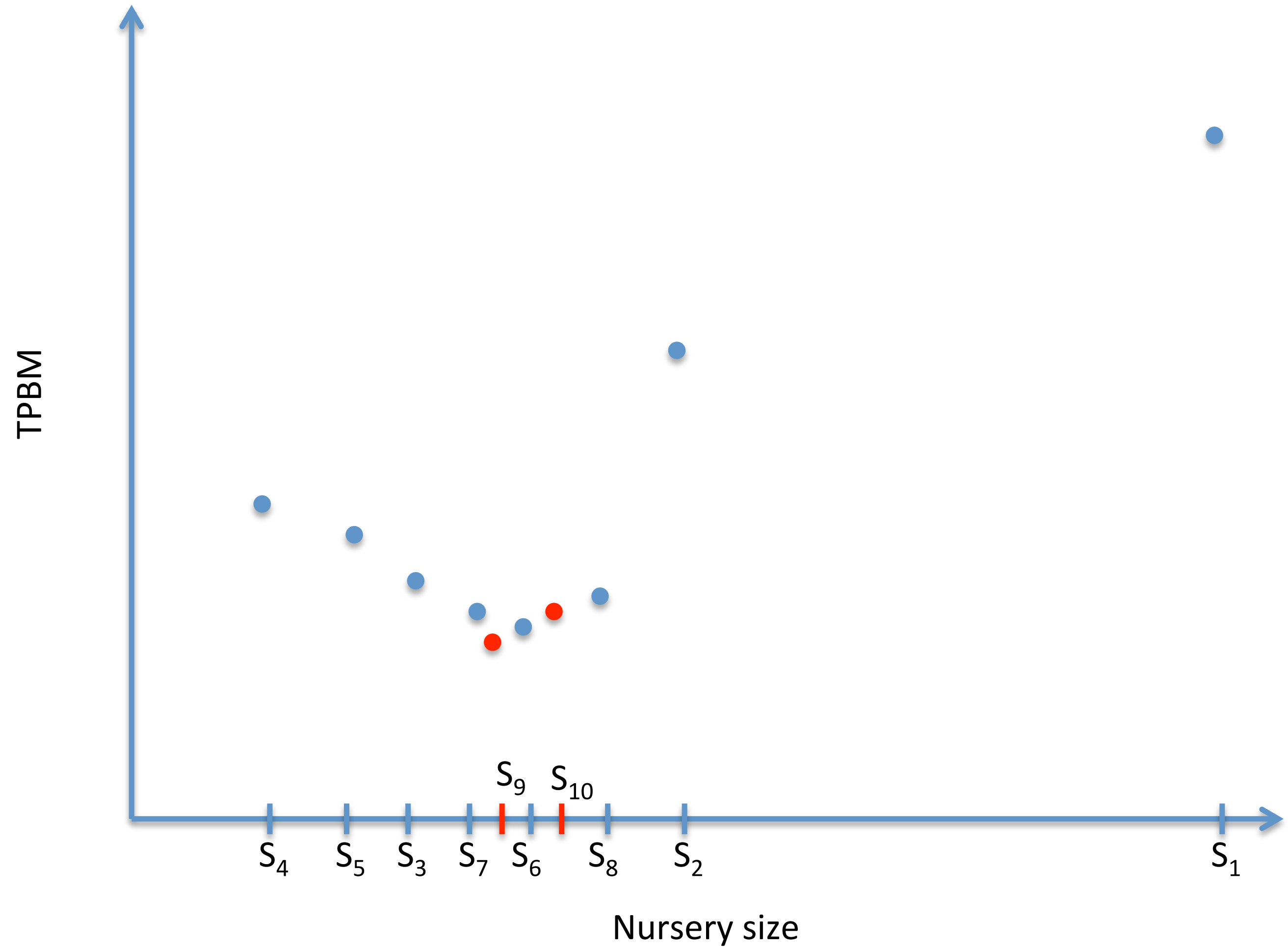


Example of TAA

University of St Andrews



Example of TAA



Binary-trees under different algorithms (compared to GHC default)

GC configuration	Speedup
-A2m	1.44
-A8m	1.69
-A64m	1.78
-H	1.38
TAA	1.72

Speedup against default of 0.5MB fixed nursery (-A500k)

Improving TAA (TAA^{+})

University of St Andrews

- Key assumption is that collection time is a good measure of cache locality
  - However, we have seen that this is not always the case

- Simple modification:

  Instead of just collection time, let

  TPBM = collection time *+ mutator time*

  where mutator time is time elapsed from the last collection to the current one

HORIZON 2020

REPHRASE

Binary-trees under different algorithms (compared to GHC default)

University of St Andrews

GC configuration	Speedup
-A2m	1.44
-A8m	1.69
-A64m	1.78
-H	1.38
TAA	1.72
TAA+	1.79

Speedup against default of 0.5MB fixed nursery (-A500k)

HORIZON 2020

REPHRASE

Drawbacks of TAA and TAA+

- Reacts to changes in program memory behaviour by *guessing* whether to increase or decrease nursery
- Reacts *after* these changes happen
- Can be very slow in responding to changes
  - Nursery size may have to be adjusted several times in order to get the right value

Copying-based algorithm (SLR)

- Use the amount of copied data in each collection (in relation to the nursery size) to estimate the performance

