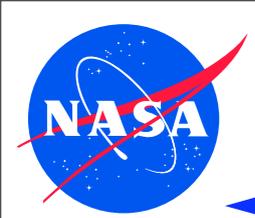
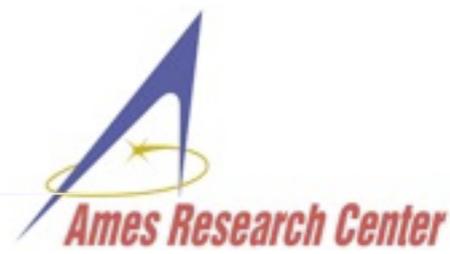


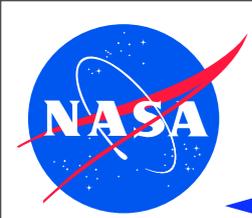
Java Pathfinder Version 6 Scalability



Goals



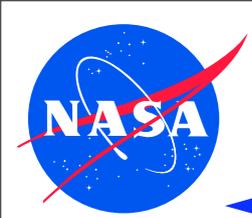
- ◆ develop new major JPF version to address scalability limiters identified in VVFCs milestone 06/30/2010
- ◆ two limiters identified in VVFCs milestone 06/30/2010:
 - (1) major: allocation time exponentially growing with heap size
 - (2) superfluous transitions/states caused by standard thread synchronization APIs
- ◆ primary goals are optimization efforts
- ◆ additional goals:
 - improve overall performance (garbage collection, state storage/matching, partial order reduction)
 - improve extensibility (can run contrary to optimization)
 - reduce complexity hotspots (e.g. partial order reduction reachability analysis)



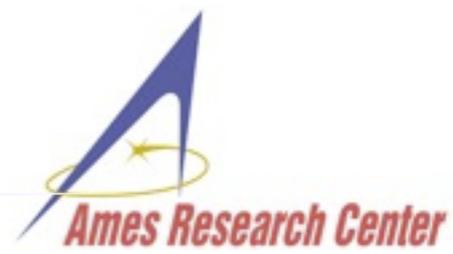
Deliverable



- ◆ JPF version 6 released 11/30/2010 on <http://babelfish.arc.nasa.gov/hg/jpf/jpf-core>
- ◆ major development effort (`hg history --stat -d">07/01/10"`):
 - total change statistics since VVFCs milestone 06/30:
 - ▶ 1439 files
 - ▶ 40197 added lines
 - ▶ 22906 removed lines
 - change statistics related to main goal (allocation time optimization):
 - ▶ 780 files
 - ▶ 19319 added lines
 - ▶ 12687 removed lines

