sociomantic

MAY 2013

CONCURRENT GARBAGE COLLECTION FOR D

LEANDRO LUCARELLA

INTRODUCTION

WHAT?

- Automatic memory management

WHAT FOR?

- Simplify interfaces
- Improve performance (!)
- Avoid memory errors
  - -Dangling pointers
  - -Memory leaks
  - -Double free

HOW?

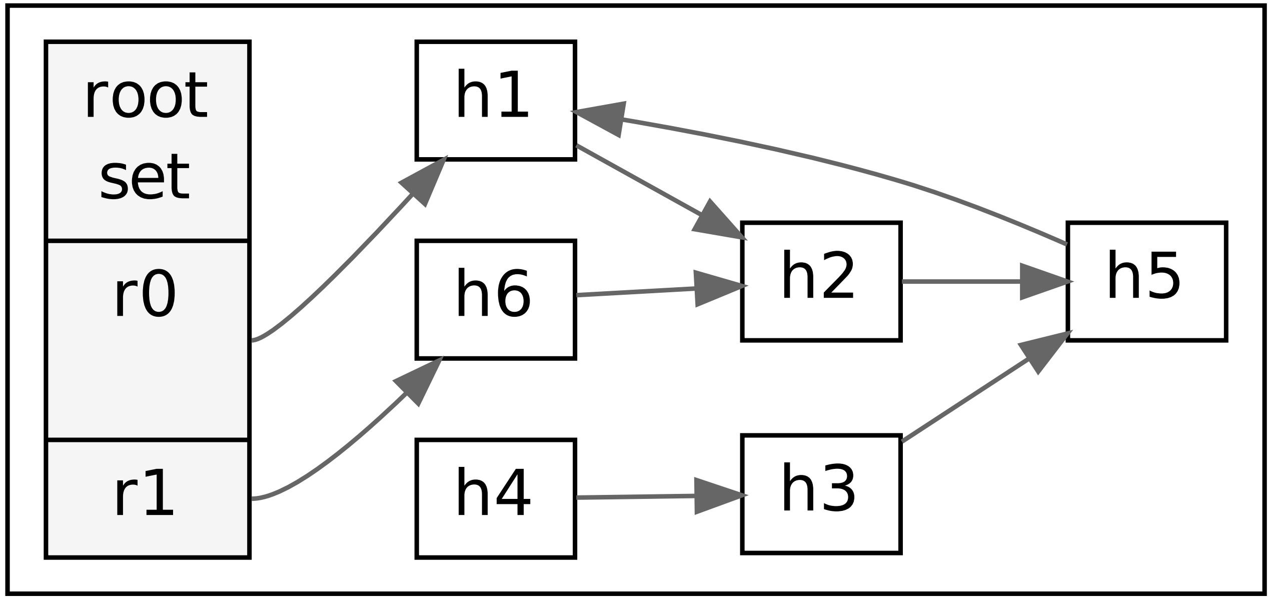
- Reference counting
- Semi-space copy
- Mark & sweep