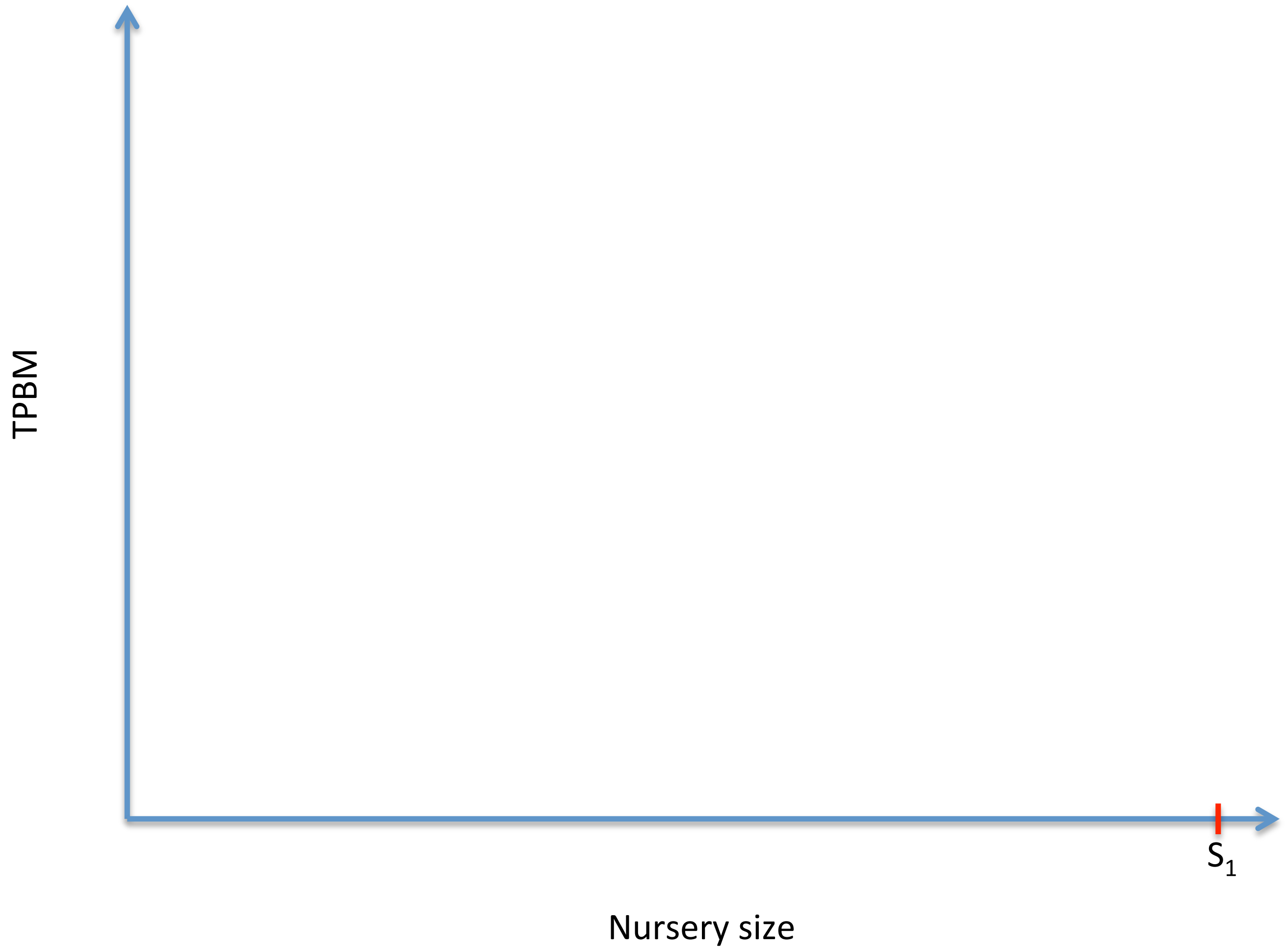
- Objective: find the *optimum* ratio of nursery size to live data, and then use it to calculate the nursery size
  - Nursery Size to Live Data Ratio (*SLR*)

- Starting with the initial *SLR*, modify it slightly after each garbage collection (and update the nursery size appropriately), until we find the right value

SLR algorithm

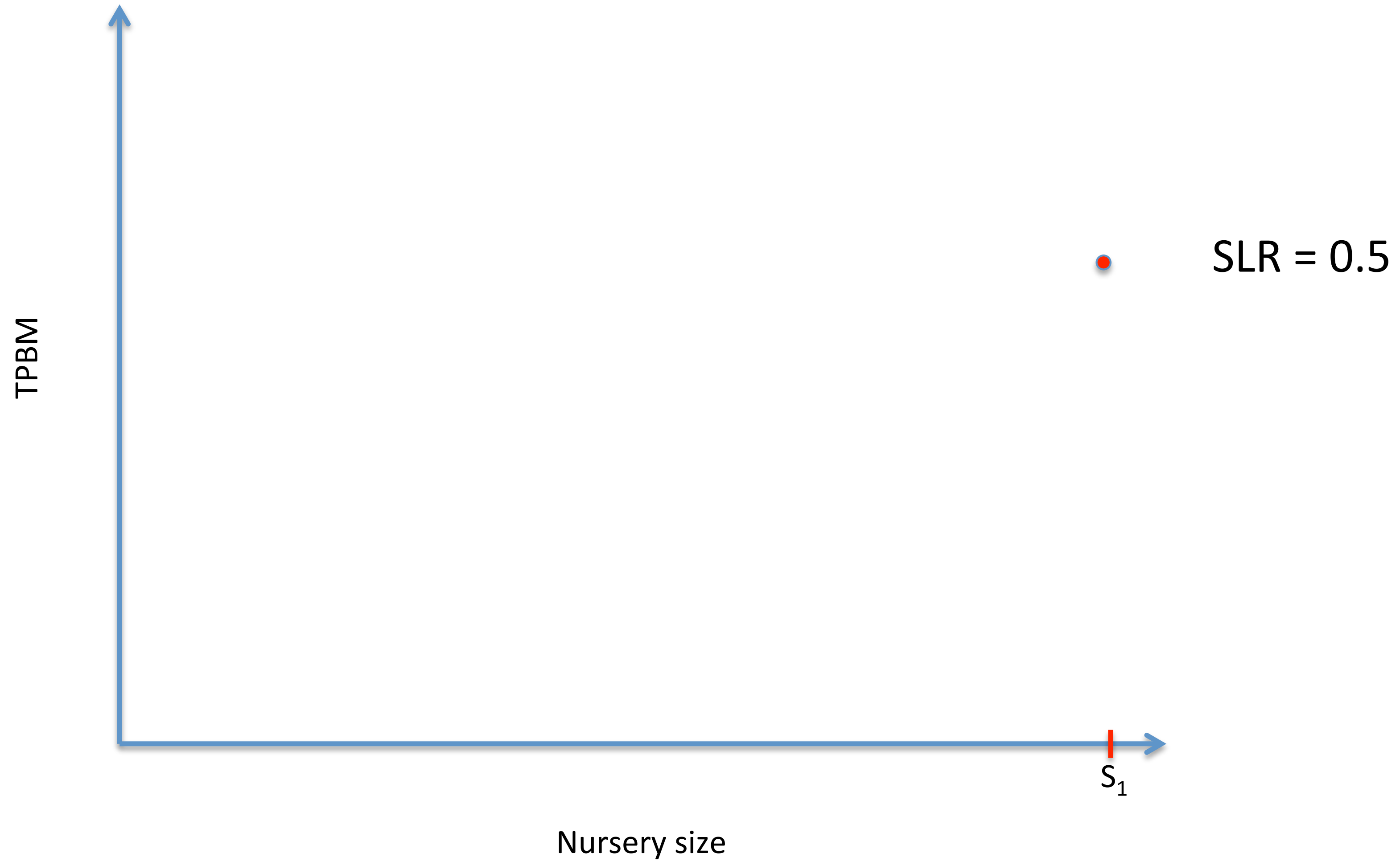
```
fun resize()
    TPBM_{n-1} = TPBM_n
    TPBM_n = (MUTTime_n + GCTime_n)/S_n
    if abs(TPBM_n - TPBM_{n-1}) < threshold then
        update_factor = update_factor_0 // reset value
        SLR_n = SLR_{n-1}
    else
        // if performance is worse, reverse
            update direction
        if TPBM_{n-1} > TPBM_n then
            update_factor = -0.9 × update_factor
        end
        SLR_n = SLR_{n-1} × (1 + update_factor)
    end
    return SLR_n × copied_n
end
```

