June 17, 2016 - PLDI

Assessing the Limits of Program-Specific GC Performance

Nicholas Jacek, Meng-Chieh Chiu, Ben Marlin, and Eliot Moss

{njacek,joechiu,marlin,moss}@cs.umass.edu
University of Massachusetts, Amherst

What this talk is about

- Garbage Collection (GC)
- Lower bounds on GC cost
- Not generic, but for a particular program run
- After-the-fact analysis – not a mechanism
- Optimal for full-heap GC
- Approximately optimal for generational GC
- Optimization methods from machine learning
- Some tricks to make optimization practical

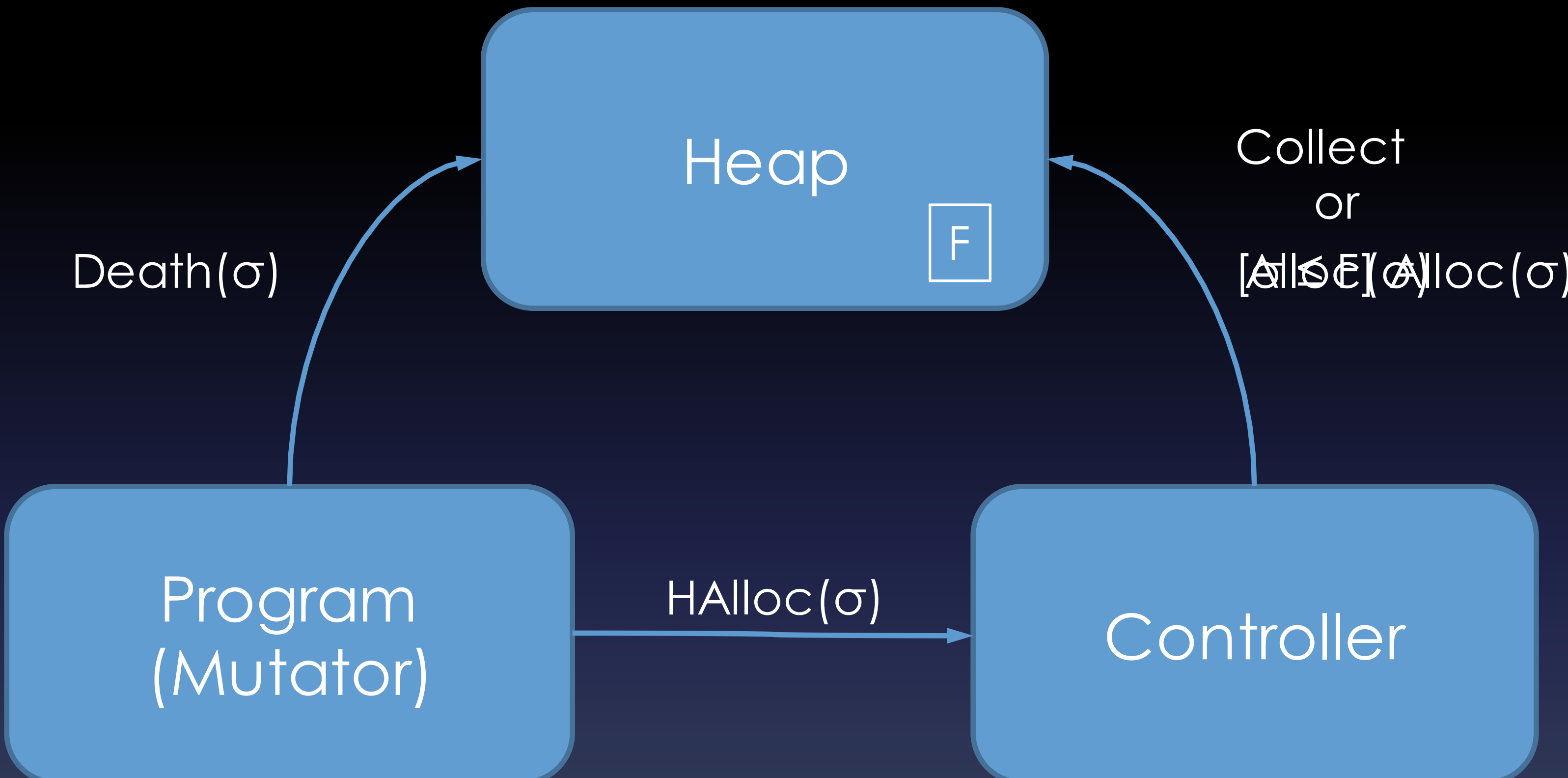
Program-Specific GC

- Existing GC performance bounds framed in terms of best a GC algorithm can do in the face of *any possible* program behavior
  - Argued by devising an adversial program
- If we tune GC *to a program*, or even to a *program run*, the problem is different:
  - What is the best we can do in the face of a particular sequence of allocations and object deaths, given heap space *S* ?

Why is this interesting?