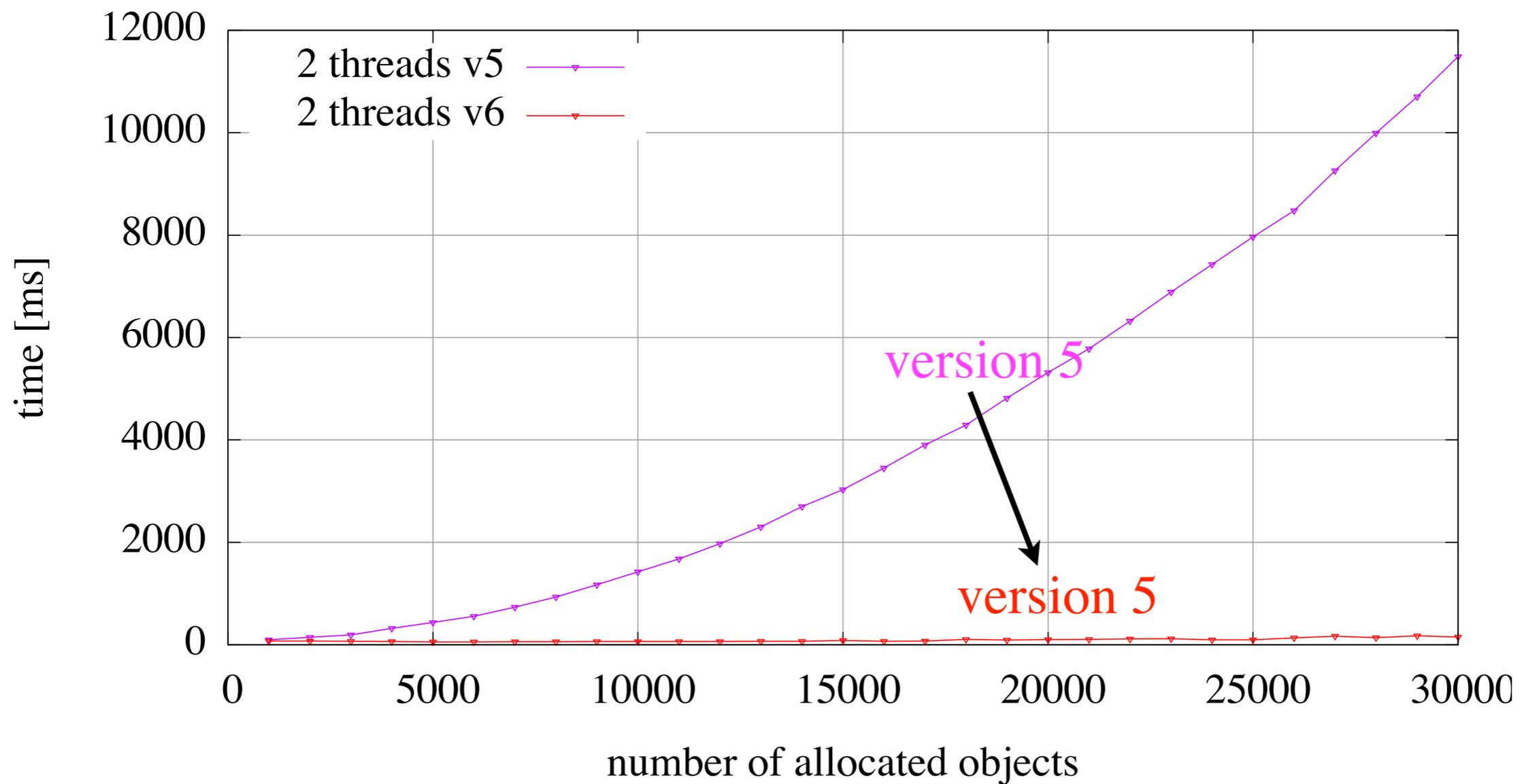


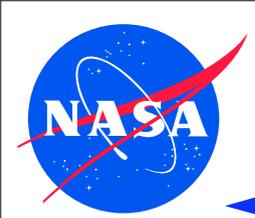
Results - Allocation Time (1)



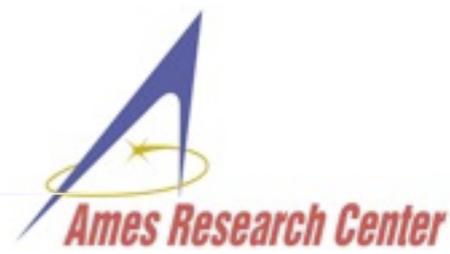
- ◆ major success: exponential allocation time problem eliminated
- ◆ v6 has linear allocation time