Example of SLR

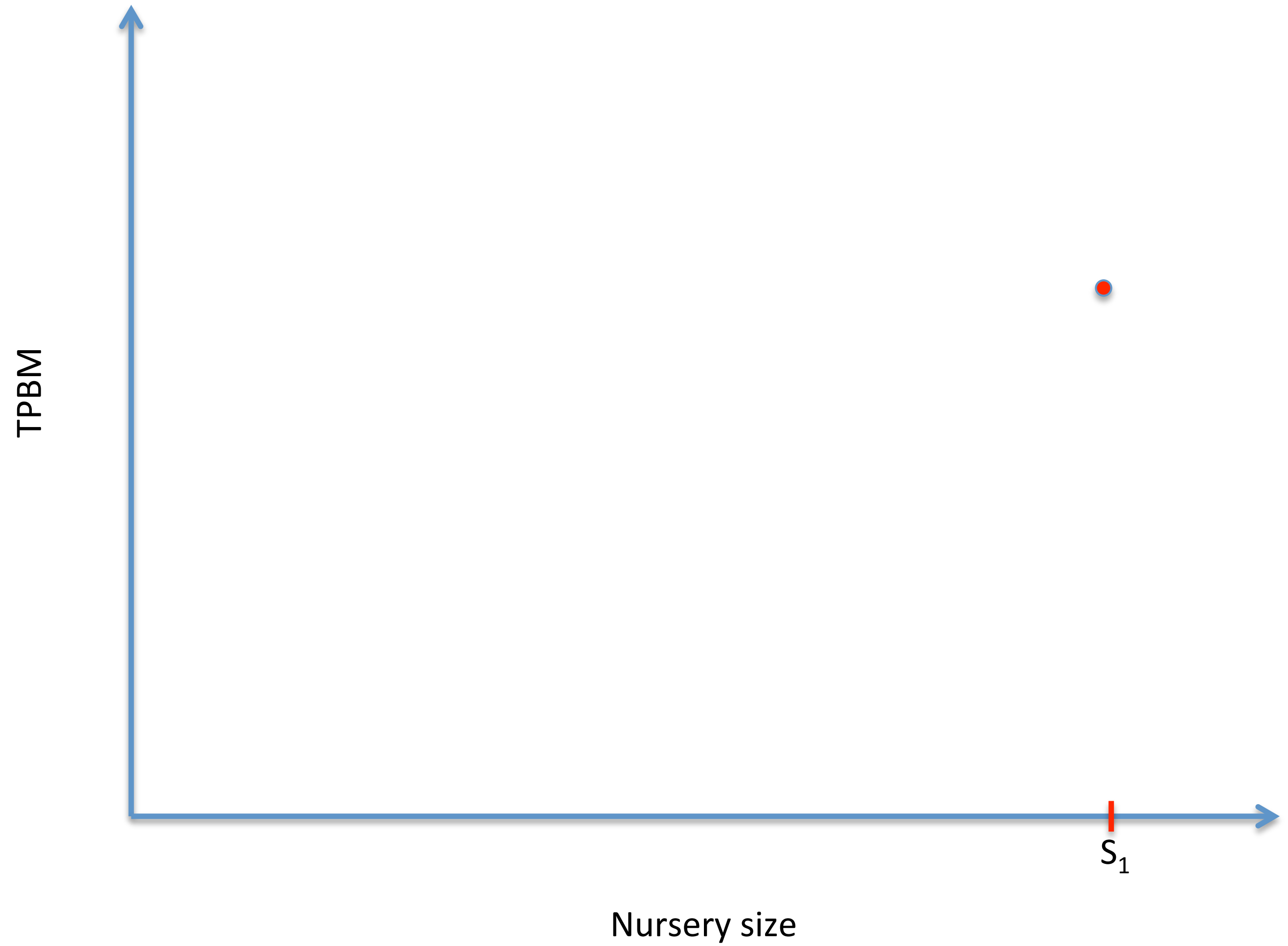


SLR = 0.5

Example of SLR



Example of SLR



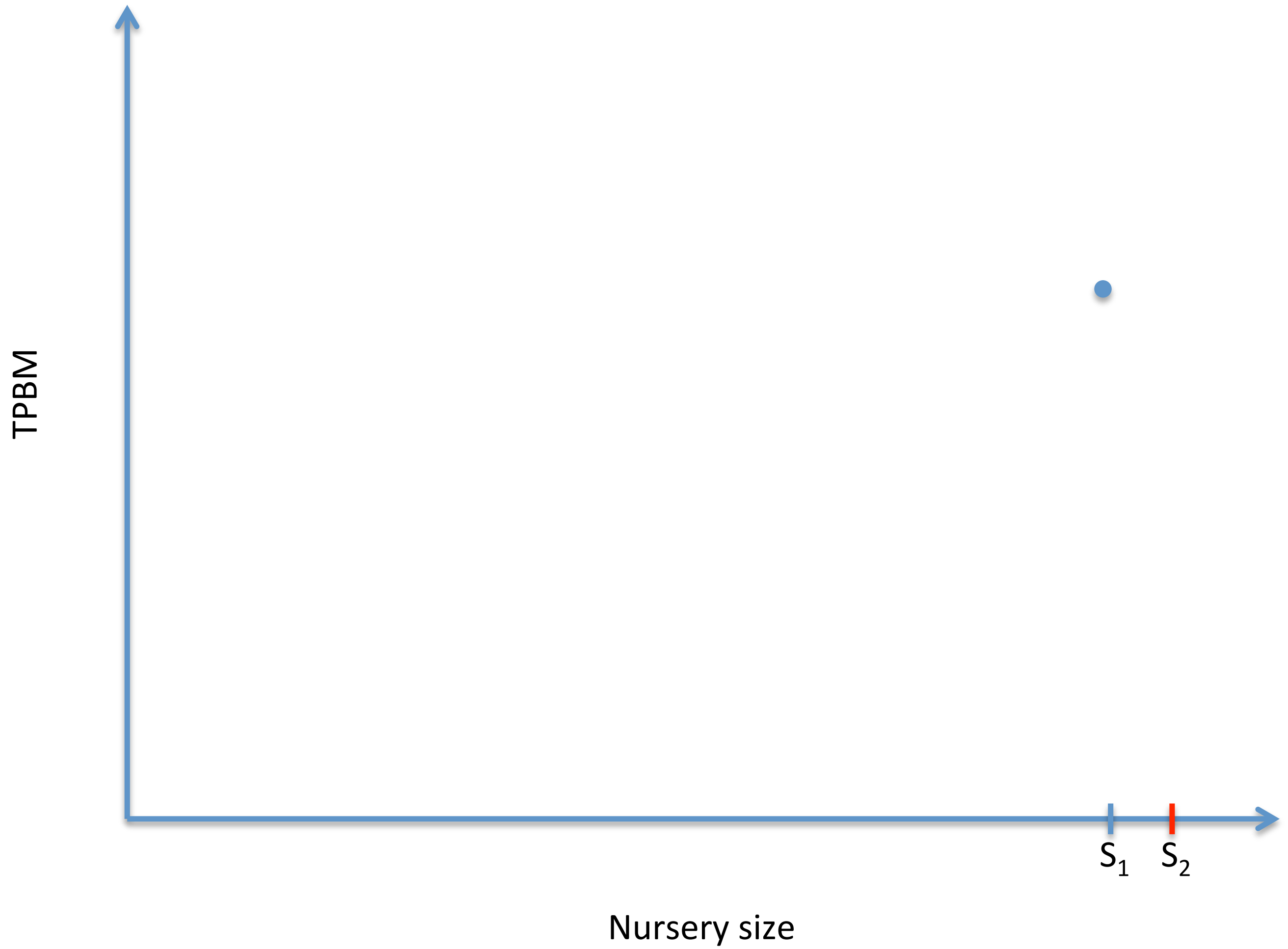
SLR = 0.5

S1 = 1MB

