
AZUL
SYSTEMS


# Stack Based Allocation in the Azul JVM™

Dr. Cliff Click
cliffc@azulsystems.com

# Background



AZUL SYSTEMS

- The Azul JVM™ is based on Sun HotSpot
  - a State-of-the-Art Java VM
- Java is a GC'd language
- HotSpot uses a fast Generational GC
  - with “test and bump” allocation
- Test and bump allocation streams through memory
- Streaming memory is hard on caches
- Azul is looking at using *Stack Based Allocation*

# Cost of using a Generational GC



AZUL SYSTEMS

- Typical allocation is pointer compare & bump
- Deallocation is “free”
  - Requires scanning the live set
  - Live set is much smaller than what was allocated
- Allocation streams through young generation
  - Similar to streaming array access in Fortran
- Young generation >> L1 cache size
  - Implies caches are flushed repeatedly
- Cache miss cost is spread out
  - Hard to spot

# Cost of using a Generational GC



AZUL SYSTEMS

- Allocation streams through young generation
- Cache miss for each new object
  - Can cover with prefetch
- Read data is dead
  - Object must be initialized
  - Can prevent read with “cache-line-zero” ops
- Must evict for each new object
  - Evicted data is mostly dead
  - Occasional live object
- Result: wasted read AND wasted write