CLASSIC ALGORITHMS

- Reference counting
- Semi-space copy
- Mark & sweep

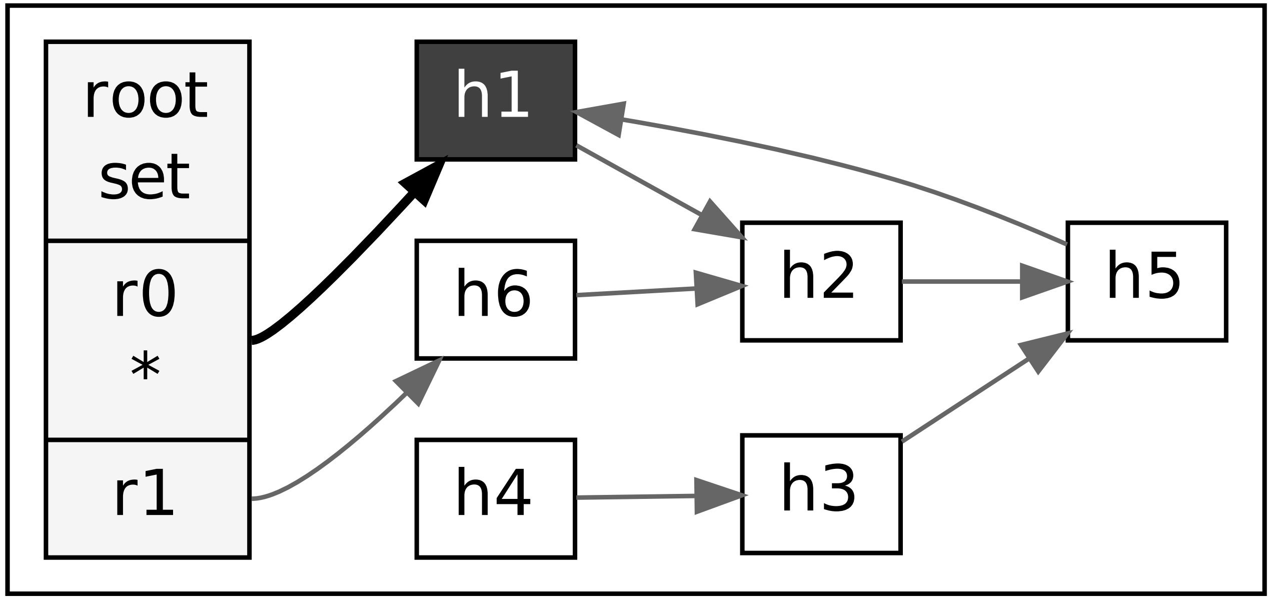
CLASSIC ALGORITHMS

- Reference counting
- Semi-space copy
- Mark & sweep



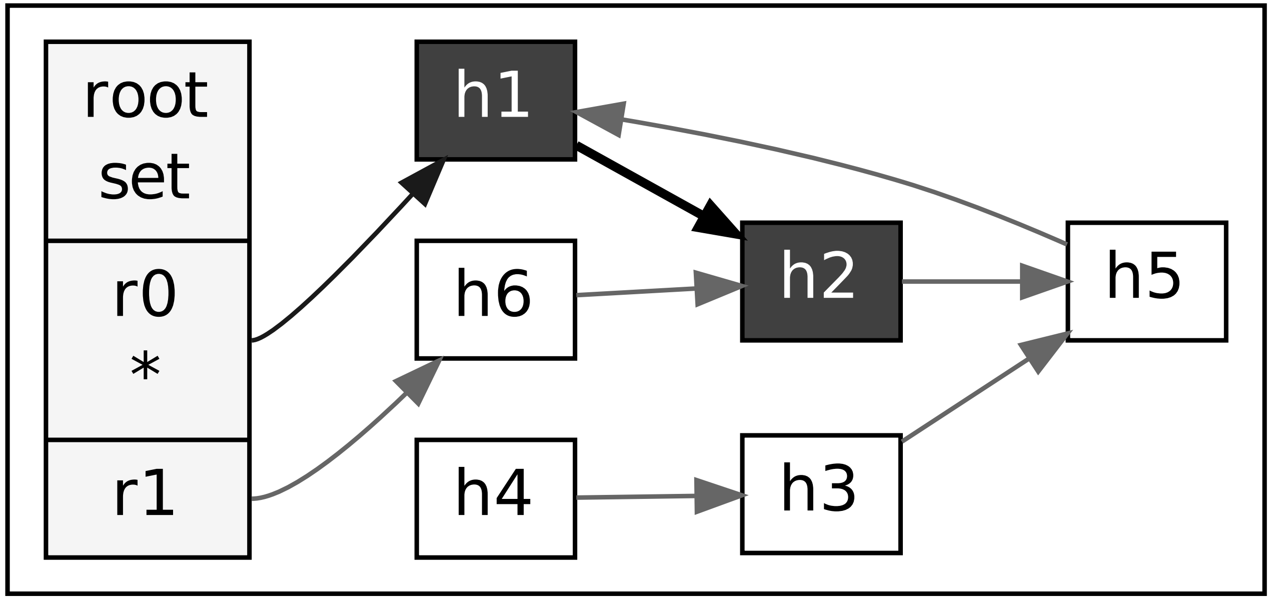
CLASSIC ALGORITHMS

- Reference counting
- Semi-space copy
- Mark & sweep



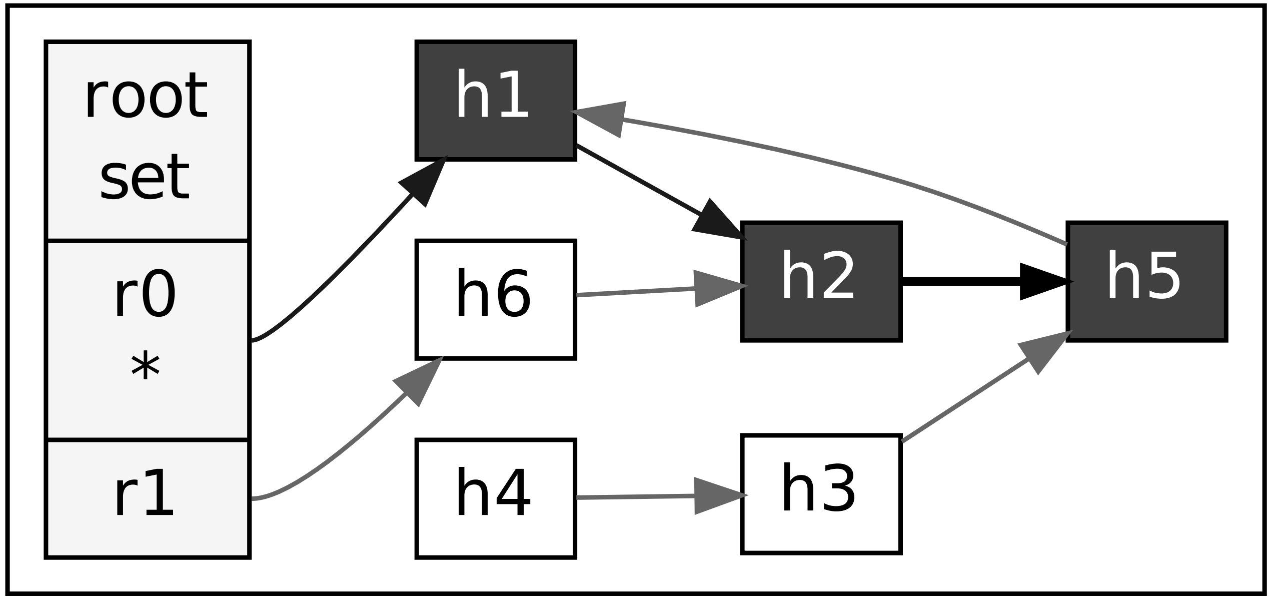
CLASSIC ALGORITHMS

- Reference counting
- Semi-space copy
- Mark & sweep



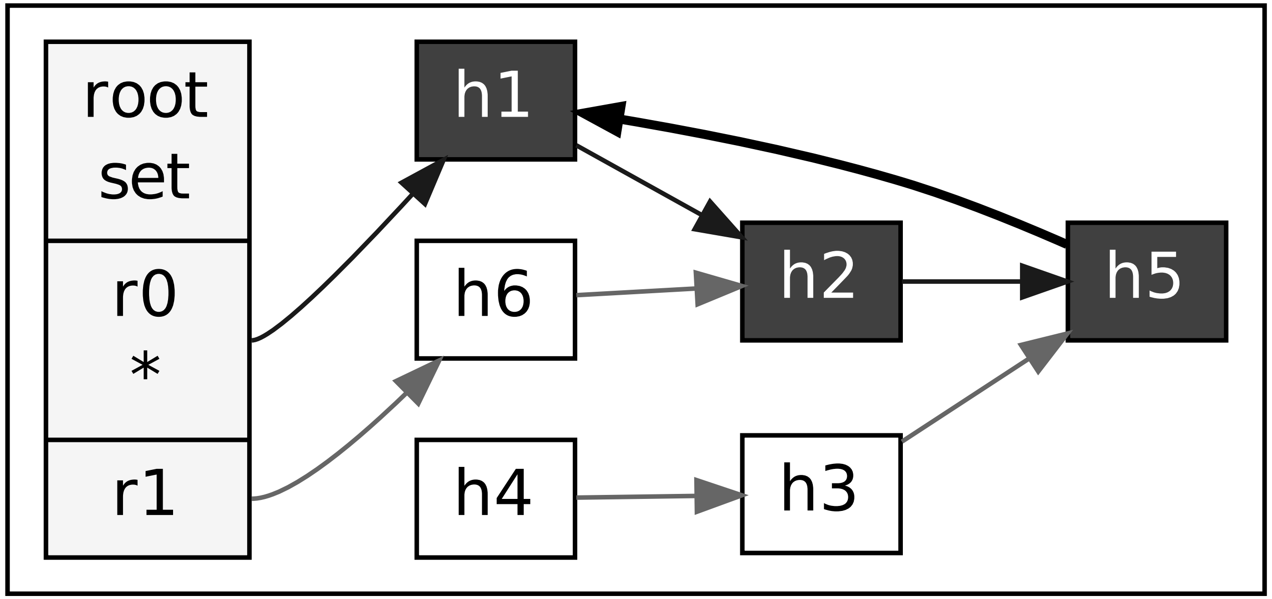
CLASSIC ALGORITHMS

- Reference counting
- Semi-space copy
- Mark & sweep



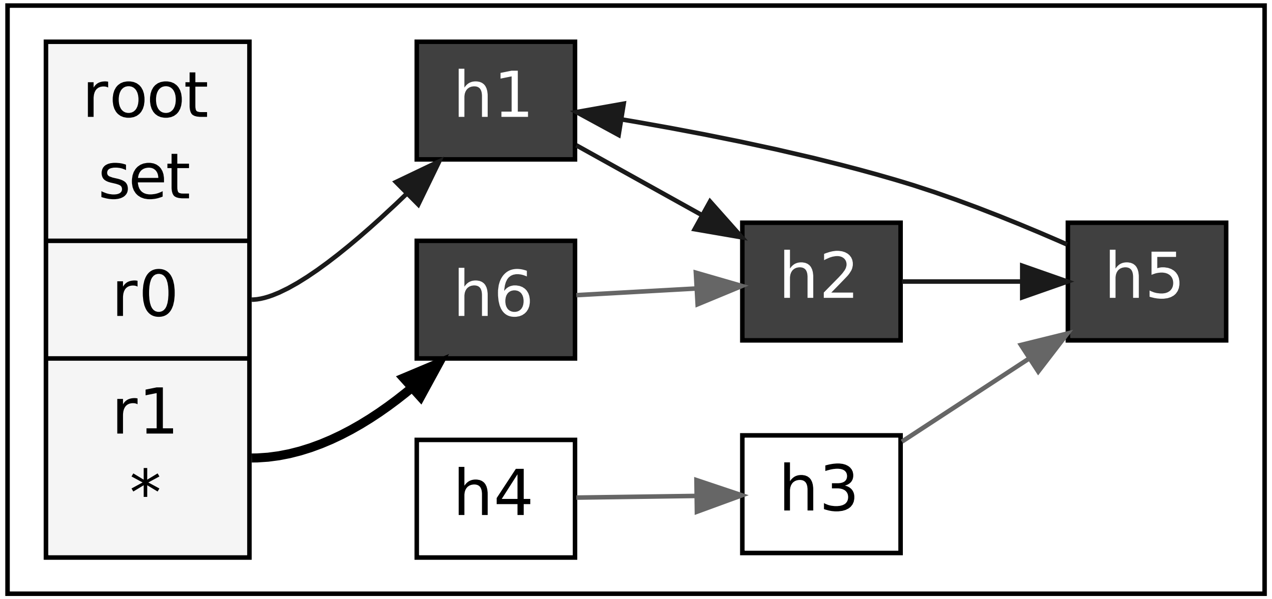
CLASSIC ALGORITHMS

- Reference counting
- Semi-space copy