SLR = 0.45, C = 0.5MB

S2 = 1.11MB

Example of SLR



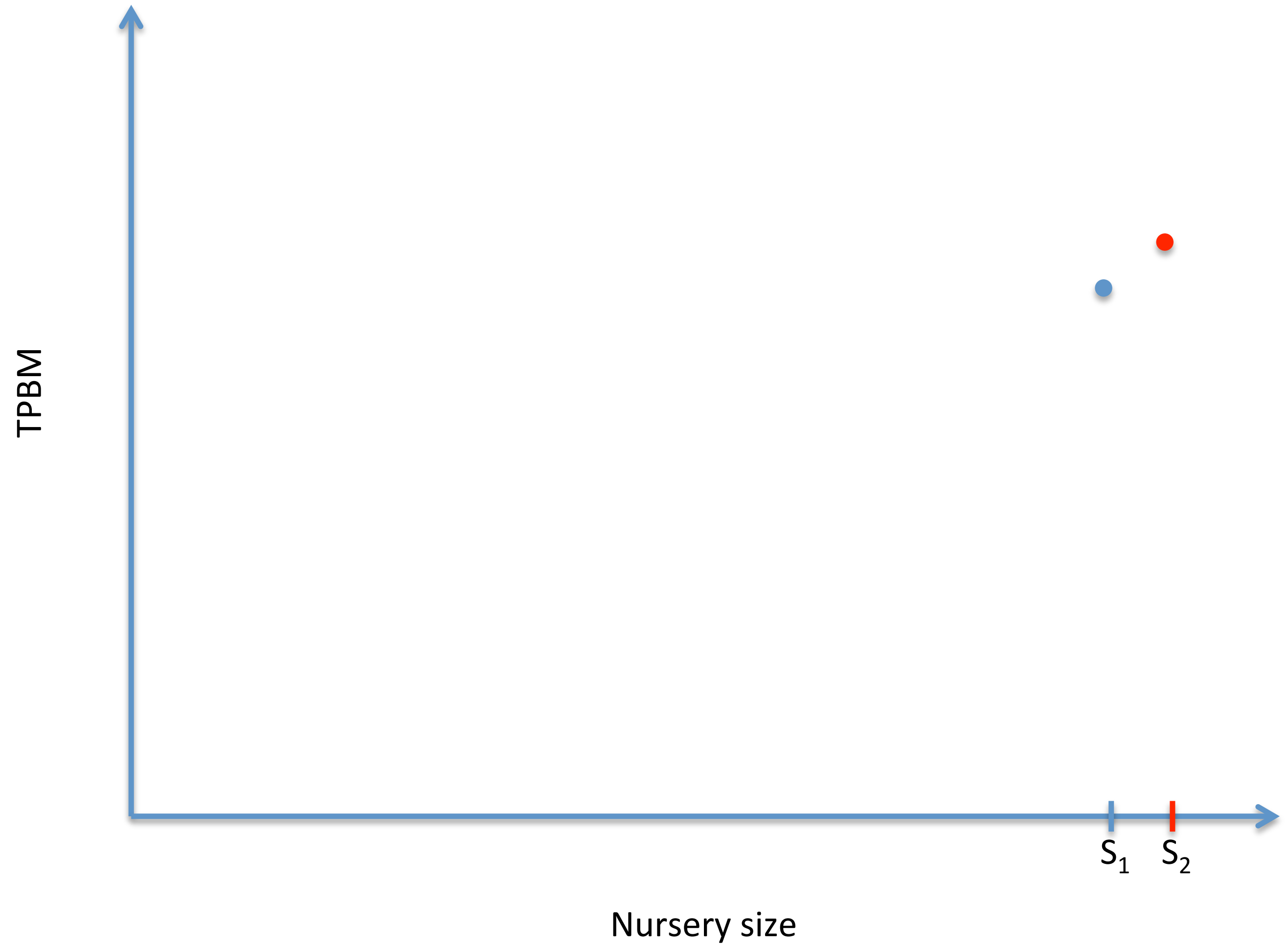
SLR = 0.5

S1 = 1MB

SLR = 0.45, C = 0.5MB

S2 = 1.11MB