Absolute Run Time

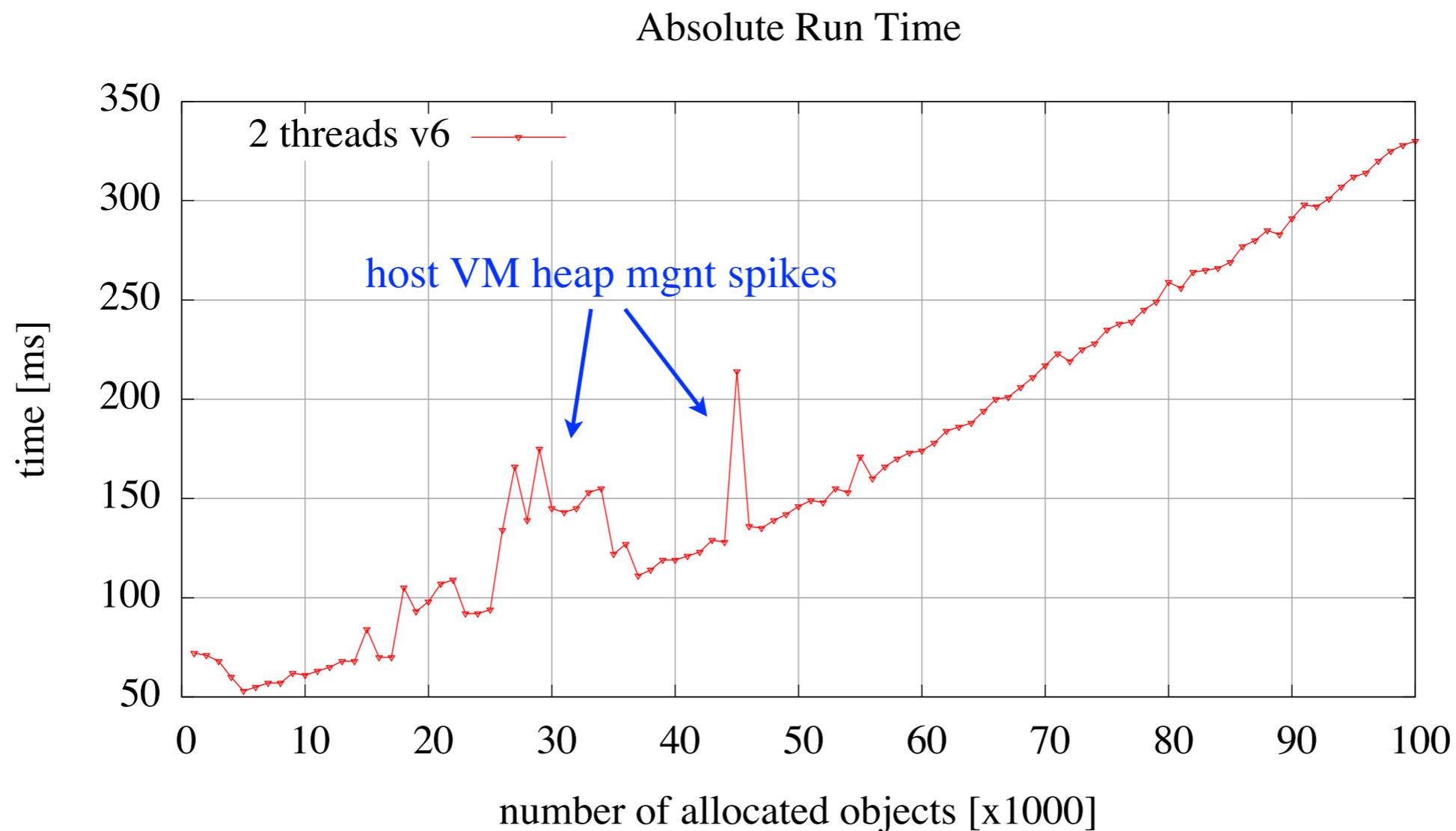


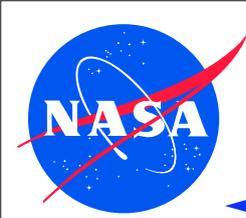


Results - Allocation Time (2)



- ◆ version 6 scales linearly up to $>10e5$ objects per thread
- ◆ absolute allocation time reduced to same order of magnitude as host VM heap management spikes (due to garbage collection / heap growth)
- ◆ \Rightarrow allocation bottleneck eliminated