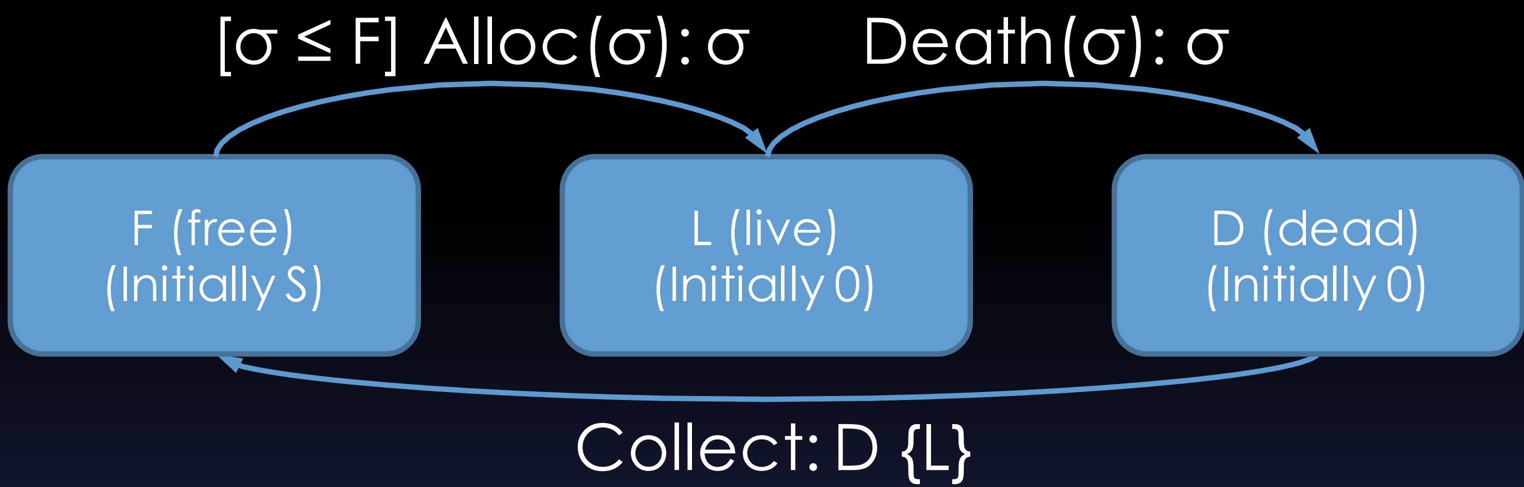
- In our larger research project we aim to tune GC for individual programs.
  - How do we know how well we are doing?
  - Suppose we can indicate x% improvement over some existing scheme. Is there more to be had?
  - If we fail to see improvement, could it be because very little is possible?
- Determines whether, and maybe for which programs, this tuning might be interesting.

Model of GC



[enabling predicate] Action(parameters)

Heap states, action effects



[enabling predicate] Action(parameters): # of bytes {cost}

- Invariant: $F + L + D = S$ (heap size)
- Actions change the state of bytes
- Visible to controller: F (*not* L or D)