Example of SLR



SLR = 0.5

S1 = 1MB

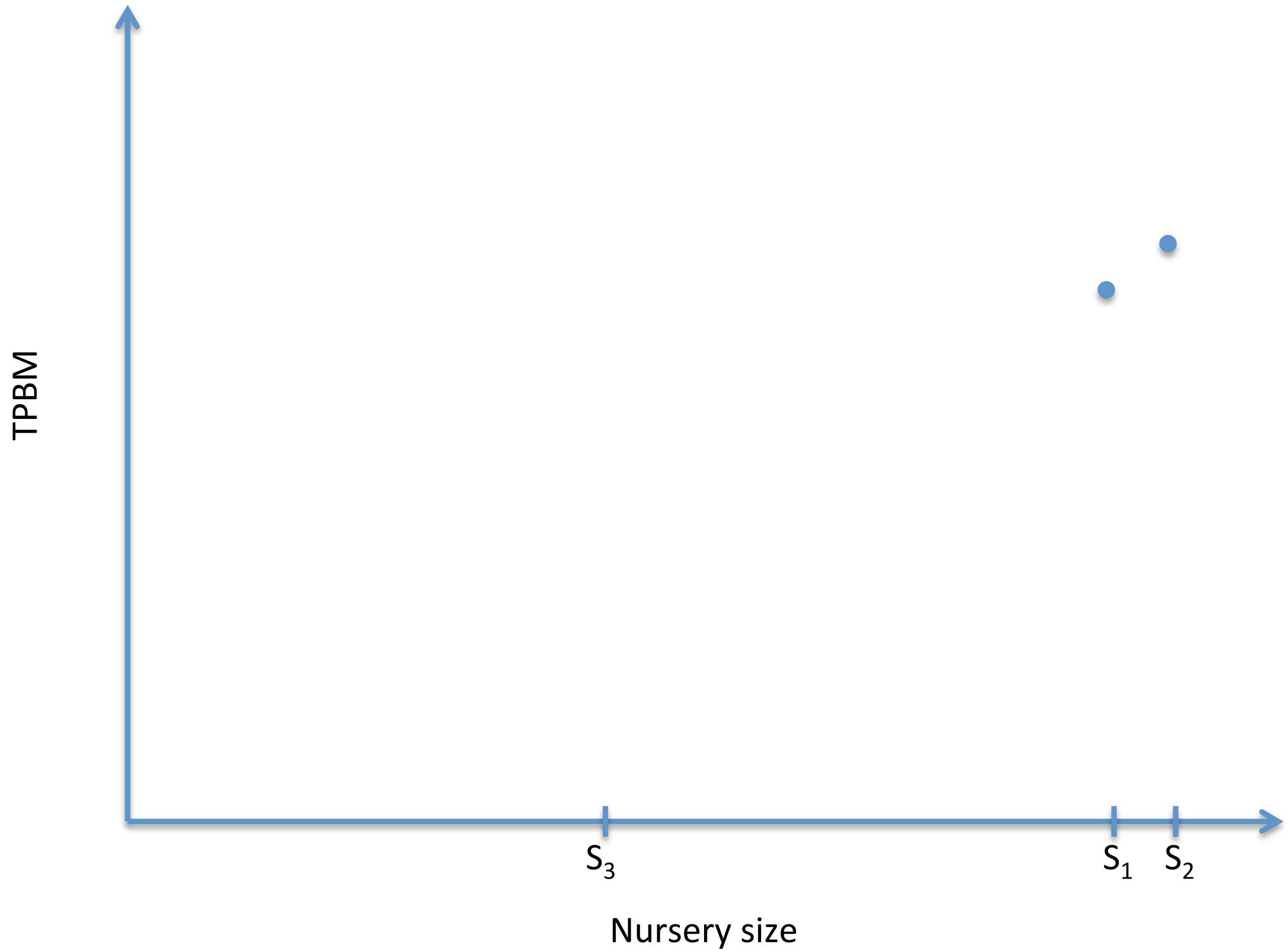
SLR = 0.45, C = 0.5MB

S2 = 1.11MB

SLR = 0.55, C = 0.3MB

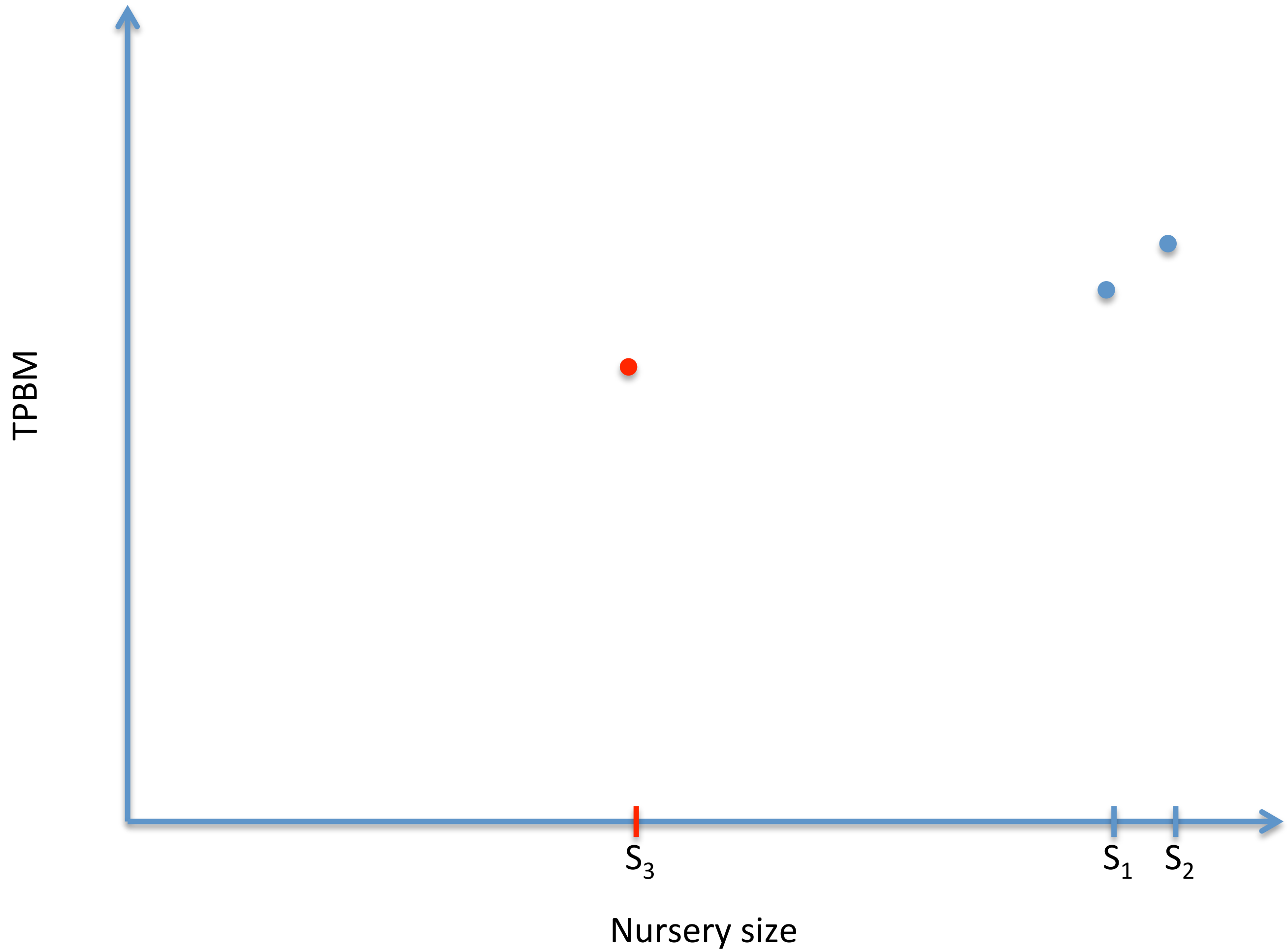