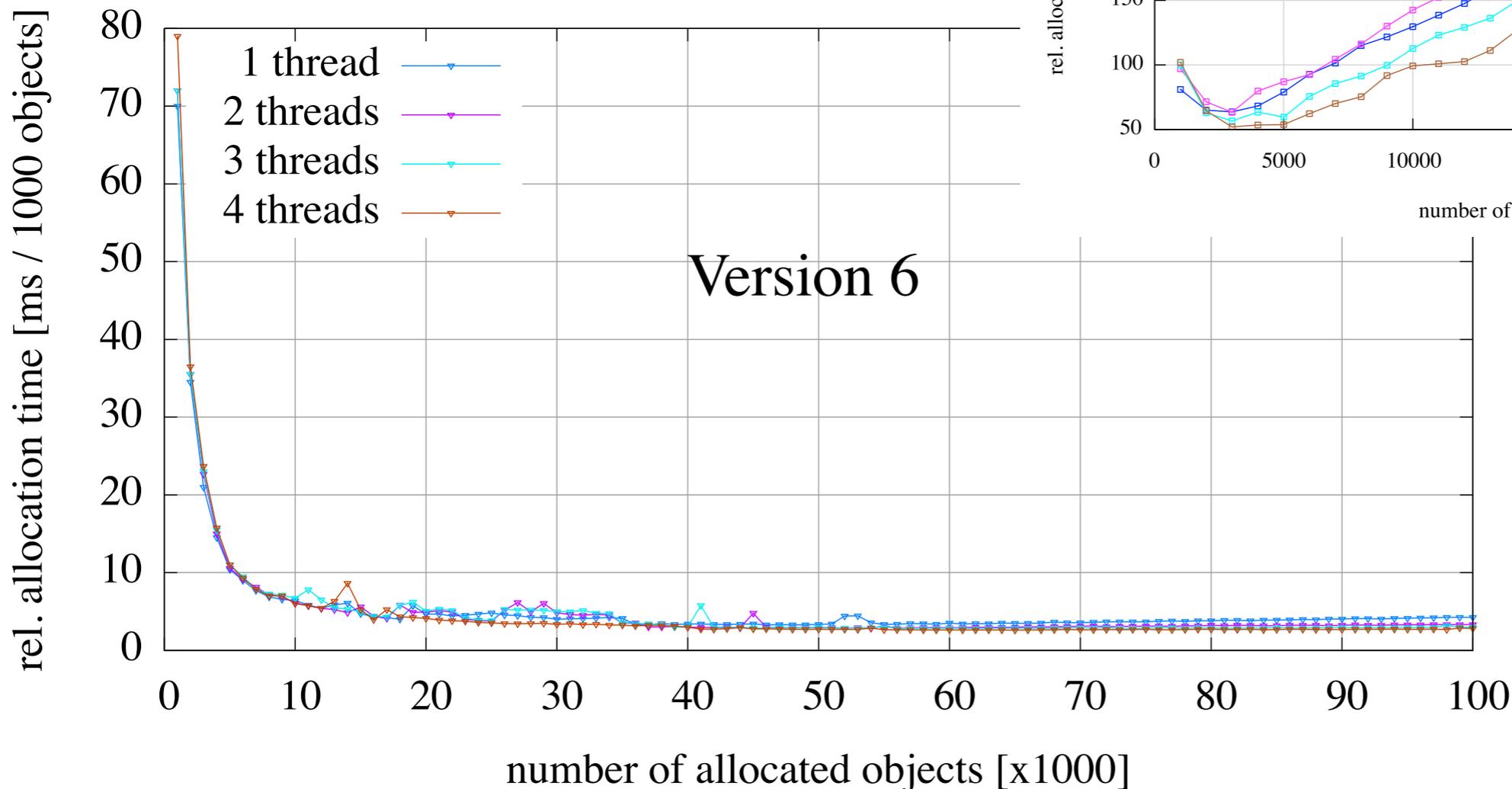


Results - Allocation Time (3)

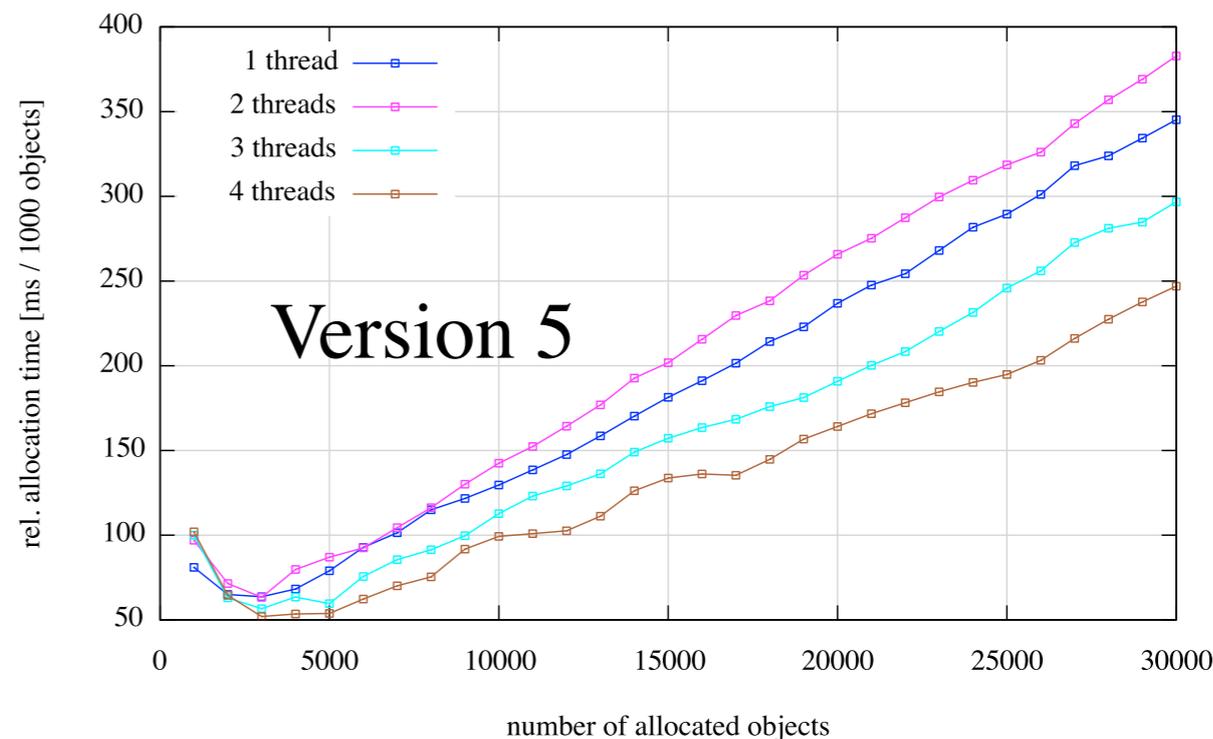


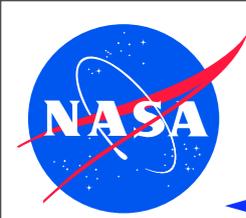
- ◆ relative allocation time in version 6 independent of number of allocating threads
- ◆ order of magnitude better for >10,000 objects
- ◆ nearly constant