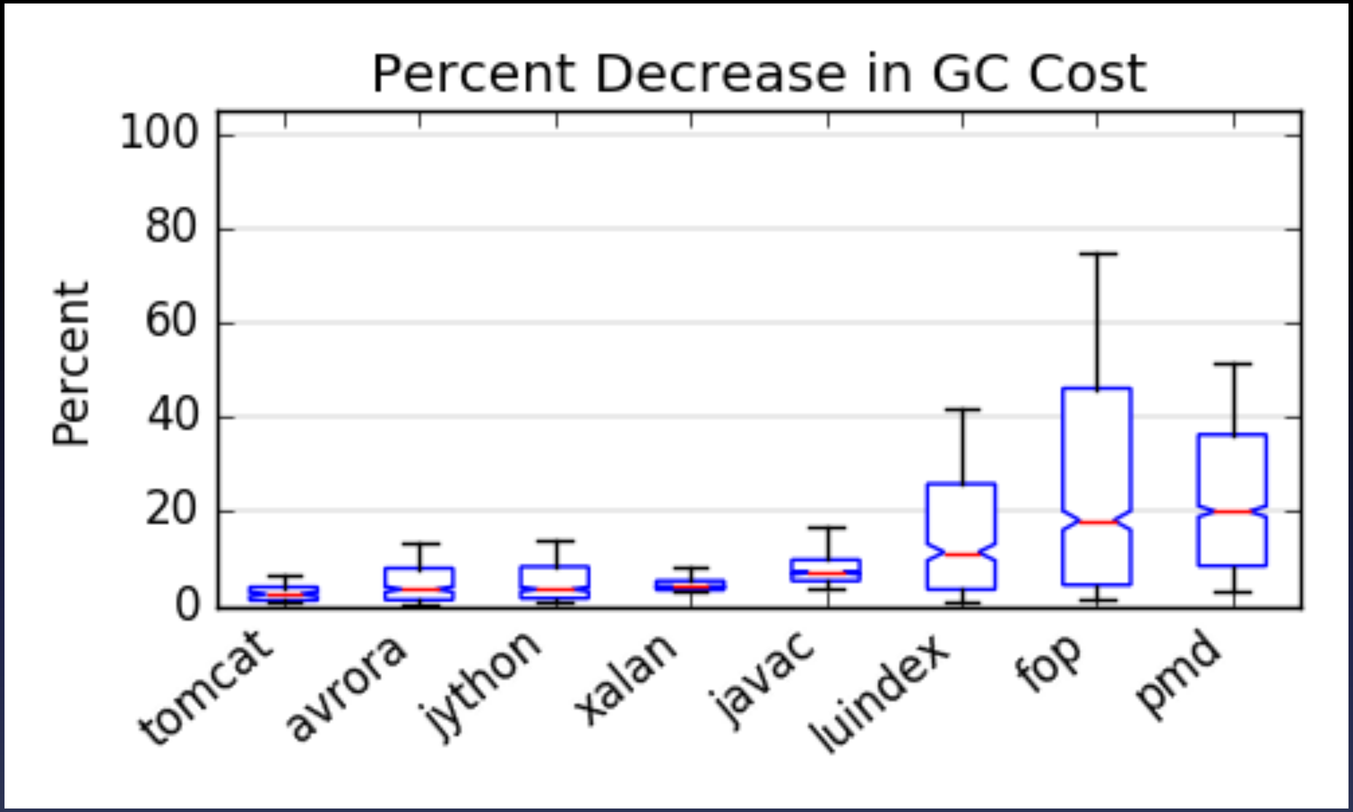
Cost model and optimization

- Cost of a GC taken as proportional to L (live bytes)
  - More or less true for tracing / copying collectors
- Since the live size at any given point during a program's execution is a fixed property of the execution, cost to collect at time *t* is a constant
  - The Markov Decision Process is *first-order* – does not depend on the history of prior decisions
- Therefore, <u>*dynamic programming*</u> will find an <u>*exact solution*</u> to determining the optimal GC schedule – places to collect to obtain minimal total cost – <u>*for a particular trace*</u> *(program execution)*

Solution Cost

- N = number of units (objects) allocated
- Cost to solve is $O(N^2)$
  - Assuming you have the live size at each point
- Can refine to $O(H \cdot N)$ where H is the heap size
  - Can look back/forward at most $O(H)$ allocations

Cost reduction: Full GC



See the paper for more Full GC results

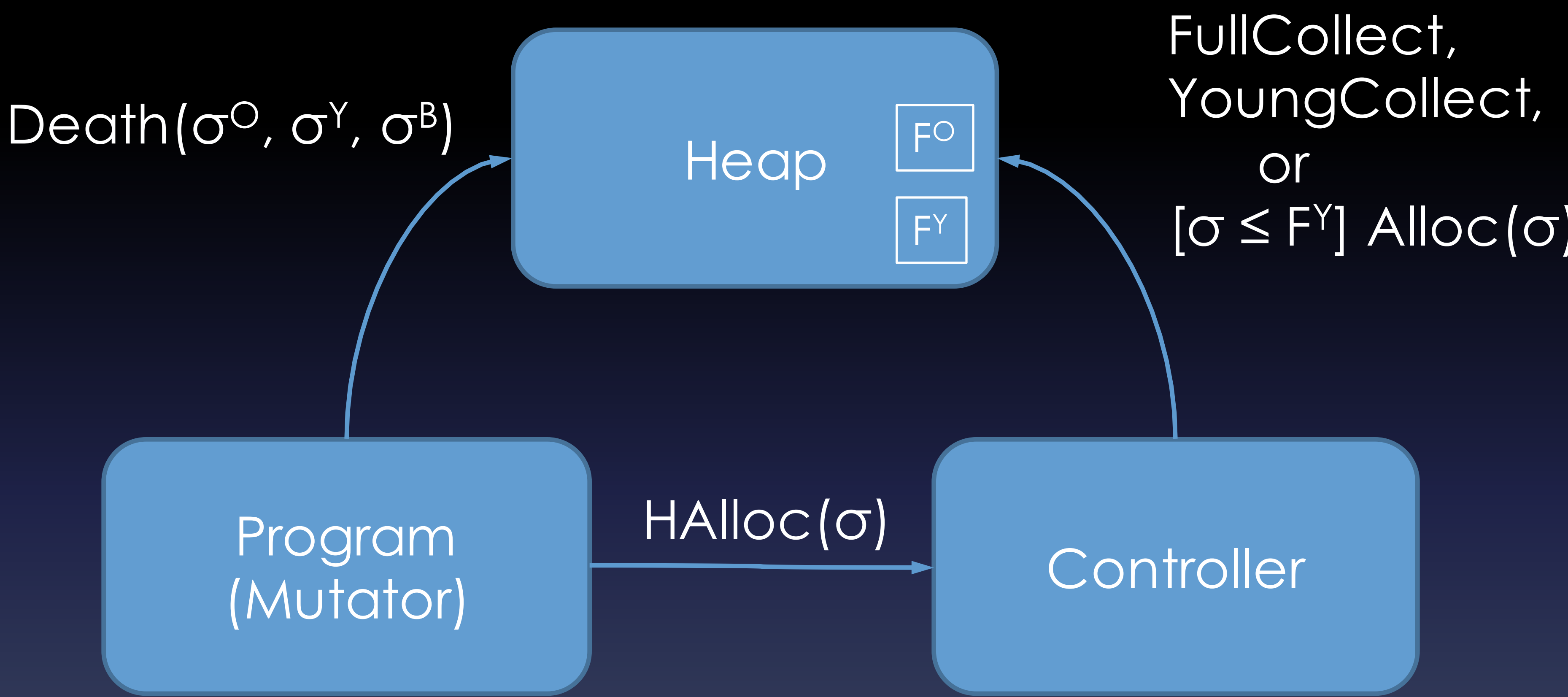
Making optimization practical

- Group allocations into blocks, say 256 Kb
  - Reduces N by a factor of (say) 5000
  - Does this by constraining when GC occurs
  - Smaller blocks do not change things much
- Pre-analyze object connectivity
  - Identify objects treated the same by GC
  - Summarize behavior in three numbers
  - No detailed simulation while optimizing!