S3 = 550k

Example of SLR



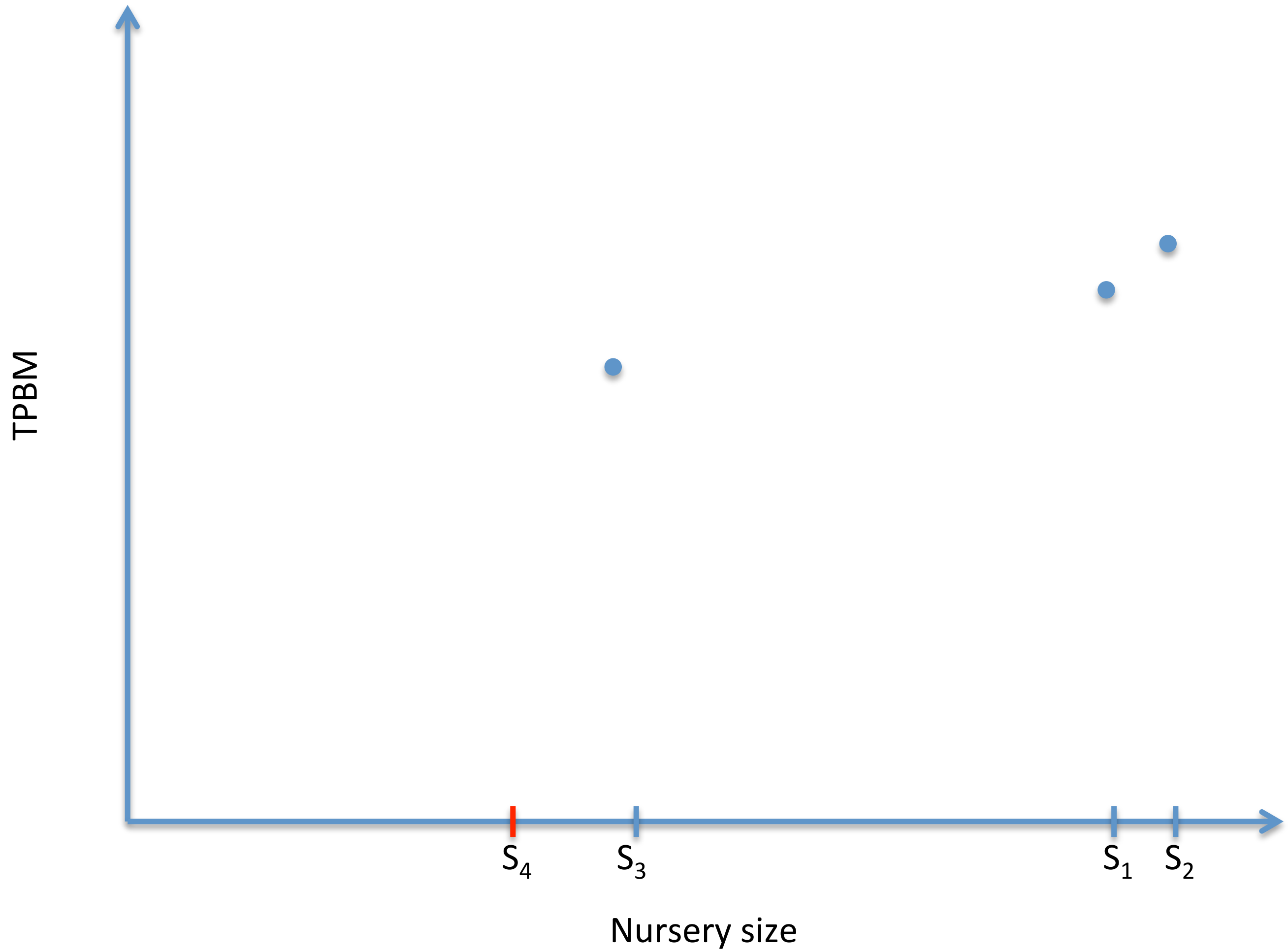
SLR = 0.5
S1 = 1MB
SLR = 0.45, C = 0.5MB
S2 = 1.11MB
SLR = 0.55, C = 0.3MB
S3 = 550k