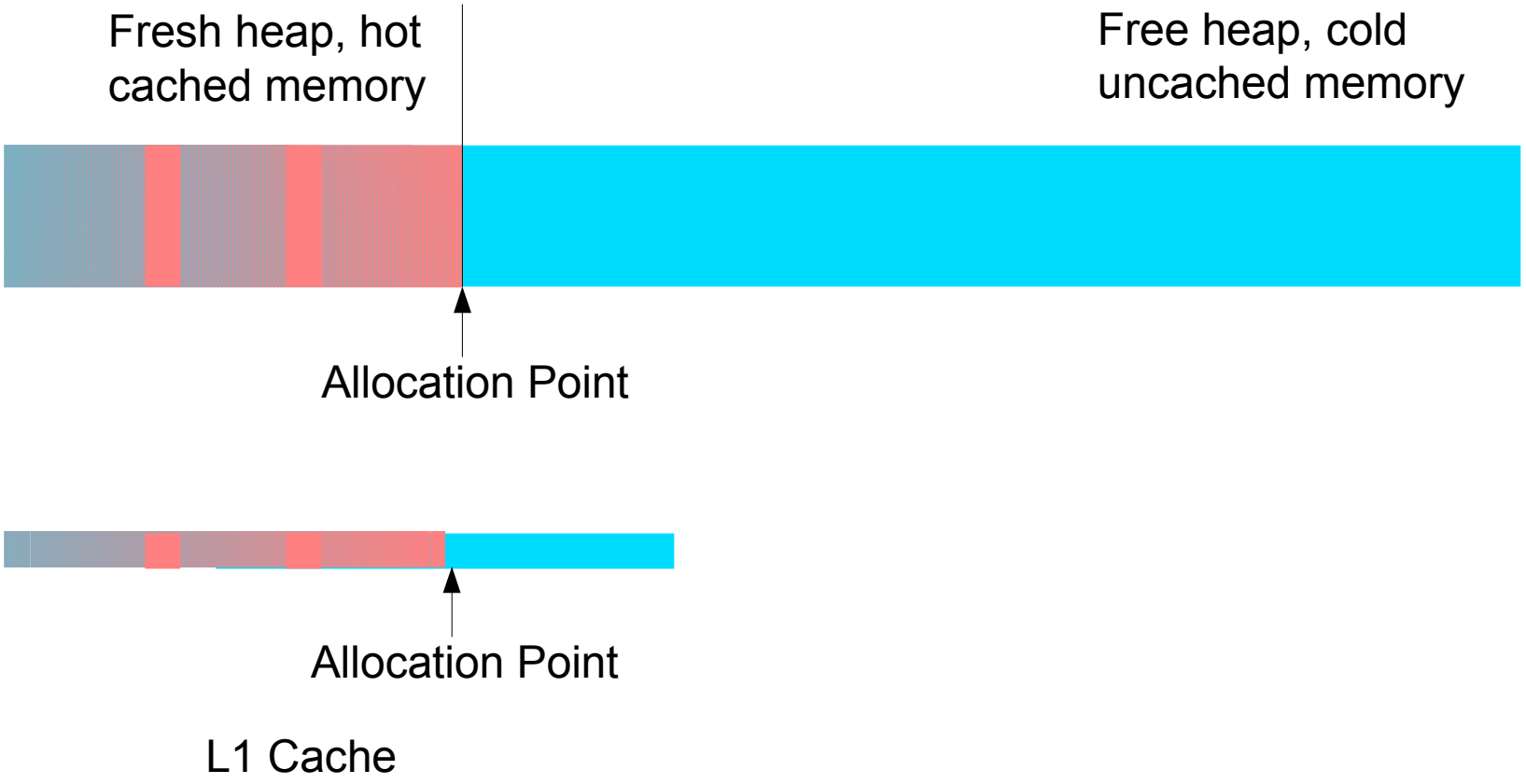
 

# Generational GC



AZUL SYSTEMS

## Memory usage, over time



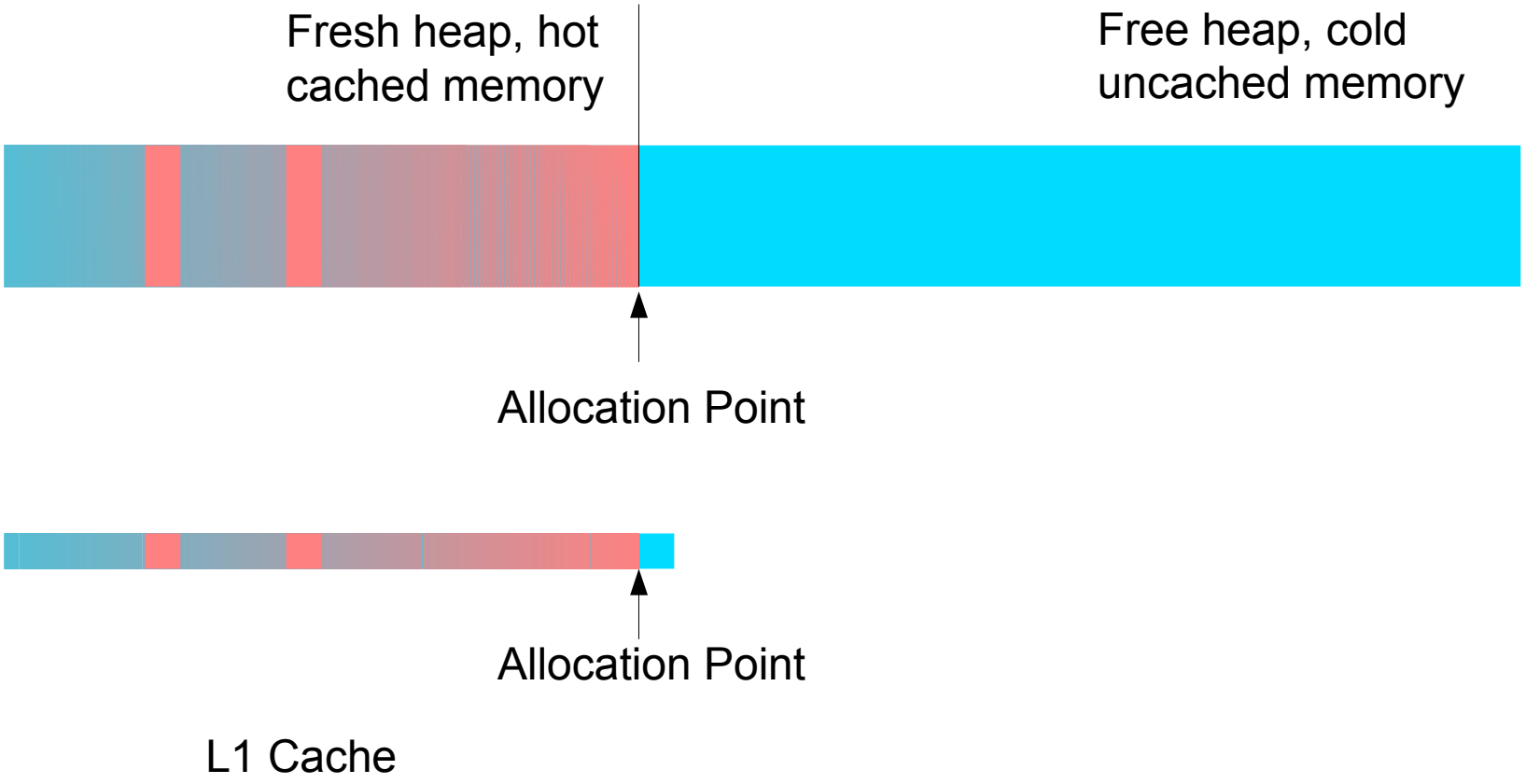

 