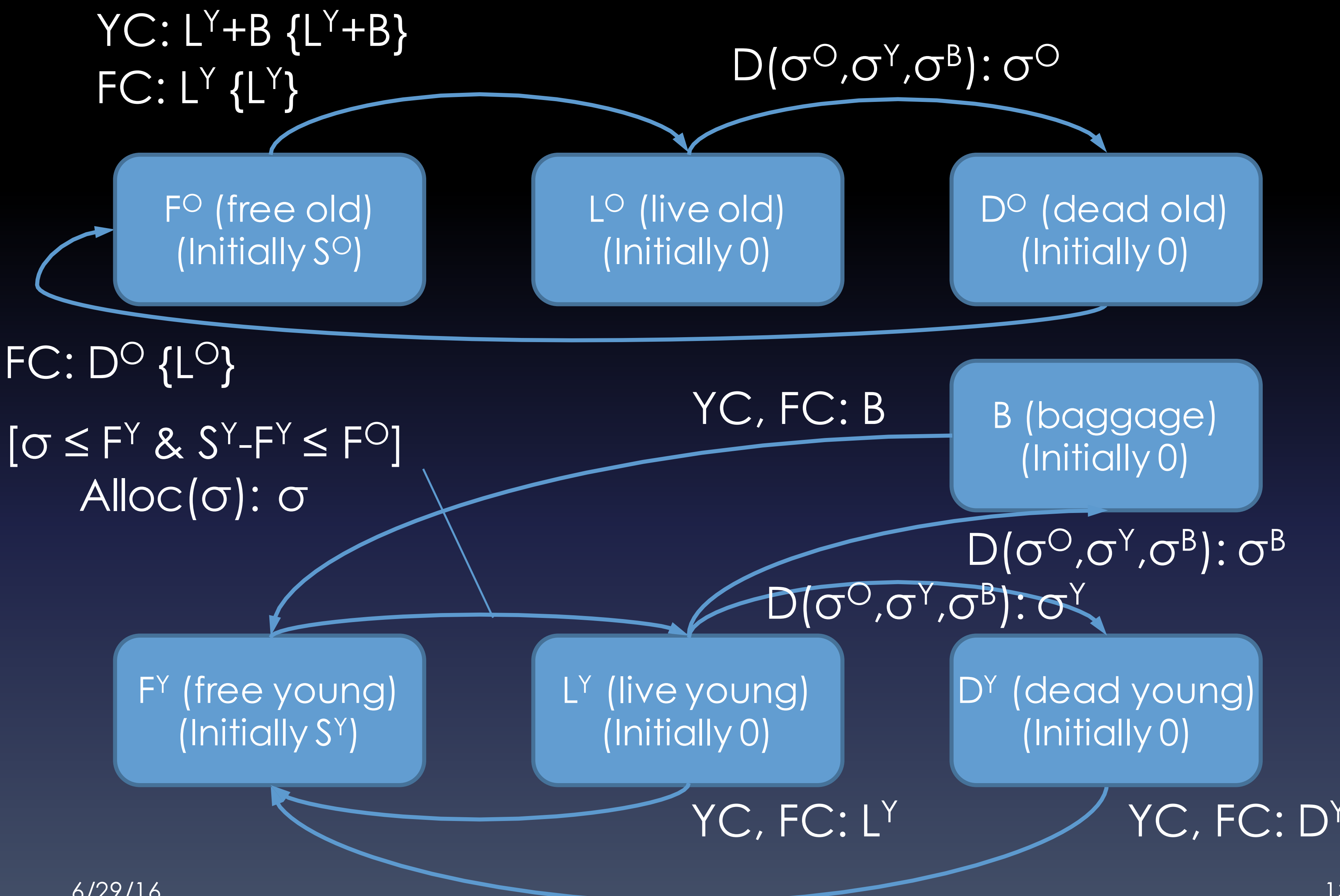
Generational Collection

- Splits heap into *young* and *old* portions
- Allocation goes into the young generation
- Full collection finds liveness of *all* objects
- Young collection *assumes* old objects live
  - Thus treats some dead young objects as if live
  - We call these *baggage*
- Young collection promotes apparently live young objects to old space