Example of SLR



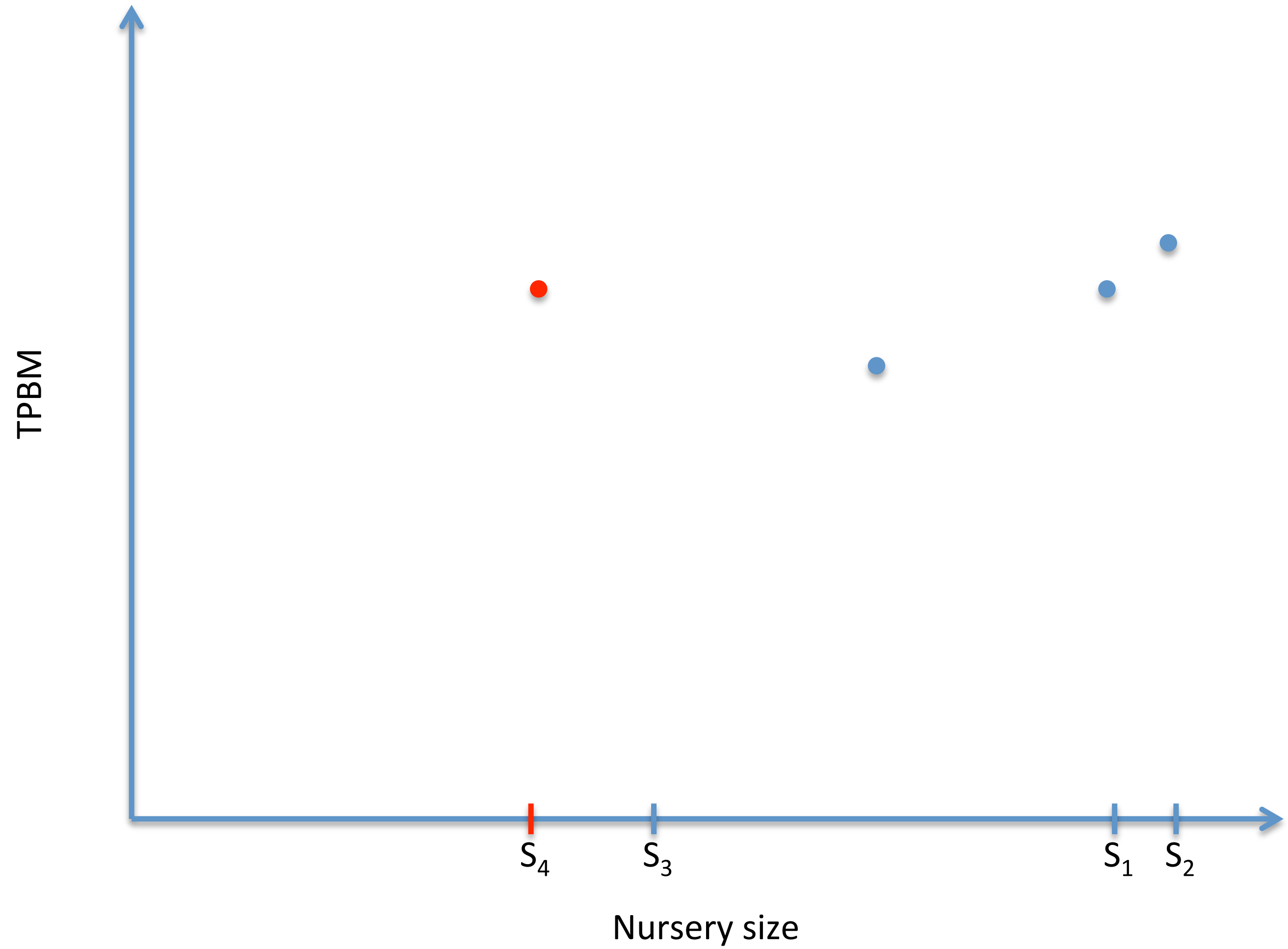
SLR = 0.5
S1 = 1MB
SLR = 0.45, C = 0.5MB
S2 = 1.11MB
SLR = 0.55, C = 0.3MB
S3 = 550k
SLR = 0.60, C = 0.3MB
S4 = 500k

Example of SLR



SLR = 0.5
  S1 = 1MB
SLR = 0.45, C = 0.5MB
  S2 = 1.11MB
SLR = 0.55, C = 0.3MB
  S3 = 550k
SLR = 0.60, C = 0.3MB
  S4 = 500k