# Generational GC



AZUL SYSTEMS

## Memory usage, over time



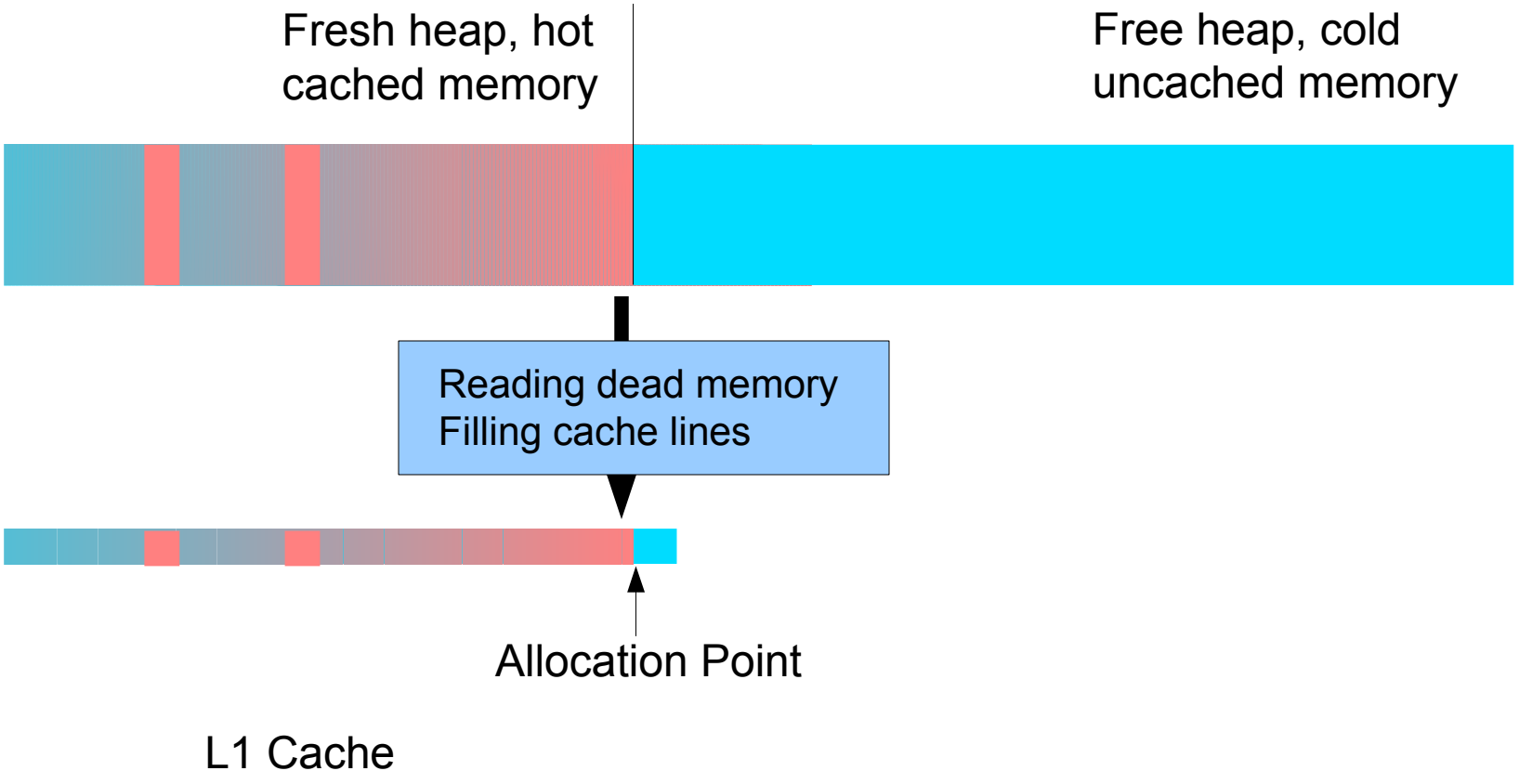

 