Model with Gen GC



States, effects with Gen GC



Impact on Optimization

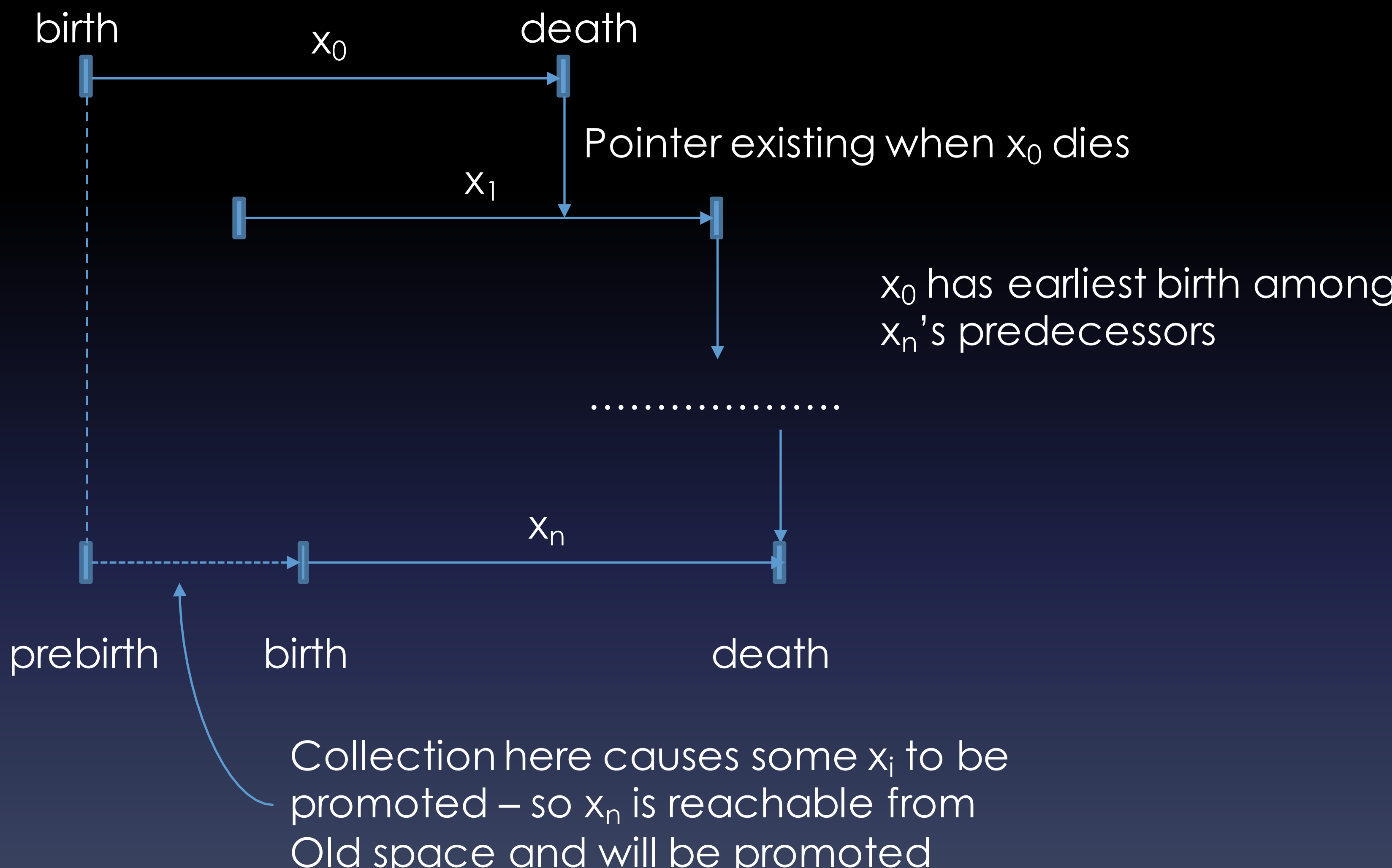
- Full GC cost, *and resulting state*, depends only on time of collection
- This is *not* true for Young GC!
  - Previous promotions affect whether a dying object goes to D^Y or B …
  - Which later affects cost (and future promotion)

- We *approximate* by considering visible states based on current time and time of previous collection

Computing Cost Efficiently

- For a given trace, can *precompute* necessary reachability information, avoiding simulation
- We have (birth, death) for each object
  - Precise death comes from Elephant Tracks' analysis
- Add *prebirth*, giving (prebirth, birth, death)
- GC of object in Young space at t > death causes promotion if previous GC was between prebirth and birth
- Aggregate blocks of objects by (p,b,d)