- Mark & sweep



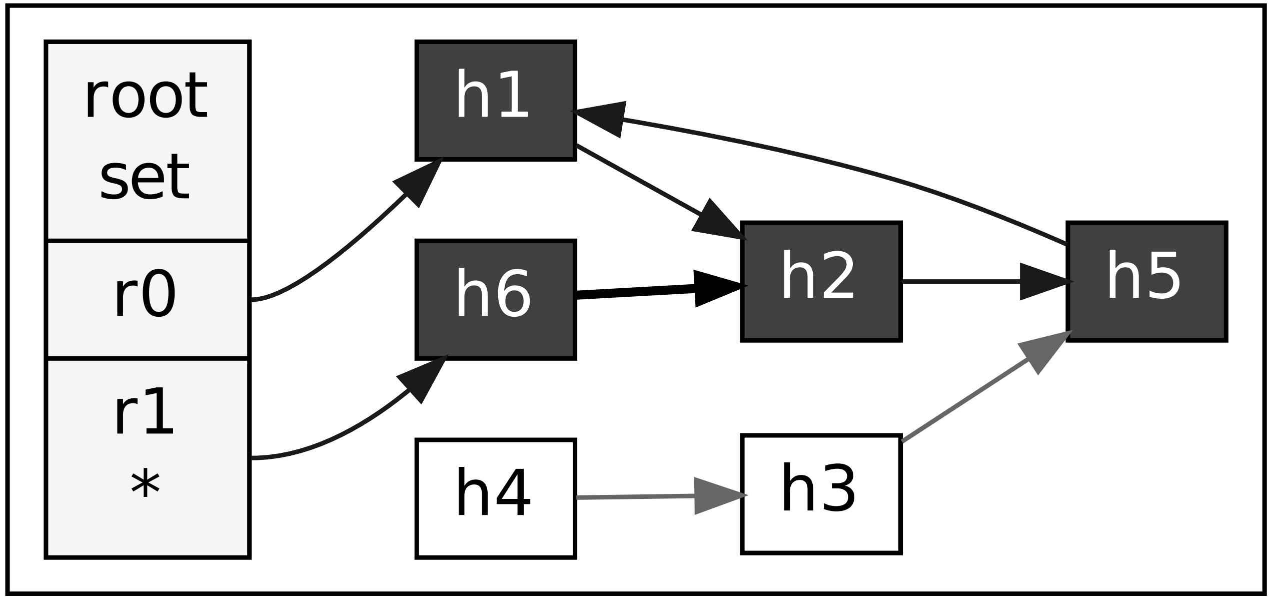
CLASSIC ALGORITHMS

- Reference counting
- Semi-space copy
- Mark & sweep



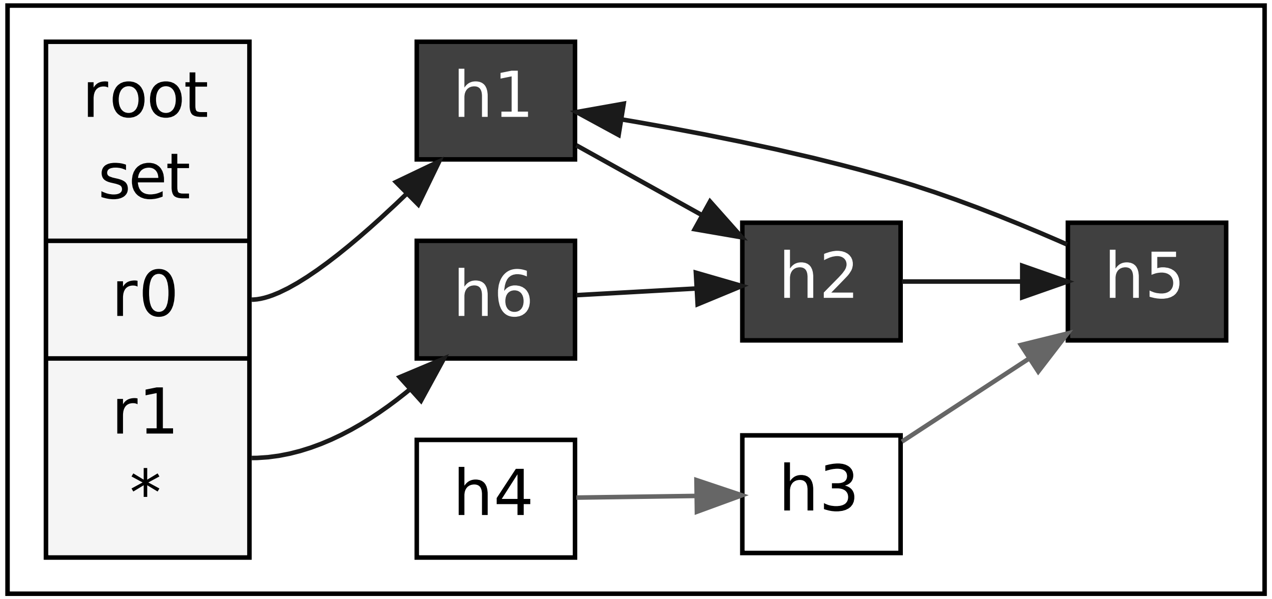
CLASSIC ALGORITHMS

- Reference counting
- Semi-space copy
- Mark & sweep



CLASSIC ALGORITHMS

- Reference counting
- Semi-space copy
- Mark & sweep



STATE OF THE ART

- 50+ years of research & development (3000+ papers)

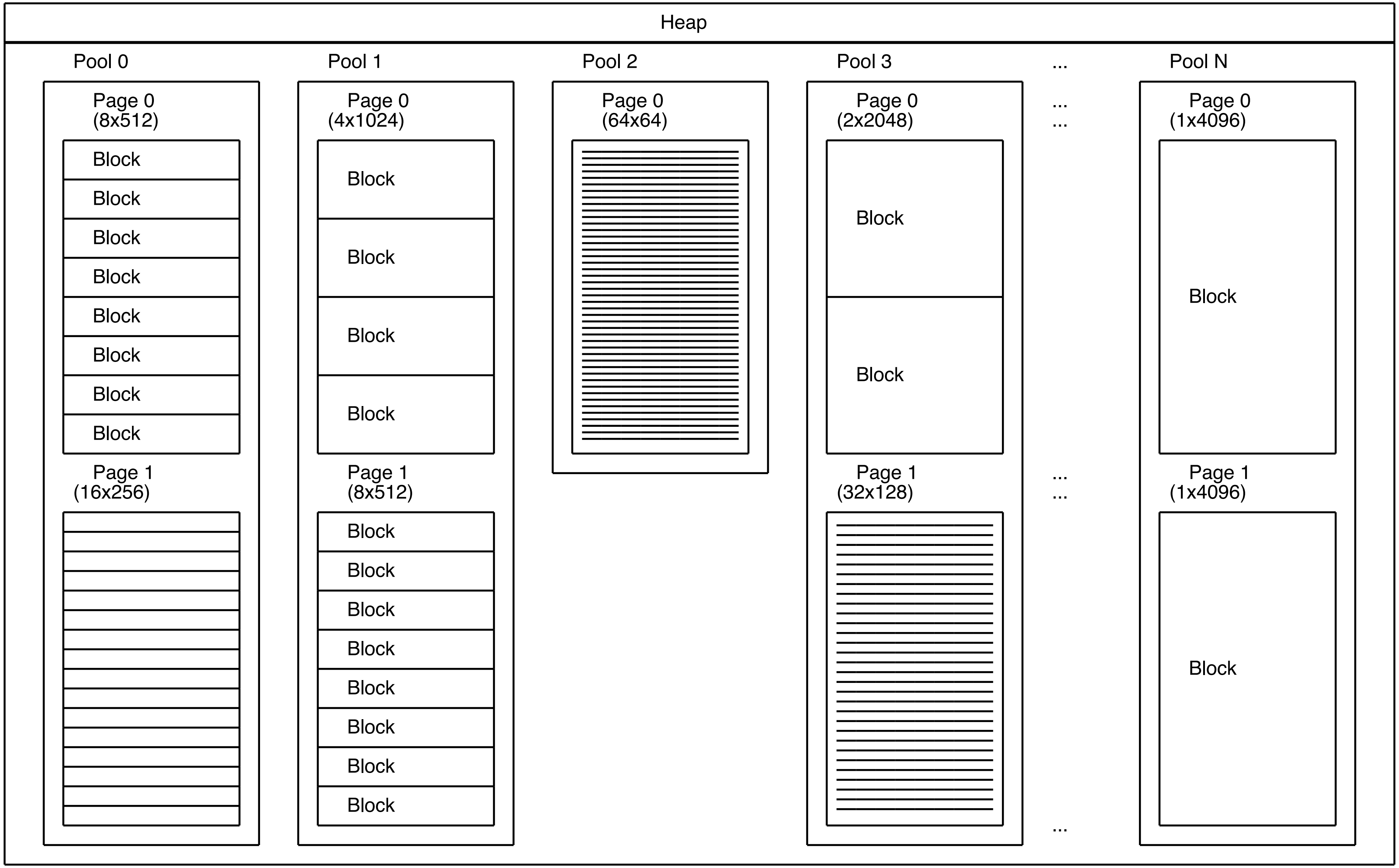
- Goal
  - ↓ Execution time
  - ↓ Number of collections
  - ↓ Collection time
  - ↓ **Pause time (maximum)**

- Techniques
  - -Partitions
  - **-Concurrency**
  - -Type information (precision/conservativeness)
  - -Static analysis

CLARIFICATION

- D1/TANGO ONLY! SORRY...
  - But all shouldn´t be too different from druntime
- UNIX ONLY
  - And tested only on Linux

HEAP STRUCTURE



HEAP > POOLS > PAGES > BLOCKS + FREE LISTS

BLOCKS

- Fixed Size
- Small Objects
  - -16 to 4096 bytes in powers of 2
  - -One page stores only one block size
  - -But blocks of the same size can live in discontinuous pages and different pools
- Big objects
  - -Size multiple of page size (4096, 8192,...)
  - -Each object lives in contiguous pages (and in the same pool)
- Flags
  - -One bit set per pool
  - -Several flags (bits) per block (mark, scan, free, etc.)