# Generational GC



AZUL SYSTEMS

## Memory usage, over time



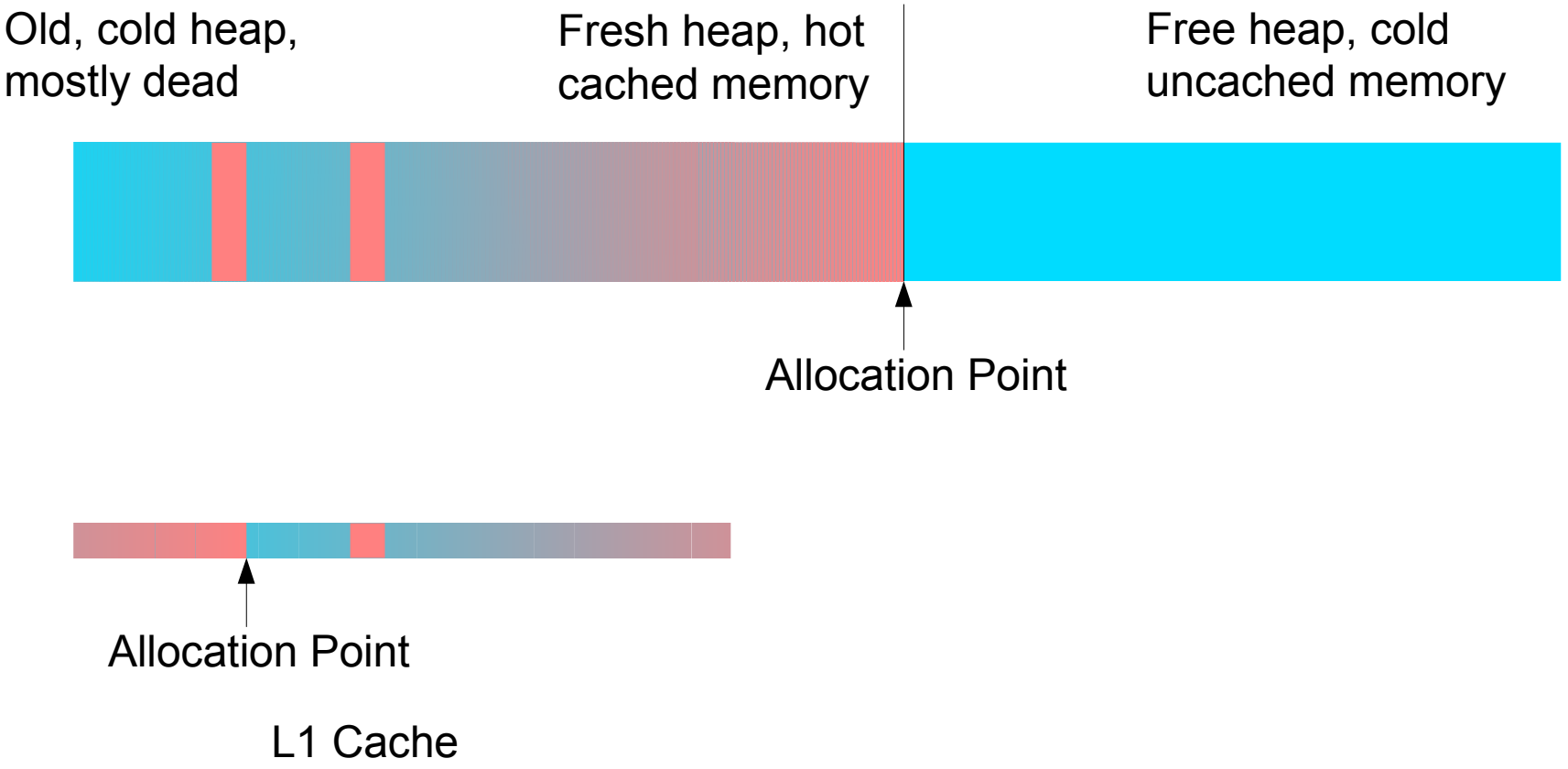

 

# Generational GC



AZUL SYSTEMS

## Memory usage, over time



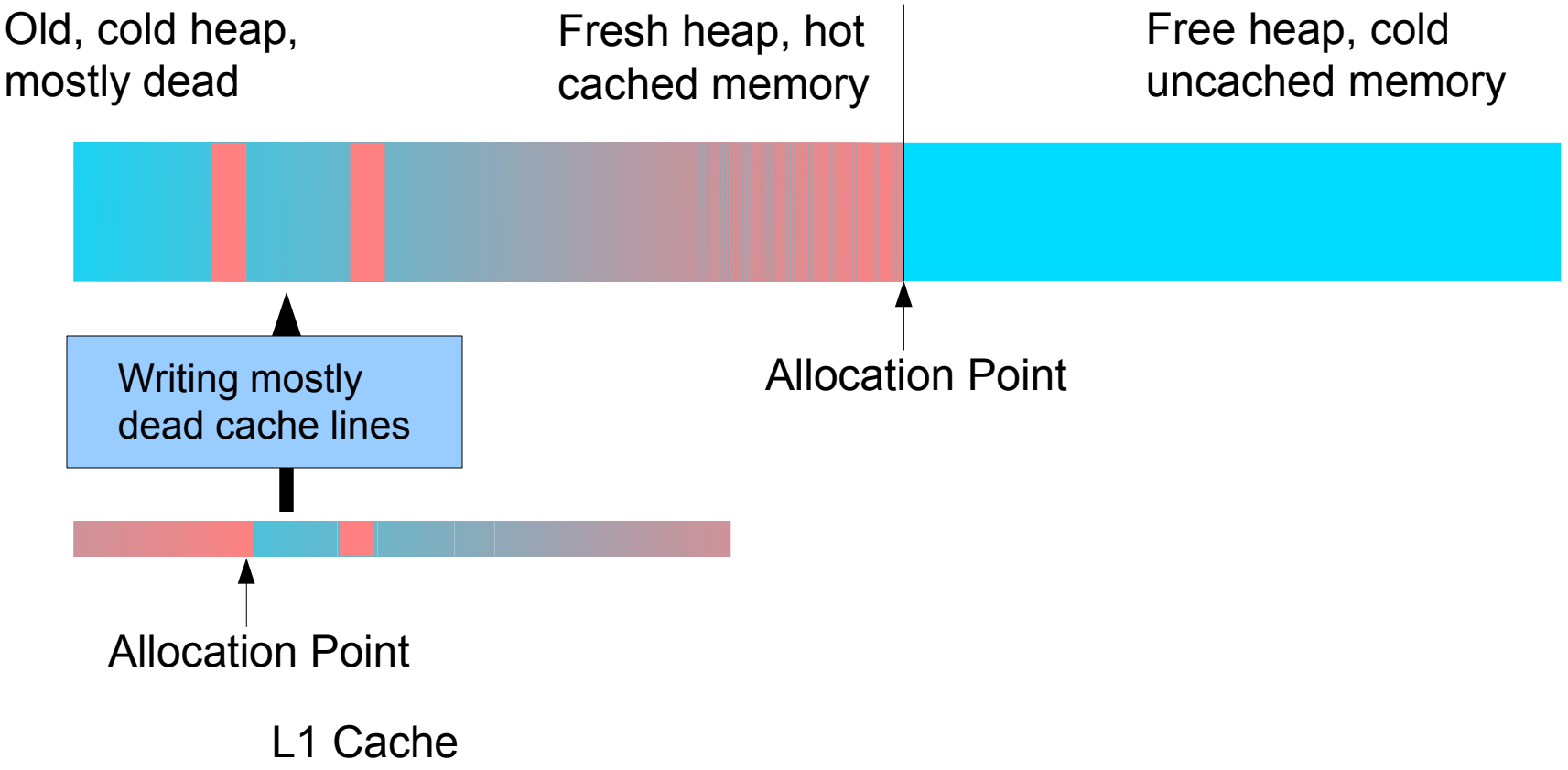

 