Prebirth Time



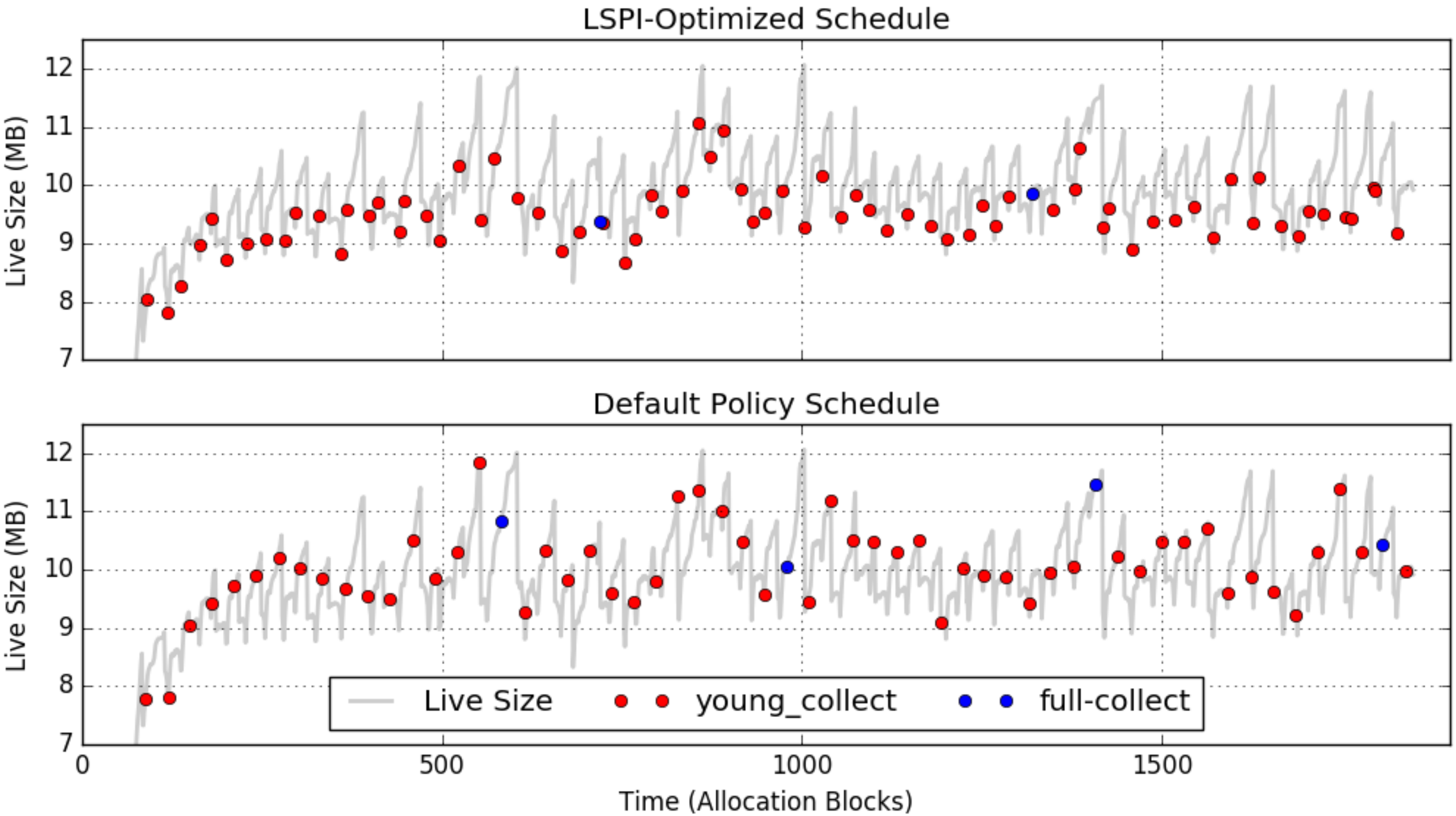
Gen GC Optimization

- Develop collection of sample states:
  - (F^Y, F^O, t_{last}, t_{now}) where t_{last} is time of most recent GC
  - Cost of legal actions (noC, YC, FC) in each state
  - Use "collect when full" to get to each t_{last}
  - Only a sample – full search space *huge!*
- Treat as Markov decision process
  - Conflation of states treated as randomness
- Apply *Least Squares Policy Iteration*

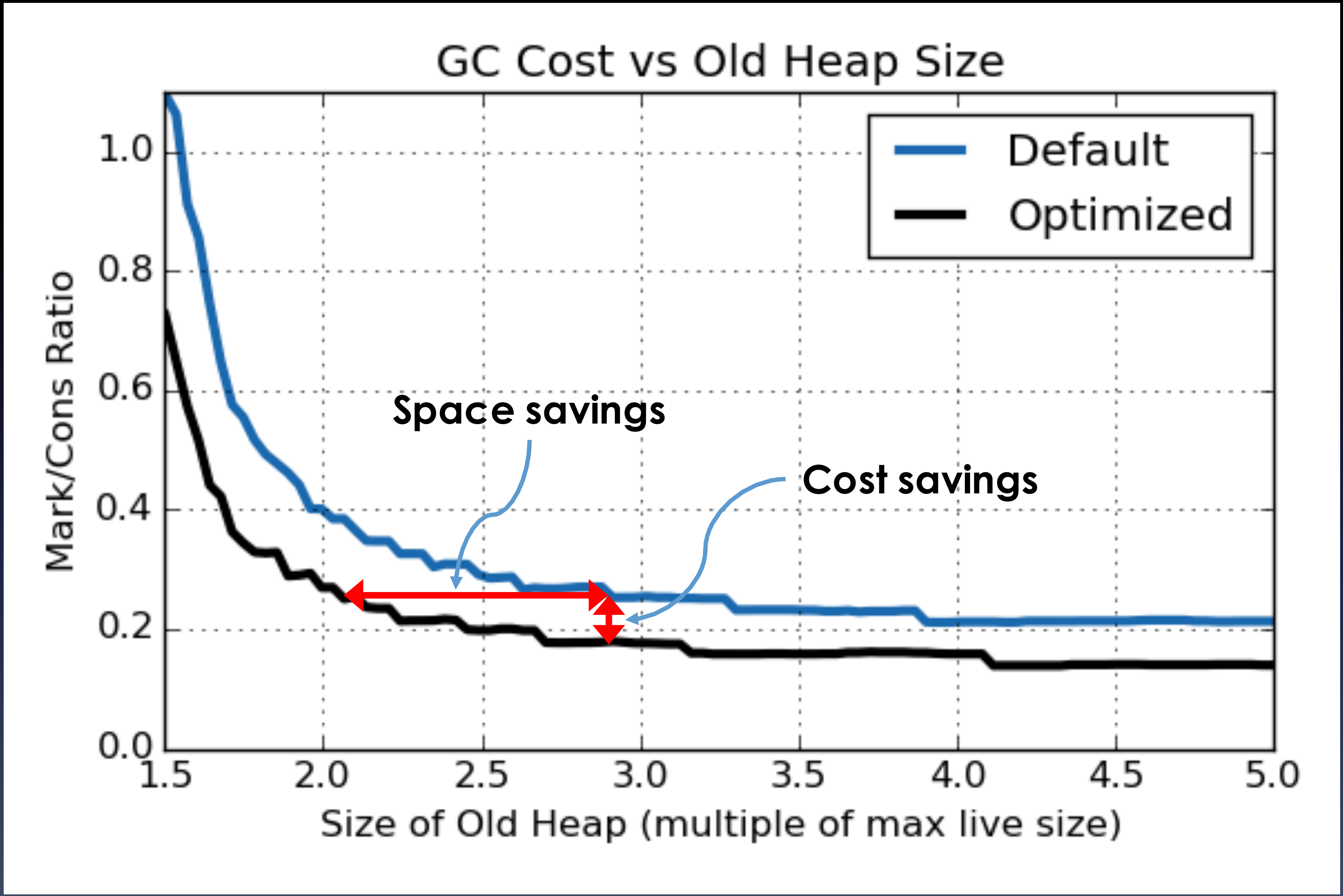
Experiments

- 7 programs from DaCapo suite + javac
- For most of them, added more inputs
- About 10^2 to 10^4 256 Kb blocks
- Ran under Elephant Tracks to generate traces of alloc, death, pointer updates
- Computed (prebirth,birth,death) times
- Aggregated into blocks
- Applied LSPI
- Compared with default: collect when full