ALGORITHM

- **Mark & Sweep**

  Iterative mark phase (no recursion)

- **Conservative**

  With a pinch of precision (`NO_SCAN`)

- **Allocation-triggered**

  Only kicks in when an allocation request can't be fulfilled

- **Stop-the-world**

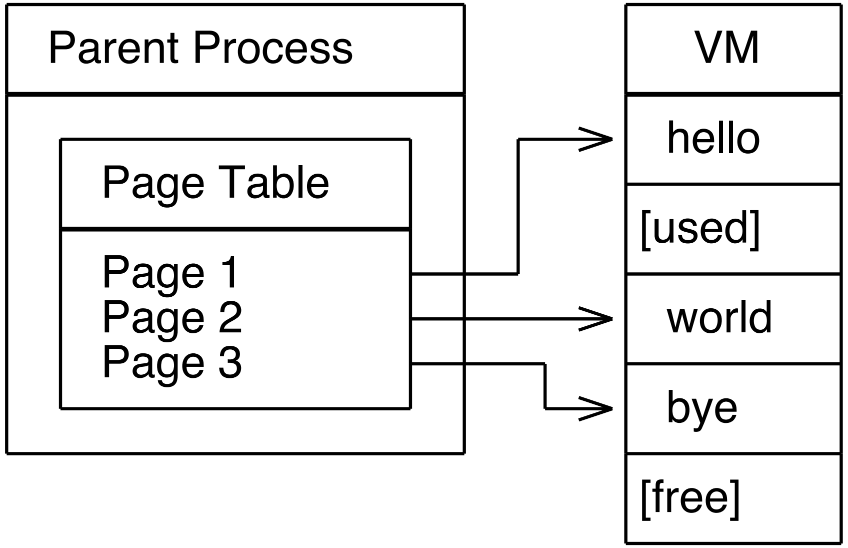
  Only in the mark phase (in theory)

- **Global lock**

  Too prone to extend the stop-the world time in practice

FORK (2)

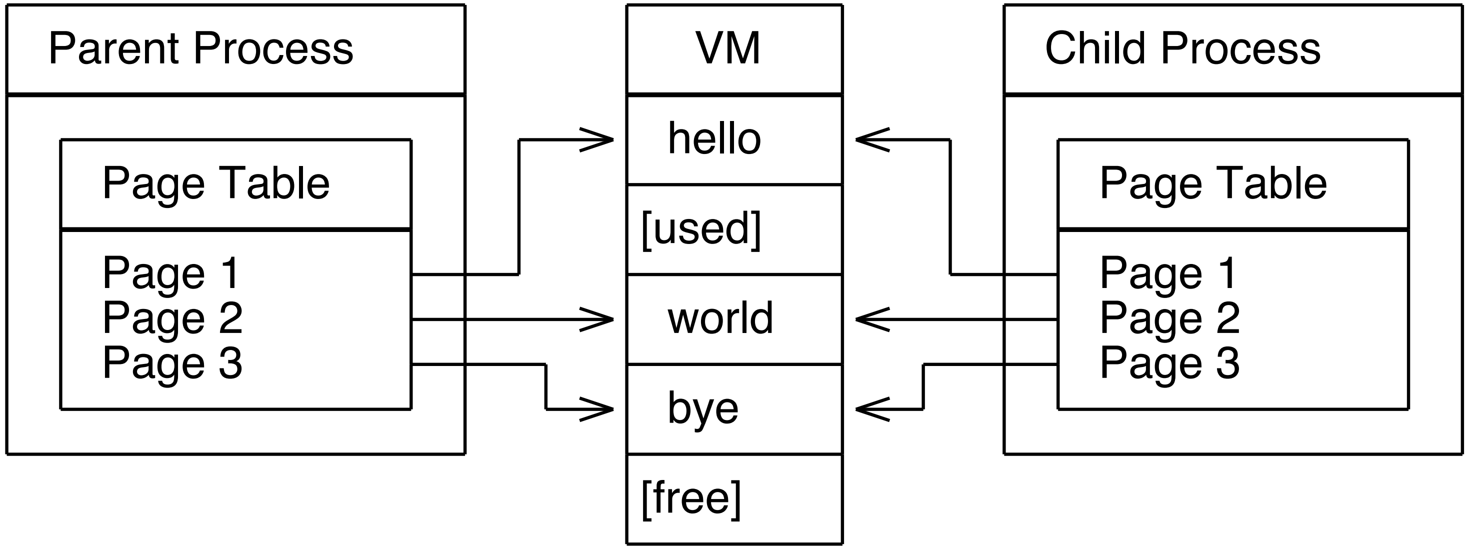
- Creates a new process (child) as a copy of the current one
- Child process is born with a **snapshot** of the parent´s memory
- Isolate modifications in parent and child´s memory
- Minimizes the actual copy of memory (COW)
- Starts with one thread only (the one called the `fork(2)`)
- Very efficient



Pre-Fork

FORK (2)

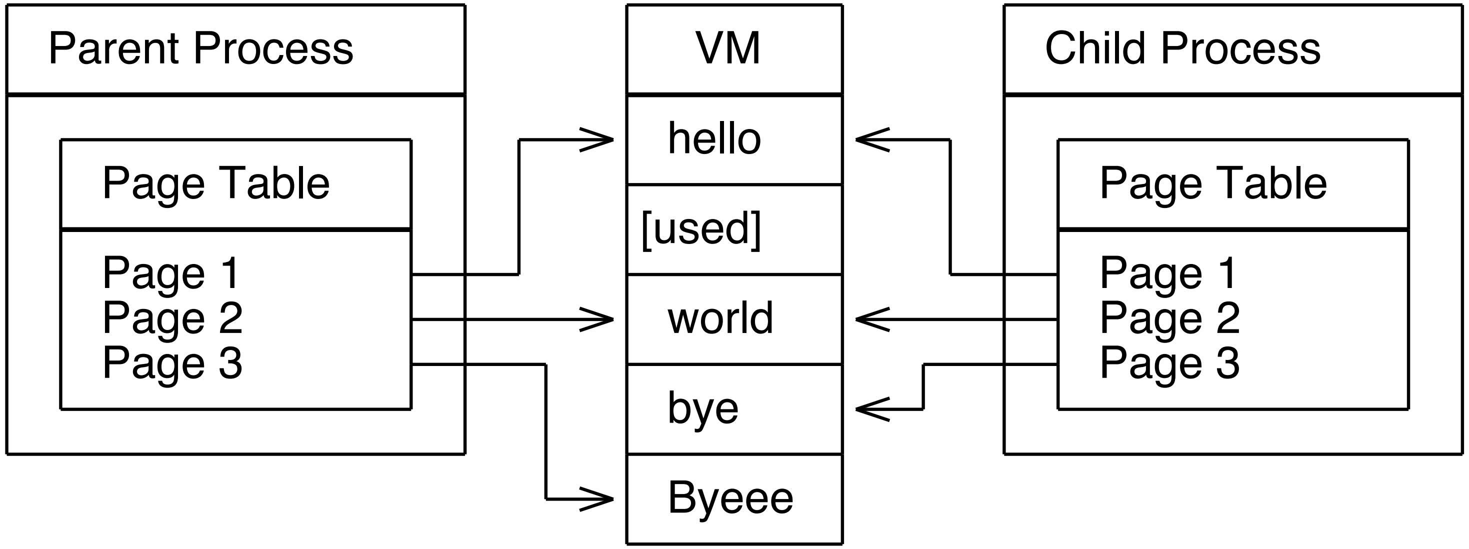
- Creates a new process (child) as a copy of the current one
- Child process is born with a **snapshot** of the parent´s memory
- Isolate modifications in parent and child´s memory
- Minimizes the actual copy of memory (COW)
- Starts with one thread only (the one called the `fork(2)`)
- Very efficient