# Generational GC



AZUL SYSTEMS

## Memory usage, over time



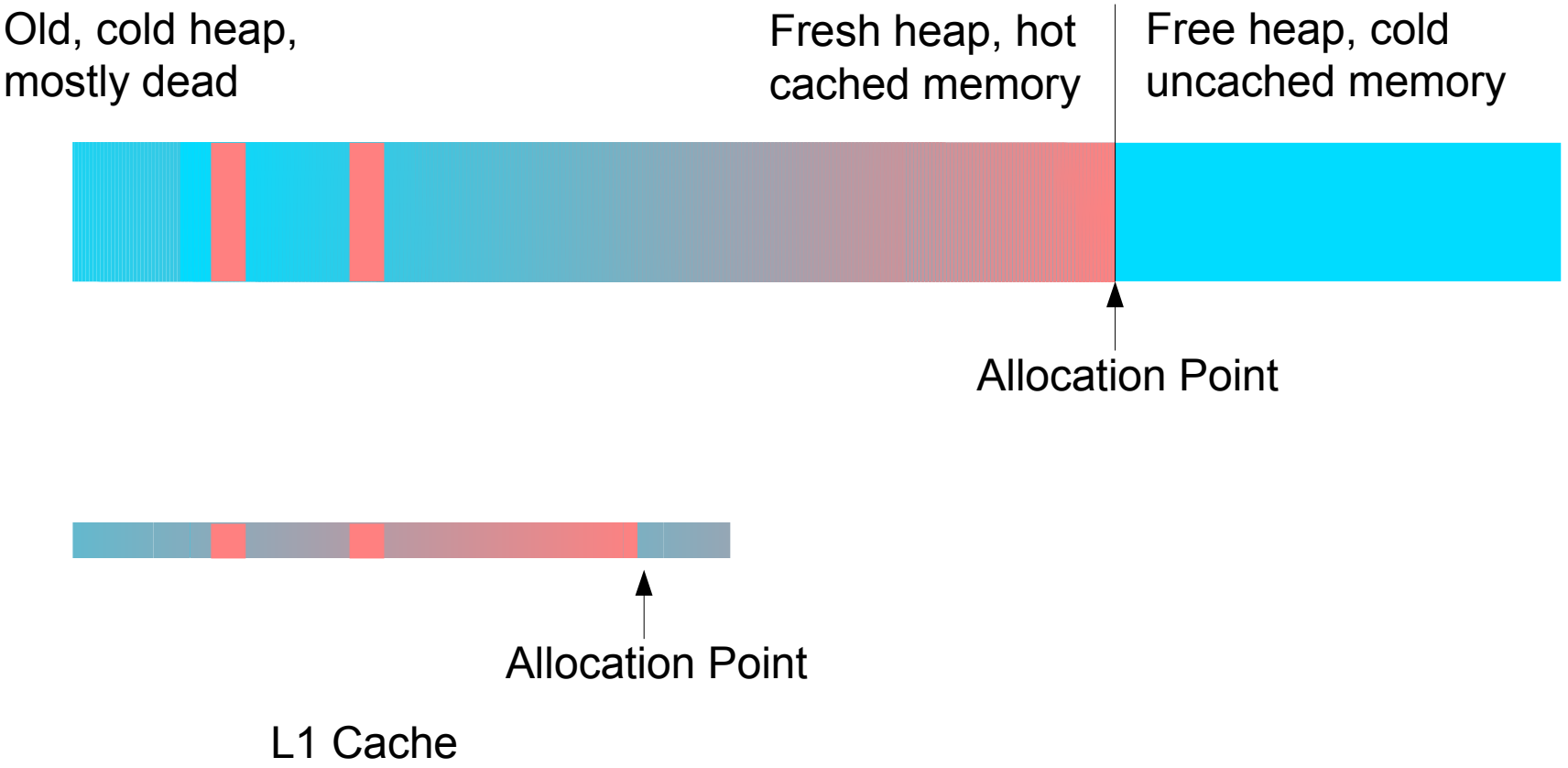

 

# Generational GC



AZUL SYSTEMS

## Memory usage, over time



# Stack Based Allocation



AZUL SYSTEMS

- Objects are allocated on the stack
  - Stack space reserved on function entry
- Deallocate is “free” on function return
- Has similar direct costs to Generational GC
  - Orthogonal to Generational GC
- Recycles memory instead of streams memory
  - Has a smaller cache footprint
  - Fewer misses, stalls
- Requires knowledge about object lifetime
  - Error if object “escapes” function return