Example of SLR



SLR = 0.5

S1 = 1MB

SLR = 0.45, C = 0.5MB

S2 = 1.11MB

SLR = 0.55, C = 0.3MB

S3 = 550k

SLR = 0.60, C = 0.3MB

S4 = 500k

SLR = 0.57, C = 0.2MB

S5 = 350k

Improvement with SLA

GC configuration	Speedup
-A2m	1.44
-A8m	1.69
-A64m	1.78
-H	1.38
TAA	1.72
TAA^+	1.79
SLR	1.96

Speedup against default of 0.5MB fixed nursery (-A500k)

Evaluation of TAA, TAA^+ and SLR

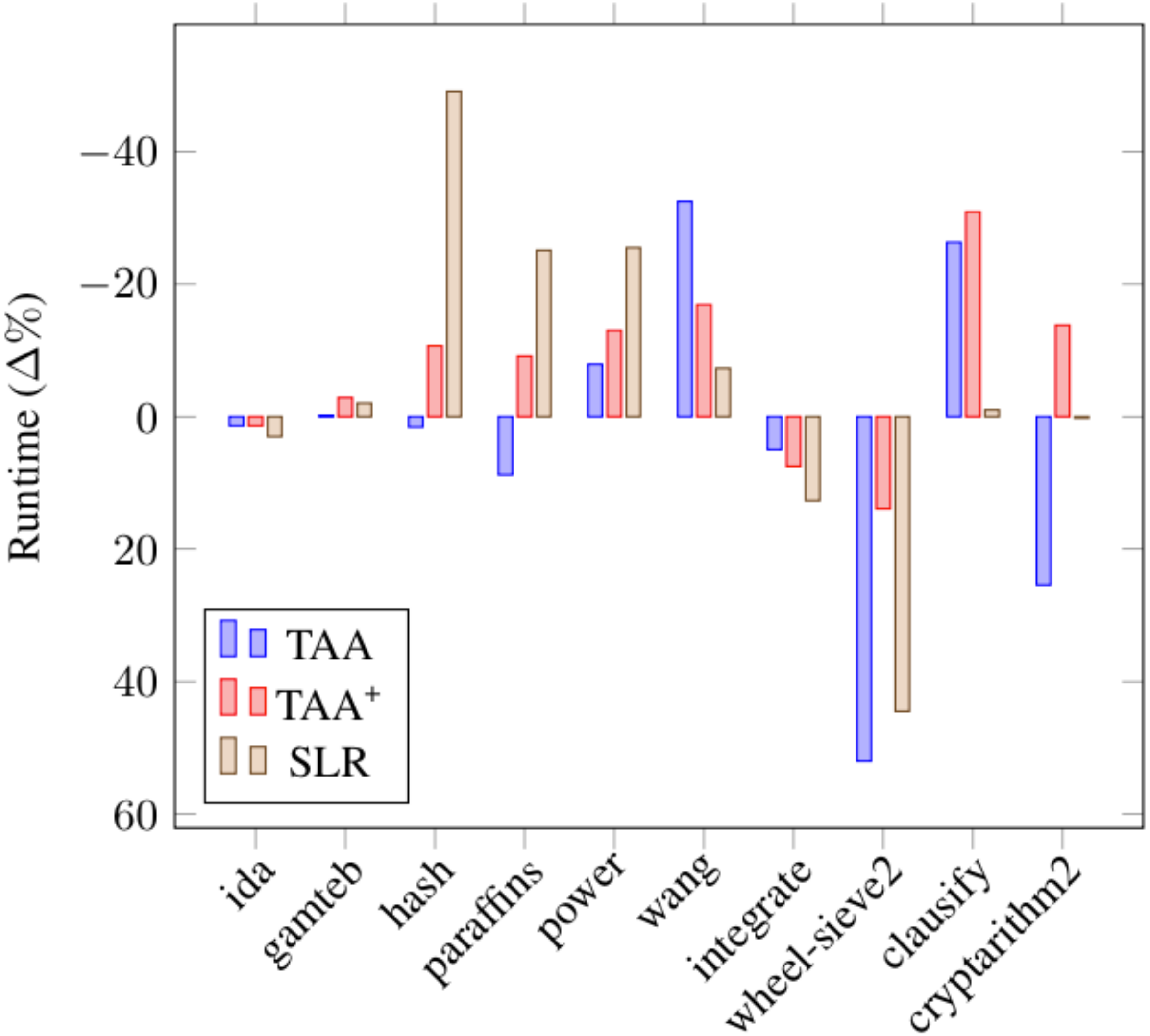
Evaluated on the *nofib* benchmark suite of 63 Haskell benchmarks (real, spectral, imaginary)

Performance	TAA	TAA^+	SLR
unaffected	58.0	51.6	61.3
positive	19.4	30.6	33.9
negative	22.6	17.7	4.8

SLR is the best default option

- Average improvement in runtime of 10%
- The best improvement 88.5%
- Worst case gives slowdown of 44.5% (in just one example)
  - compared with 52.0% for TAA, 28.2% for TAA^+

Main Affected Benchmarks



SLR v. TAA

University of St Andrews

- Both can suffer from “initial slowdown”, where nursery size is incorrectly guessed at the beginning
- Relevant information (TPBM) calculated *after* collections, and cannot be obtained *beforehand*
- SLR adapts better to memory usage changes, since nursery size is always modified in proportion to the amount of copied data

Conclusions

University of St Andrews

- Nursery size can have a significant impact on the performance of functional programs
- We have established a relation between nursery size and the execution time of a program
  - The interplay between cache locality and the amount of data copied during the collection
- Introduced two novel algorithms for dynamic tuning of the nursery size: TAA^{+} and SLR
  - SLR gives the best overall performance, and is a sensible default for GHC

HORIZON 2020

REPHRASE

Future Work

- Quantify memory irregularity and relate it to improvements in execution time
- Study influence of other factors on GC performance and include these in more sophisticated
  - e.g. amount of data accessed
- Investigate TAA+ and SLR for imperative languages
  - e.g. C++, Java
- Dynamic nursery resizing algorithms for parallel programs

University of St Andrews

THANK YOU!

http://rephrase-ict.eu

@rephrase_eu