Relative Allocation Time

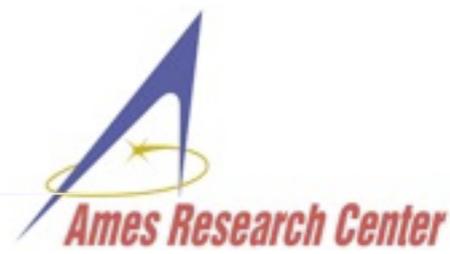


Relative Allocation Time

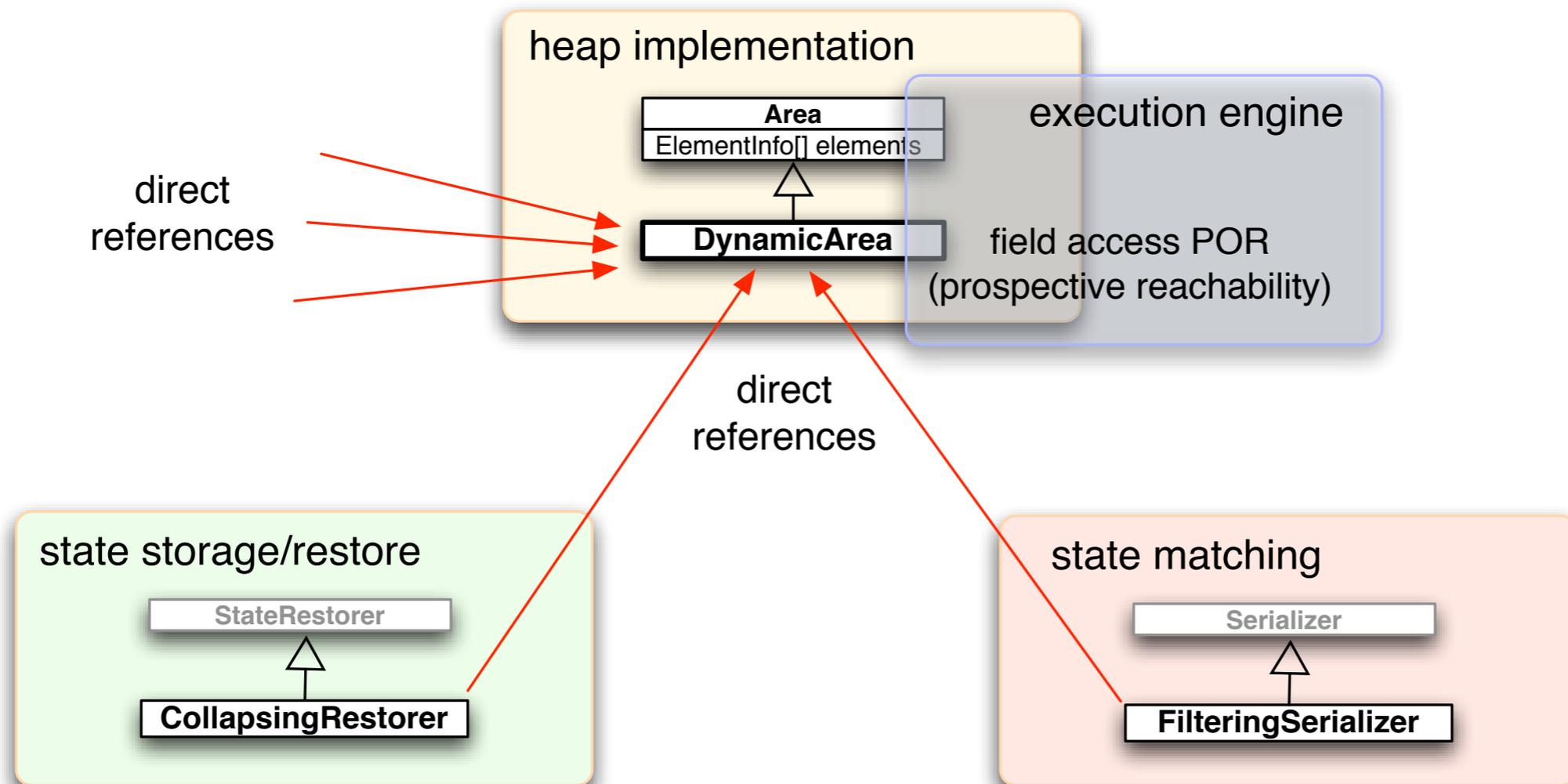


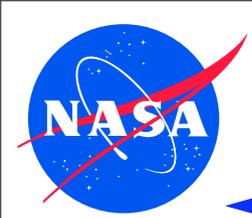


Challenges: Heap Replacement Insufficient

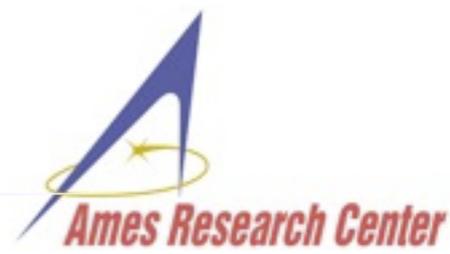


- ◆ allocation time was dependent on old heap implementation (required array of address space size)
- ◆ heap implementation was hardwired into state storage/restore and state matching components (both crucial for model checker)