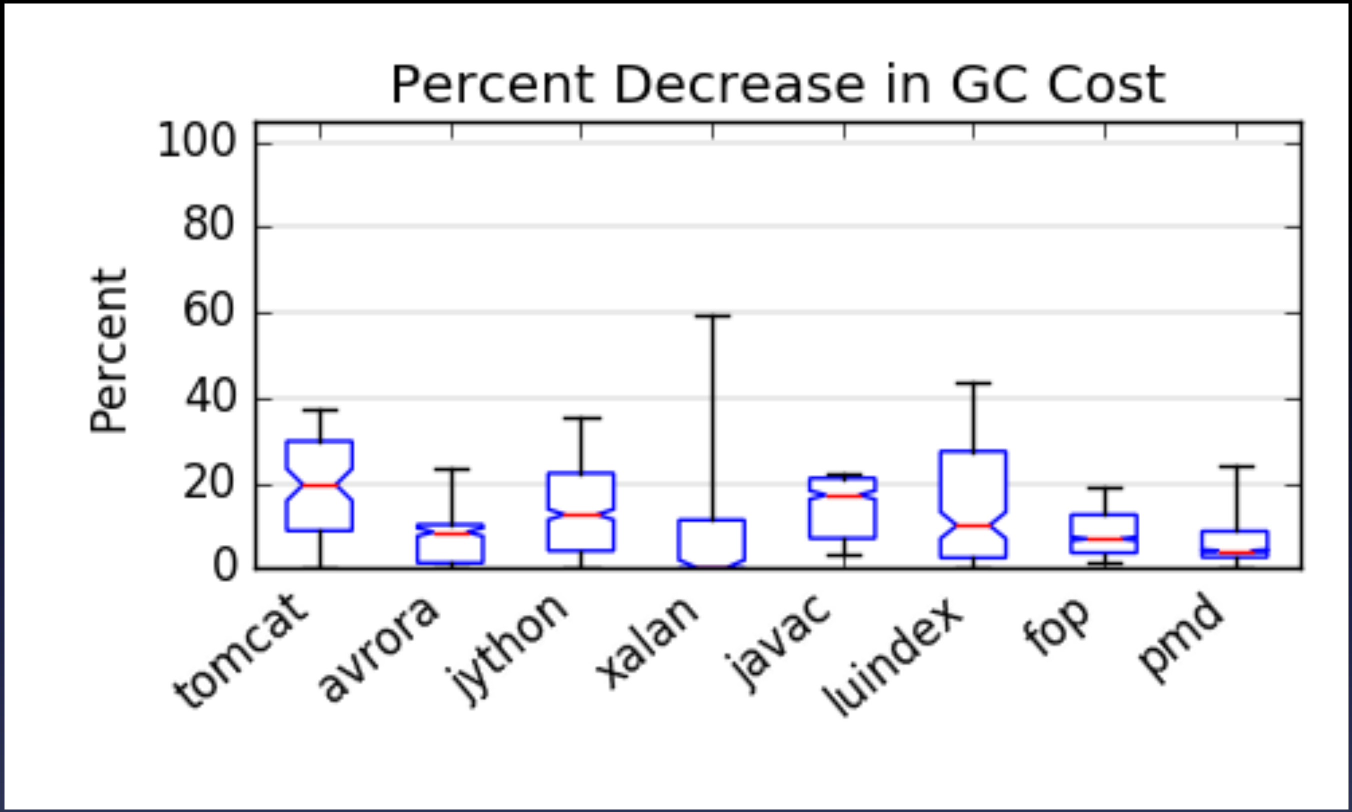
Results: Sample Schedules



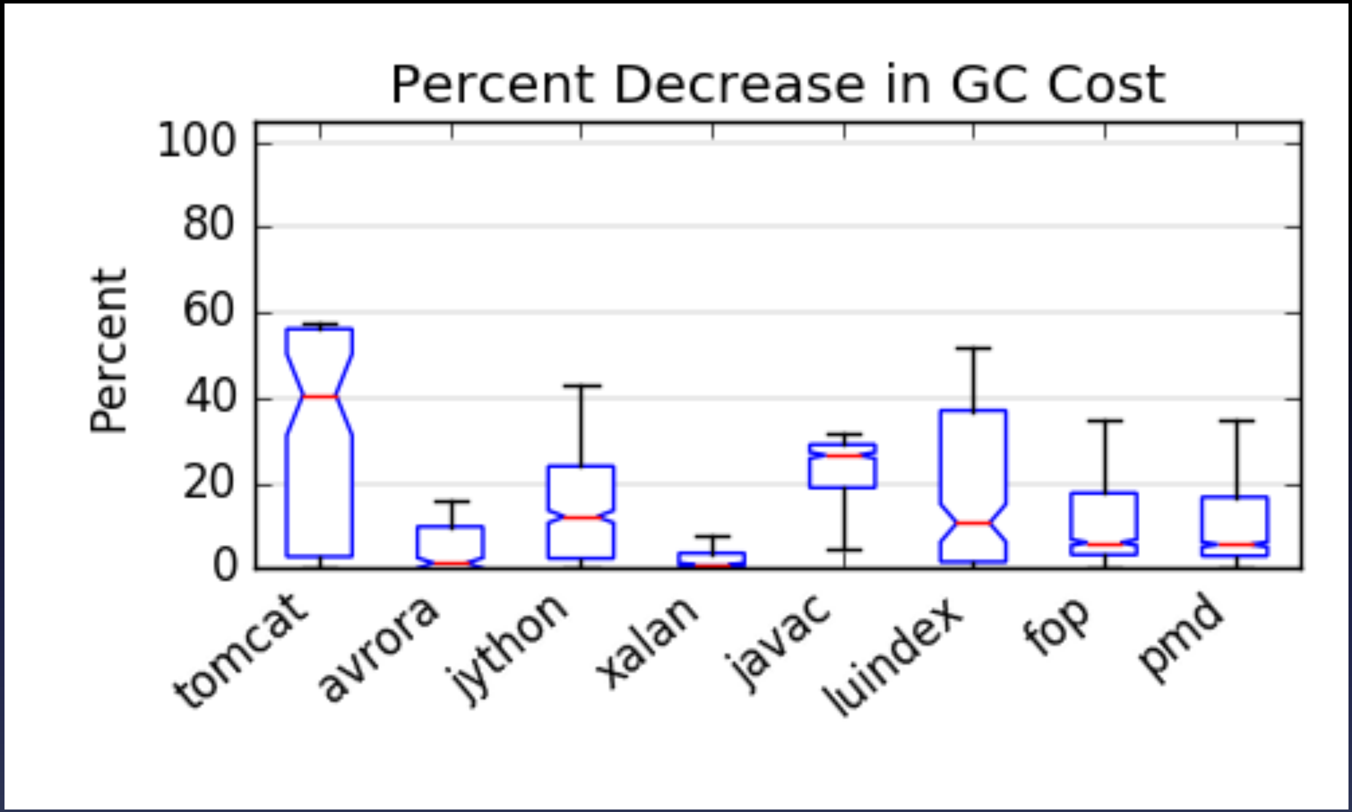
Cost vs Heap size (S^Y=8Mb)



Reduction: Gen GC (S^Y=4Mb)



Reduction: Gen GC (S^Y=8Mb)



Additional Results

- Improvement is not better for smaller blocks – more fragmentation, etc.
- Cost of optimization is ~ $O(N^{2.2})$
- More graphs in paper

Ongoing Work

- Program-specific policies
  - Reinforcement learning, or "deep" learning
    - Collect feature-rich traces
    - Select features
    - Train GC triggering mechanism
  - Some success for "self test"
  - Generalization (over runs, heap sizes) hard
- Hard lower bounds
  - Bound the baggage
  - Solve with dynamic programming

Conclusions

- First per-program GC cost bounds
- Program-specific GC policies promising
  - For at least some programs and heap sizes
  - Possibly useful reduction in GC time, or
  - Possibly substantial reduction in space
- Sometimes hard to improve over default
- Hard, not approximate, lower bounds would be welcome

Credits

- Nicholas Jacek – RL work
- Meng-Chieh (Joe) Chiu – ET work
- Ben Marlin – co-PI

- NSF Grant CCF-1320498

Markov Decision Process

- A set of *states*, *S*
- A set of *actions*, A
- Probabilistic transition function:

$$S_0 \xrightarrow{\text{a: p, cost}} S_1$$

Reinforcement Learning

- States $s \in S$, actions $a \in A$