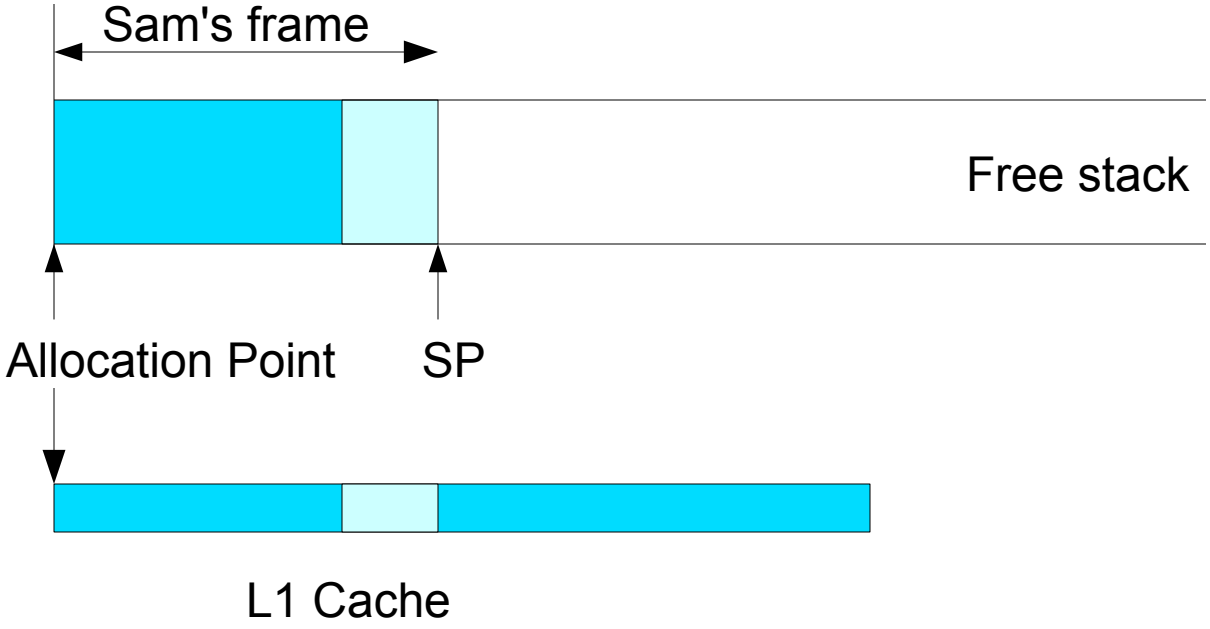
 

# Stack Based Allocation



AZUL SYSTEMS

***Call Sam***




```
Sam {
   a = new;
   b = new;
   Ted();
   bad = Uli();
   bad.crash;
}
```

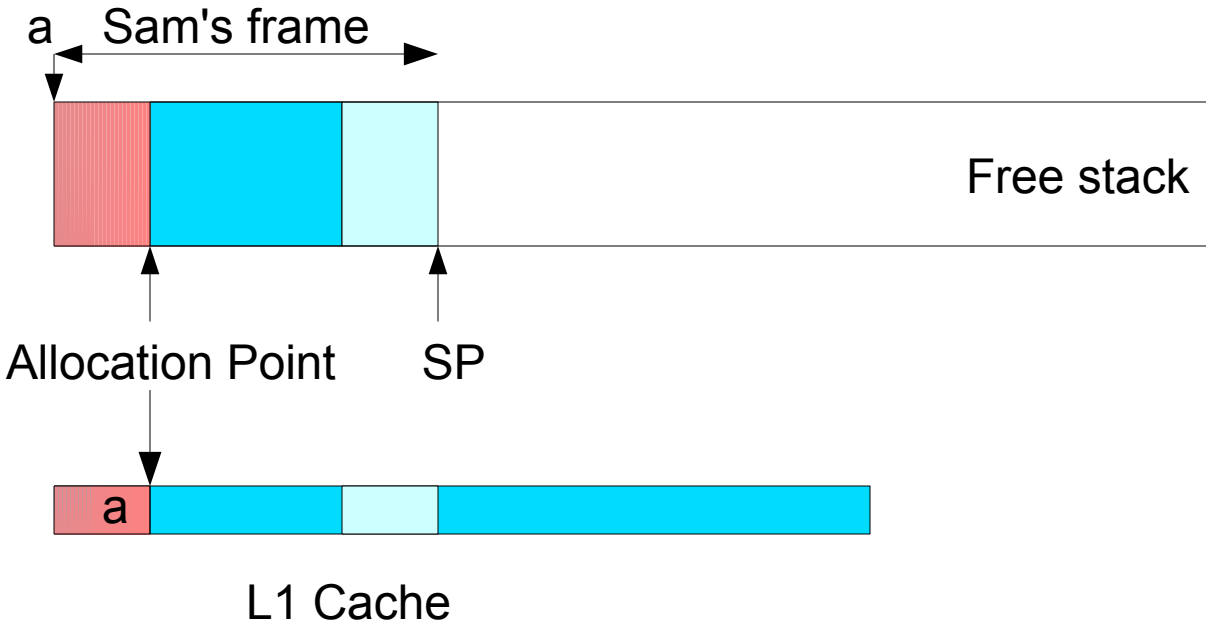
 

# Stack Based Allocation



AZUL SYSTEMS

## *Allocate object a*




```
Sam {
   a = new;
   b = new;
   Ted();
   bad = Uli();
   bad.crash;
}
```