Post-Fork

FORK (2)

- Creates a new process (child) as a copy of the current one
- Child process is born with a **snapshot** of the parent´s memory
- Isolate modifications in parent and child´s memory
- Minimizes the actual copy of memory (COW)
- Starts with one thread only (the one called the `fork(2)`)
- Very efficient



Parent write to Page 3

MAIN ALGORITHM

- Based on “Non-intrusive Cloning Garbage Collector with Stock Operating System Support” (Gustavo Rodriguez-Rivera and Vince Russo)
- Minimizes pause time through concurrent mark phase using `fork(2)`
- Parent process keeps running the program
- Child process runs the mark phase
- Results are communicated through shared memory
- Minimal synchronization: `fork(2) + waitpid(2)`

PROBLEMS

- Thread that triggered the collection is blocked until the end of the collection is completed (including the concurrent mark phase)

- Other threads might be potentially blocked too (global lock)

→ Real pause time ~= total collection time (not very concurrent in practice)

EAGER ALLOCATION

- Creates a new pool before starting the concurrent mark phase
  - -Resolves the memory allocation with the new pool
  - -Runs the mark phase really concurrently
- Let all program threads keep running in parallel to the mark phase
- Compromise
  - ↑ Memory usage
  - ↓ Real pause time

EARLY COLLECTION

- Triggers a preemptive collection before the memory is really exhausted
- Let all program threads keep running in parallel to the mark phase
  - Until the memory is exhausted
  - Doesn´t guarantee small pauses all the time
- Might run more collections than necessary
- Compromise
  - ↑ CPU usage (potentially)
  - ↓ Pause time (not guaranteed)

Combinable

- Eager allocation avoids blocking
- Early collection minimize potential high memory usage

OTHER IMPROVEMENTS

- Configurable at initialization-time
- Through environment variables `(D_GC_OPTS=fork=0 ./prog)`
- Old compile-time options converted to initialization-time options

```
mem_stomp

sentinel
```

- New options

```
pre_alloc
min_free
malloc_stats_file
collect_stats_file
fork
eager_alloc
early_collect
```

GENERALITIES

- Multiple runs (20-50)
  - -Minimize measurement errors
  - -Results expressed in terms of:
    - -Minimum
    - -Average
    - -Maximum
    - -Standard deviation
- Minimize variance between runs
  - `-cpufreq-set(1)`
  - `-nice(1)`
  - `-ionice(1)`
- 4 cores

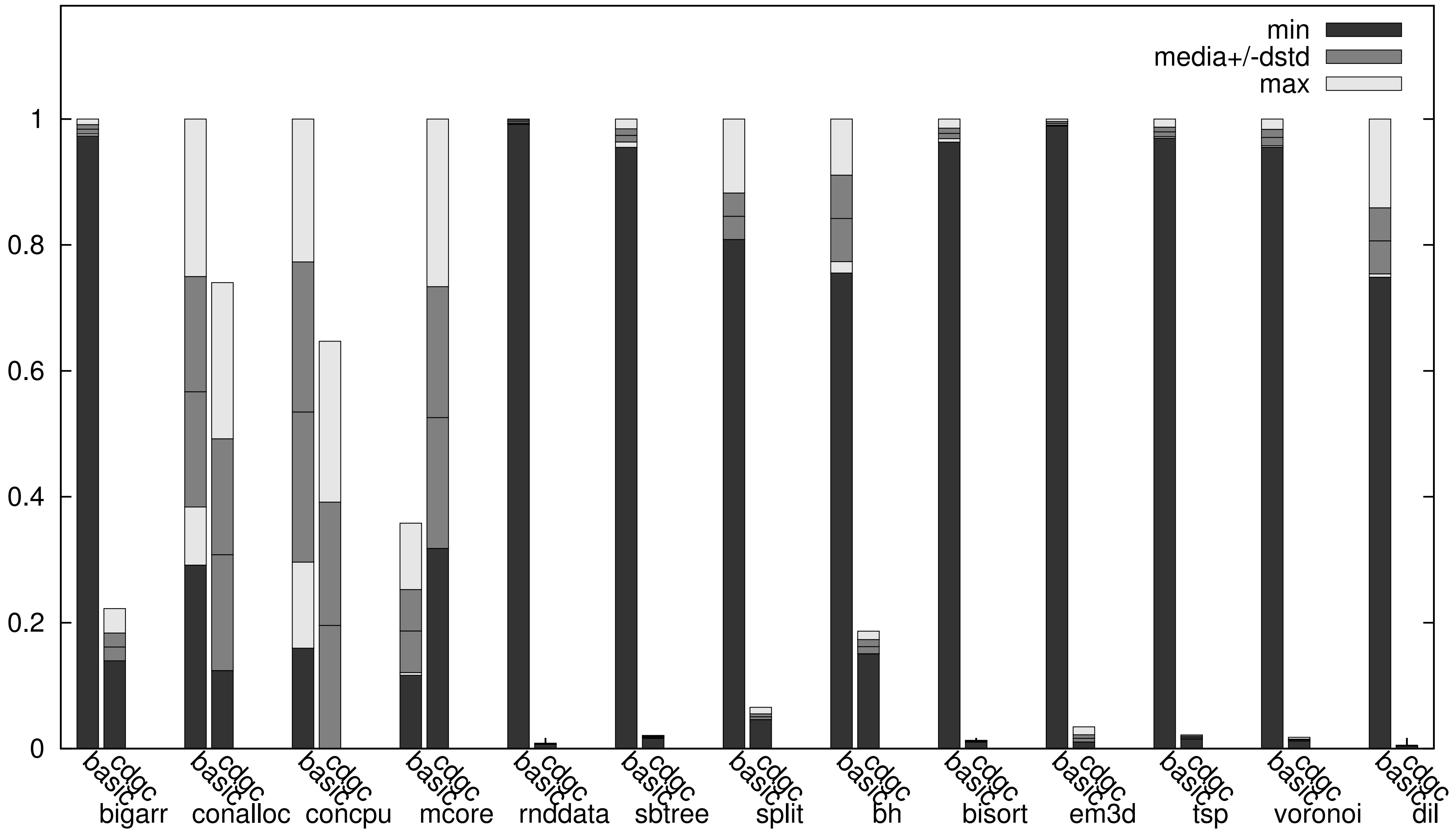
TESTBED

- Trivial programs (7)
  - -Stress particular aspects
  - -Don't perform a useful task
  - -Pathological cases
- Small programs – Olden Benchmark (5)
  - -Relatively small (400–1000 SLOC)
  - -Perform an useful task
  - -Manipulate lots of lists and tree structures, allocating a lot
  - -Not exactly fair to GC benchmarking
- Real program – Dil (1)
  - -D compiler written in D
  - -Fairly big and complex (32K+ SLOC, 86 modules, 300+ classes)
  - -Written without GC (limitations or advantages) in mind
  - -Strings, dynamic and associative array manipulation