- ◆ execution engine (partial order reduction) was piggybacking on DynamicArea garbage collector

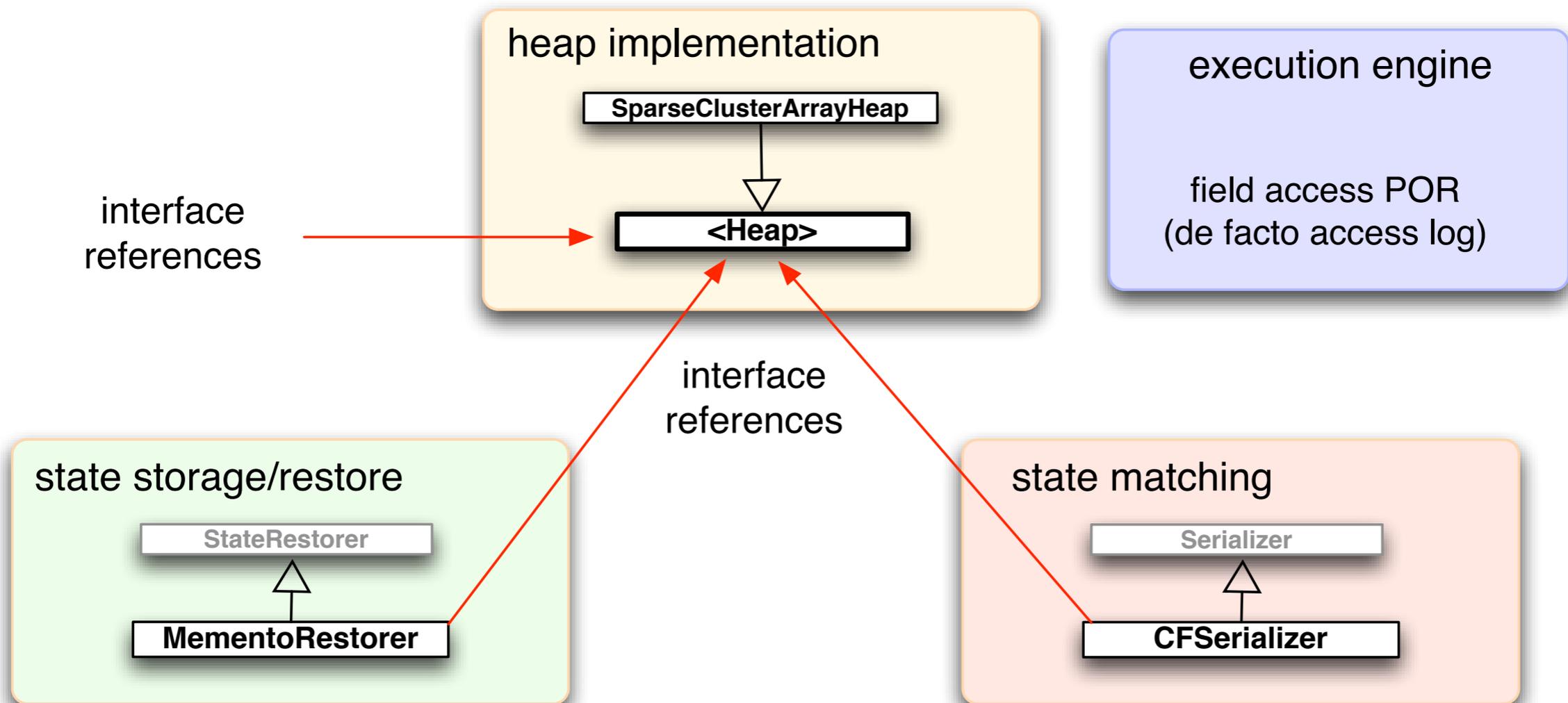


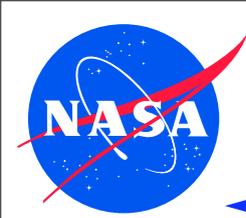


Solution: Redesign



- ◆ isolate heap implementation by means of abstract *Heap* interface
- ◆ design interface so that it does not require storage allocation for whole address range, and does not assume consecutive reference values
- ◆ implement per-thread clustered heap with efficient live object enumeration and free lists