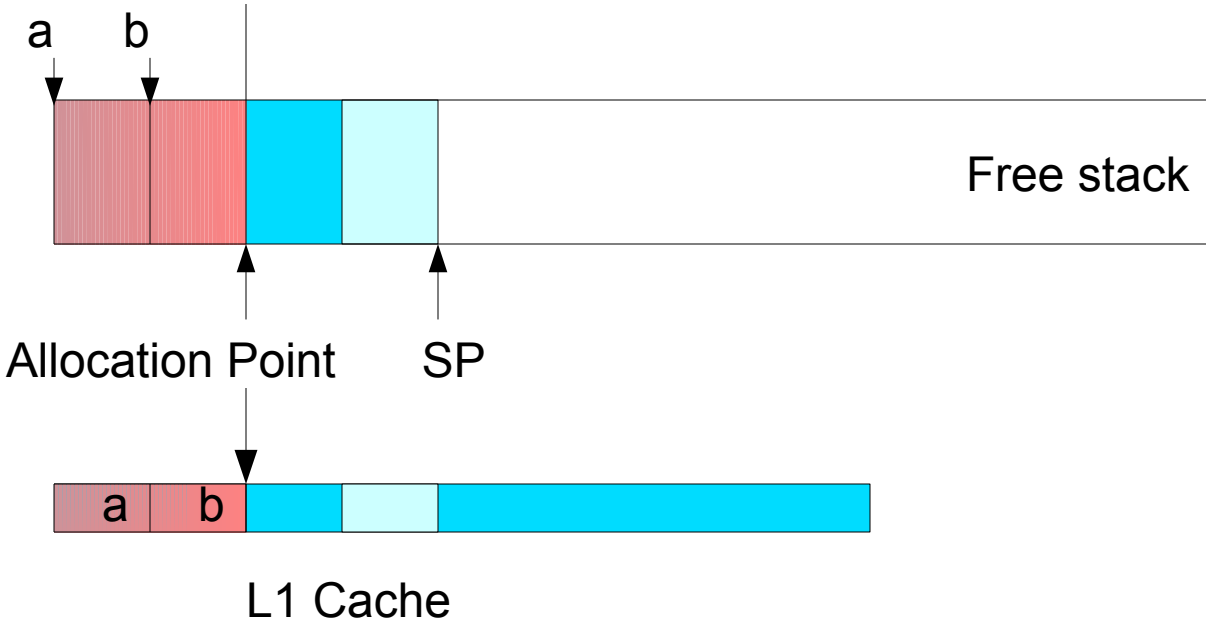
 

# Stack Based Allocation



AZUL SYSTEMS

## *Allocate object b*




```
Sam {
   a = new;
   b = new;
   Ted();
   bad = Uli();
   bad.crash;
}
```

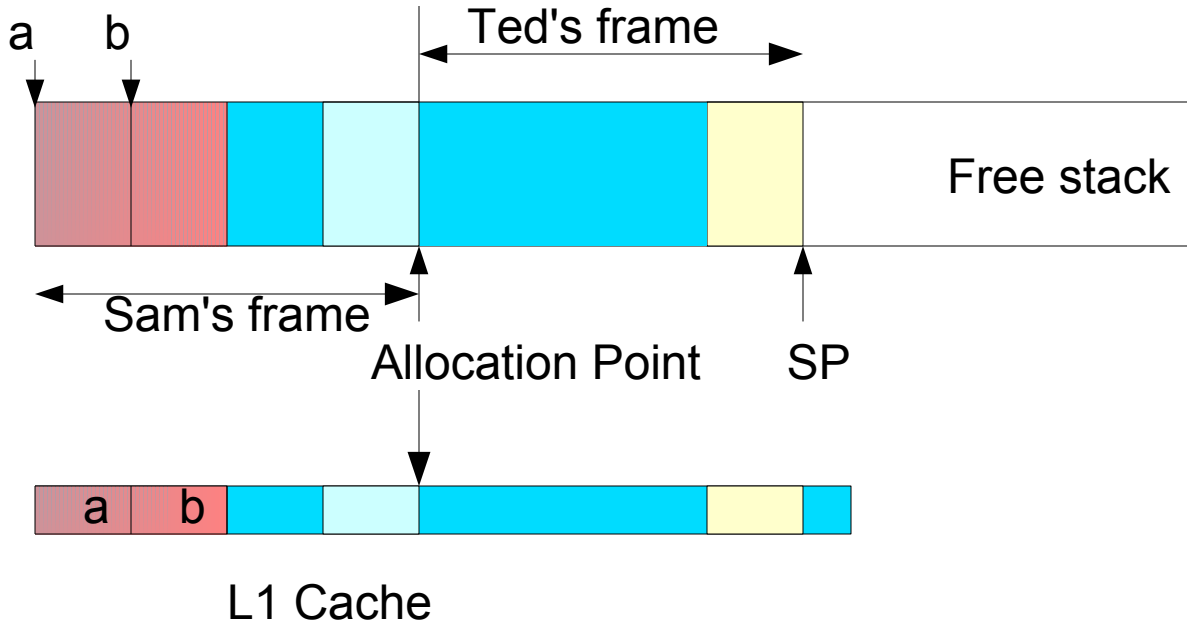
 

# Stack Based Allocation



AZUL SYSTEMS

***Call Ted***




```
Sam {
   a = new;
   b = new;
   Ted();
   bad = Uli();
   bad.crash;
}
Ted {
   c = new;
   return;
}
```