- Transition function T: $S \times A \rightarrow S$
- "Policy" π: $S \rightarrow A$ (chooses action)
- Bellman equation: <u>*value*</u> of each action in each state, for given policy π:

  $Q^{\pi}(s,a) = c_0 + Q^{\pi}(s', \pi(s'))$, for $s' = T(s,a)$

  In matrix form:

  $$Q^{\pi} = C + \Pi^{\pi} Q^{\pi}$$

- Given C and Π^{π} can solve for Q^{π}

Policy Iteration (PI)

- Sequence of policies π_0, π_1, …
- Monotonically improving cost
- Determine Q^{π_m} for π_m
- Form $\pi_{m+1}(s) = \arg\min Q^{\pi_m}(s,a)$
  - That is, lowest cost action in each state, where remaining decision are as for π_m
  - This is no worse than π_m
- Iterate until reach convergence: π_*

Least Squares PI

- Approximate Q^π as linear combination of a fixed set of basis functions $\Phi_i(s,a)$
  - Weights θ_i
  - Matrix form: $Q^\pi = \Phi\ \theta^\pi$
- Solves with standard linear methods
- We use one basis function for each (s,a)
  - So: exact, not (additional) approximation
- Cost is $O(N^3)$ per iteration, worst case
  - $O(N^{2.2})$ in practice
  - Using blocks was an important choice!

Learning "Take Home"

- Use a big sample of states and cost
  - Approximation comes in this sampling
  - Our samples not random, but not full space
- Solve with standard linear methods
- Cost is $O(N^{2.2})$
- Not really learning – an optimization technique borrowed from RL