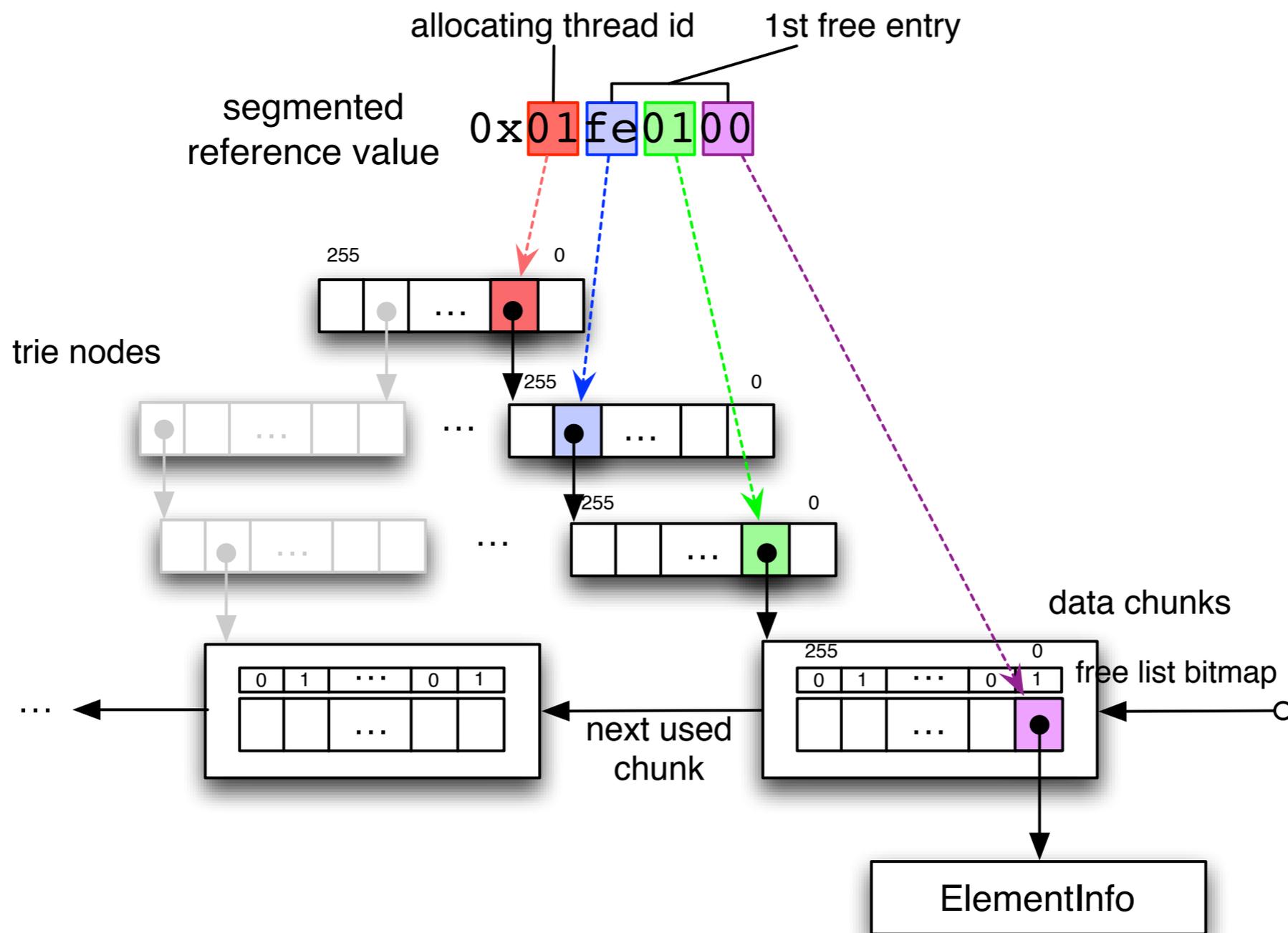


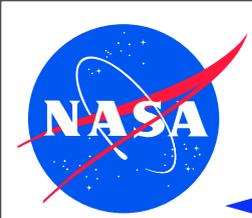


Solution: SparseClusterArrayHeap



- ◆ const time get & set operations (but thread and per-thread object limit)
- ◆ memory efficient for sparse address space (garbage collection holes)
- ◆ efficient enumeration of entries