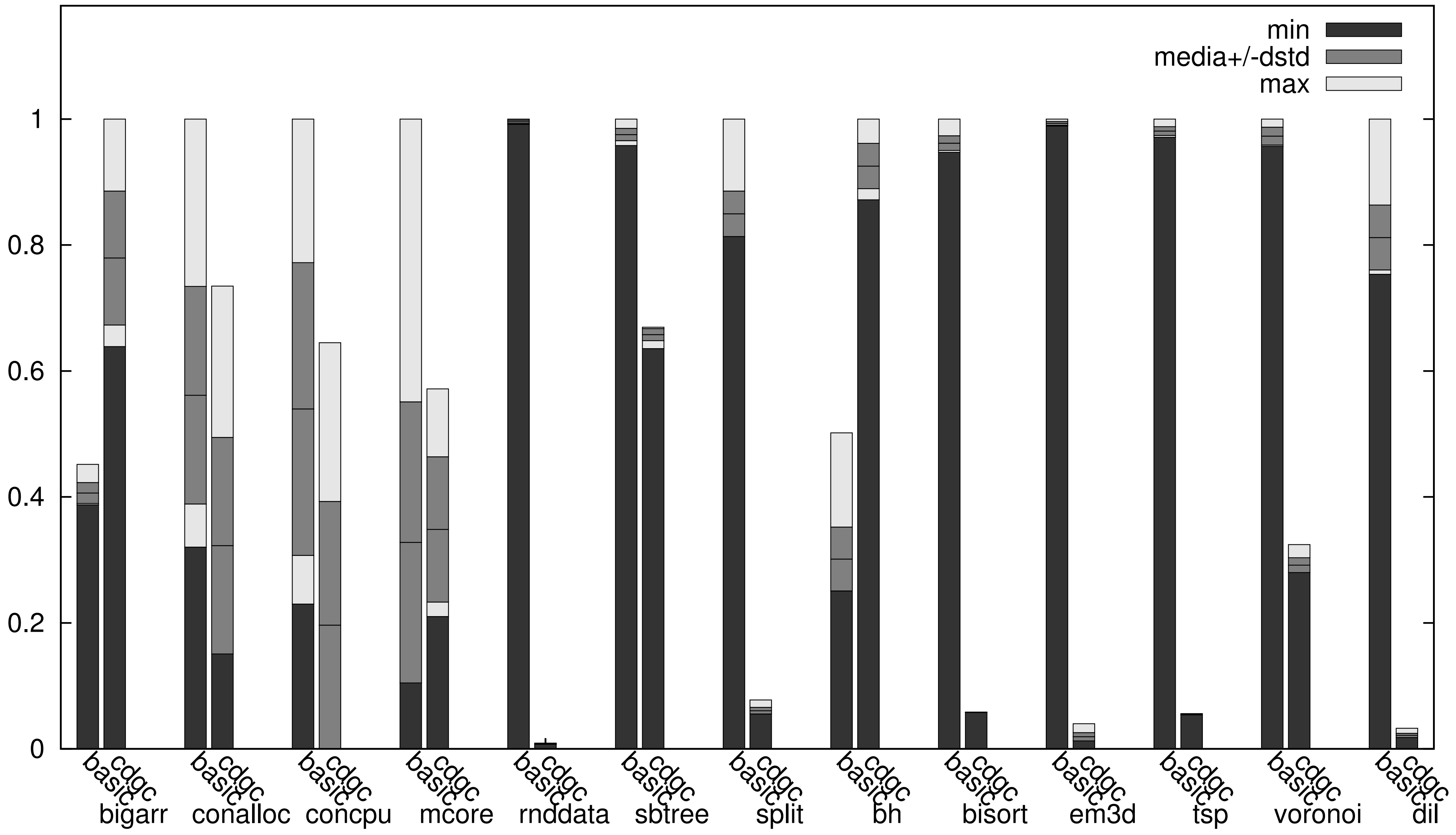
METRICS

- MAXIMUM STOP-THE-WORLD TIME
- MAXIMUM REAL PAUSE TIME
- PEAK MEMORY USAGE
- TOTAL EXECUTION TIME

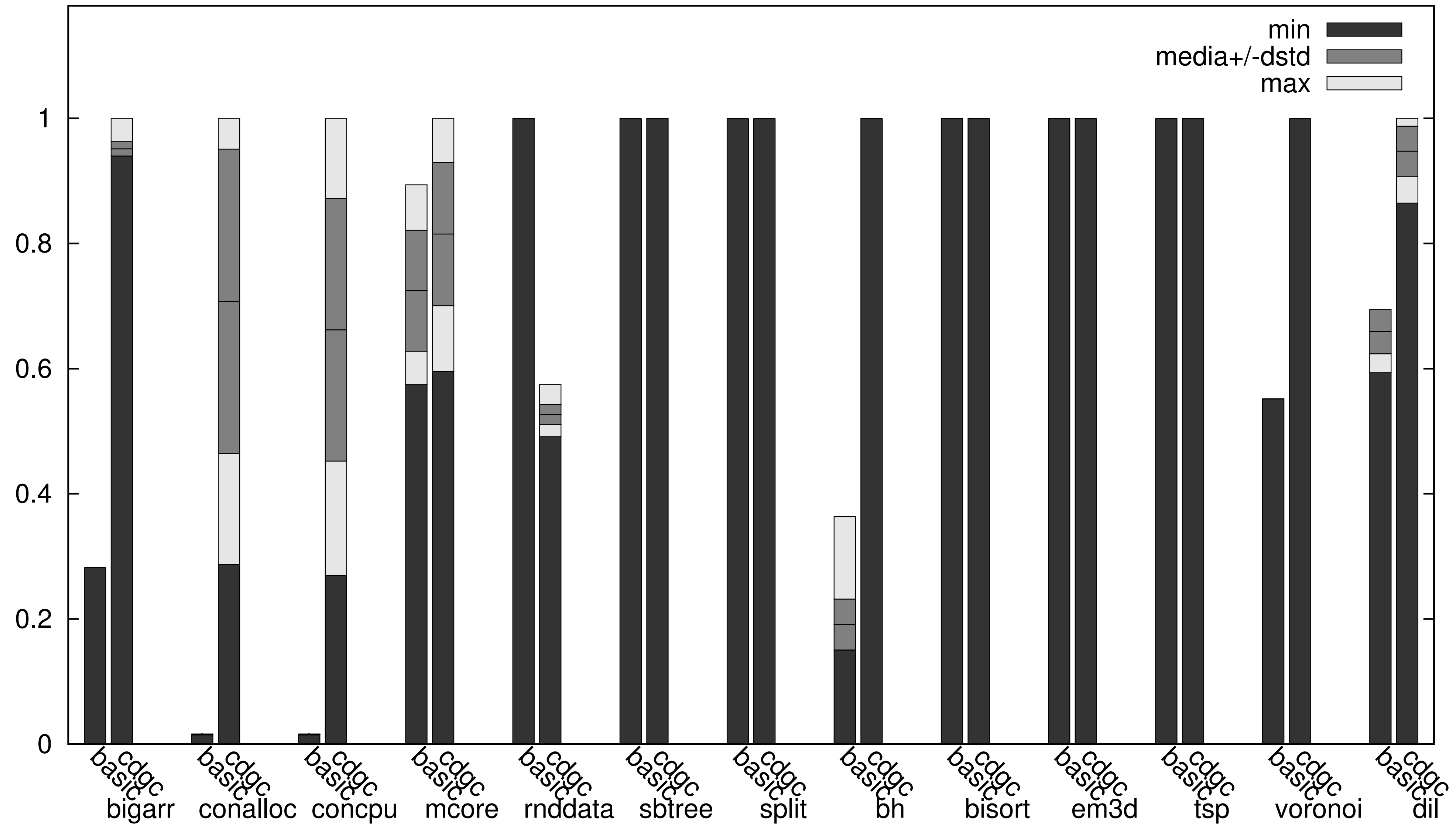
MAXIMUM STOP-THE-WORLD TIME



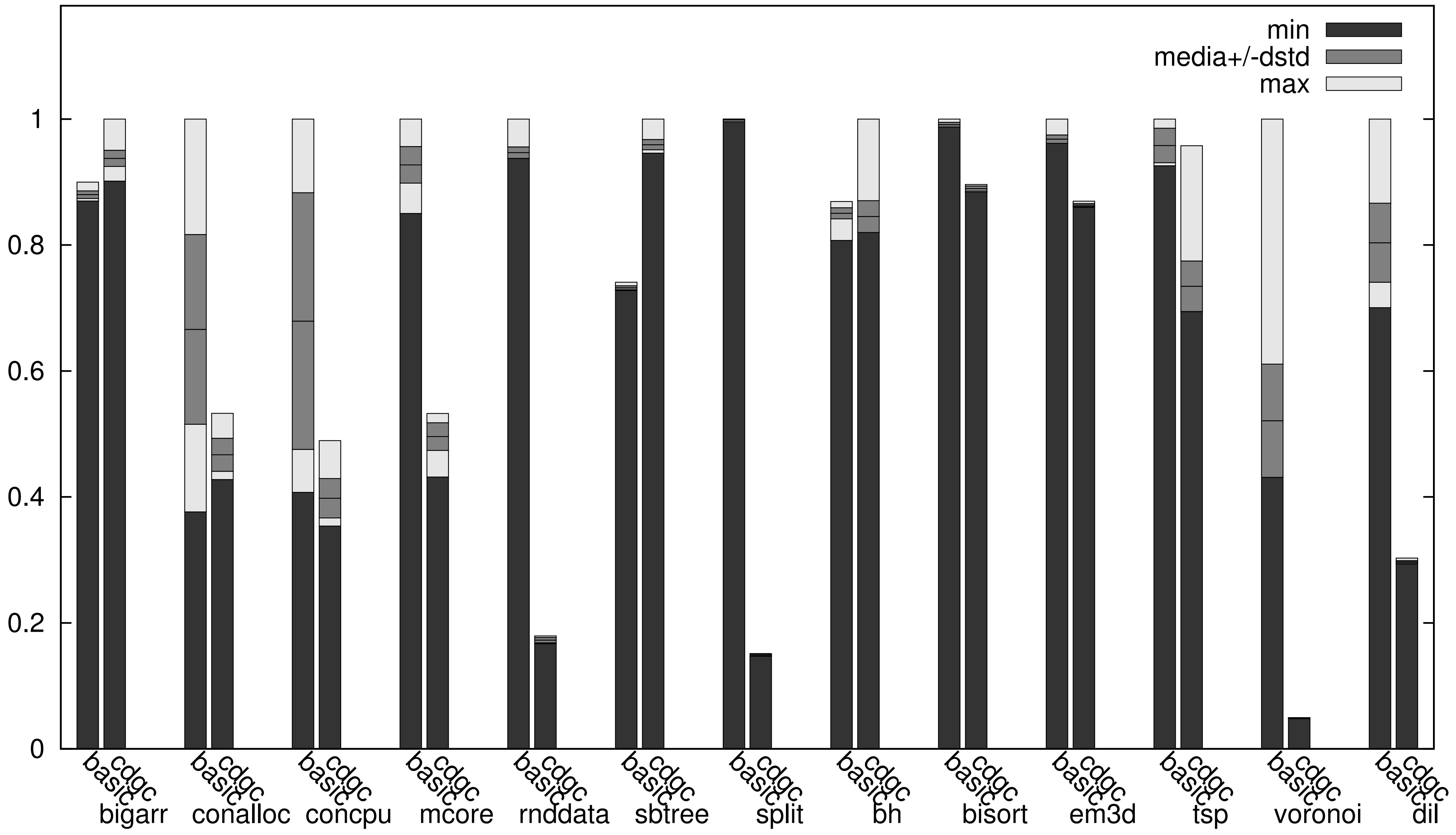
MAXIMUM REAL PAUSE TIME



PEAK MEMORY USAGE



TOTAL EXECUTION TIME



SUMMARY

- **STOP-THE-WORLD TIME 160 TIMES LOWER**
  DIL: 1.66s -> 0.01s
- **REAL PAUSE TIME 40 TIMES LOWER**
  DIL: 1.7s -> 0.045s
- **PEAK MEMORY USAGE COULD BE 50% HIGHER**
  DIL: 213MiB -> 307MiB
- **TOTAL EXECUTION TIME 3 TIMES LOWER**
  DIL: 55s -> 20s
- **TESTED IN REAL WORLD**
  USED IN SOCIOMANTIC FOR ALMOST 2 YEARS

PROBLEMS, LIMITATIONS AND OUTSTANDING ISSUES

- Memory usage explosion with eager allocation
  - Probably partly due to an (already fixed) bug
- Improve prediction for early collection
- Experiment with `clone(2)`
- Possible DEADLOCK when using glibc
  - internal glibc mutex + signals + stopped threads

FUTURE WORK

- Sweep phase
- Concurrency ! Global Lock
- Stop-the-world without using signals
- Moving collector

QUESTIONS

END

THANK YOU