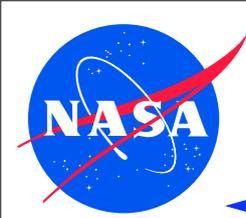


More Heap Redesign Benefits



- ◆ state storage, state matching, partial order reduction (POR) redesign ⇒ better extensibility + significant performance improvements for larger systems under test (less garbage collection cycles, less states)
- ◆ new design enables efficient implementation of property specific state matching (e.g. under-approximation for finding concurrency defects)
- ◆ non-trivial Java Swing example (jpf-awt/src/examples/RobotManager-threaded.jpf - complex NullPointerException in multithreaded graphical user interface program):

version	time	states	search depth
v5	15sec	8829	587
v6 _{generic}	10sec	6176	508
v6 _{concurrency}	1sec	360	89



State Optimization (1)



- second identified limiter (superfluous re-execution) was caused by standard Thread.join():

```

class Thread {
    ..
    public synchronized void join(..){
        while (isAlive()){
            wait();
        }
    }
}

```

scheduling point ① : lock acquisition

requires

scheduling point ② : signal wait

Thread t1

```

..
t2.join();
..

```

Thread t2

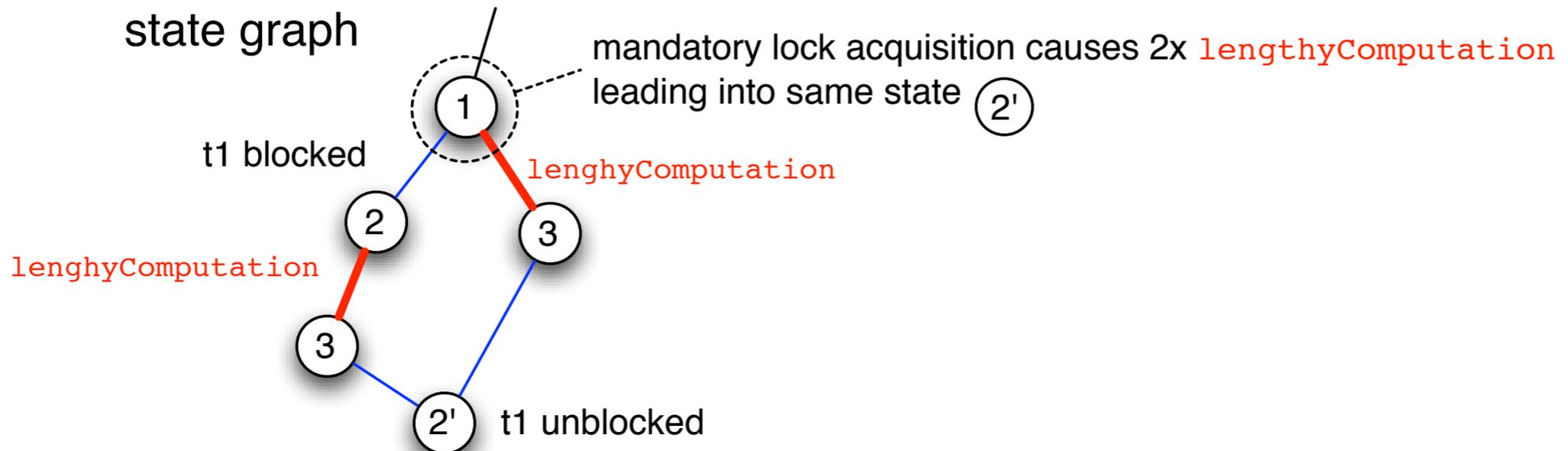
```

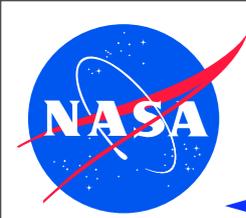
public void run(){
    lengthyComputation();
}

```

scheduling point ③ : thread termination

state graph





Results - State Optimization (2)



- ◆ lock acquisition only required for generic (application) signal waits
- ◆ specific signal waits (Thread.join) can be implemented inside virtual machine - no need for loop that requires lock protection
- ◆ depending on joined thread, savings can be significant:

```

===== system under test
application: gov/nasa/jpf/bench/AllocBench.java
arguments: 2 10000
..
starting 2 threads with 10000 objects each
===== statistics
elapsed time:      0:00:05
states:           new=9, visited=2, backtracked=10, end=2
heap:             gc=94, new=40331, free=40061
instructions:     603507

```

version 5

allocates twice as many objects as required due to superfluous lock in Thread.join()

```

===== system under test
application: gov/nasa/jpf/bench/AllocBench.java
arguments: 2 10000
..
starting 2 threads with 10000 objects each
===== statistics
elapsed time:      0:00:00
states:           new=5, visited=0, backtracked=4, end=1
heap:             new=20366, released=20036, max live=10348, gc-cycles=5
instructions:     303379

```

version 6