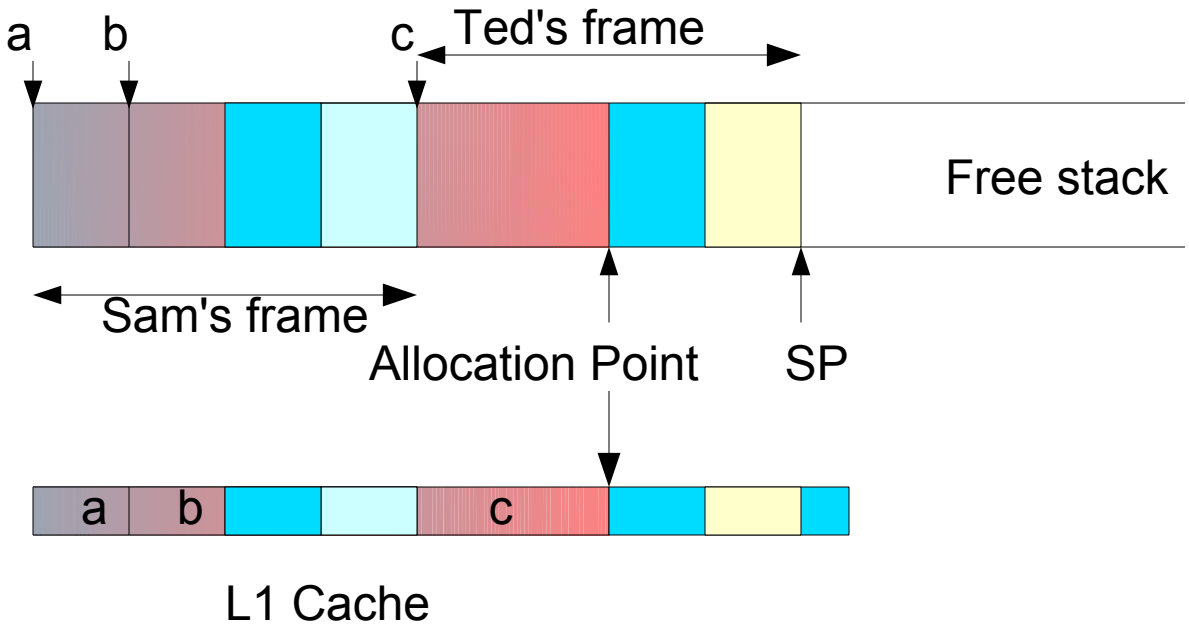
 

# Stack Based Allocation



AZUL SYSTEMS

## *Allocate object c*




```
Sam {
   a = new;
   b = new;
   Ted();
   bad = Uli();
   bad.crash;
}
Ted {
   c = new;
   return;
}
```

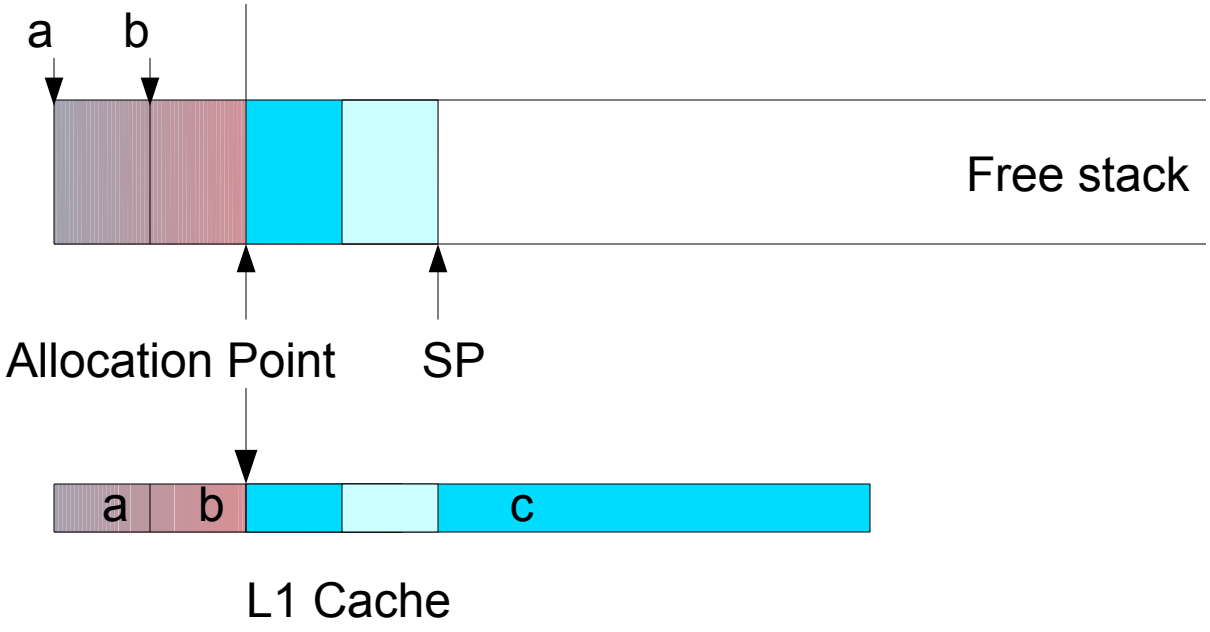
 

# Stack Based Allocation



AZUL SYSTEMS

## *Exit Ted, free c*




```
Sam {
   a = new;
   b = new;
   Ted();
   bad = Uli();
   bad.crash;
}
Ted {
   c = new;
   return;
}
```

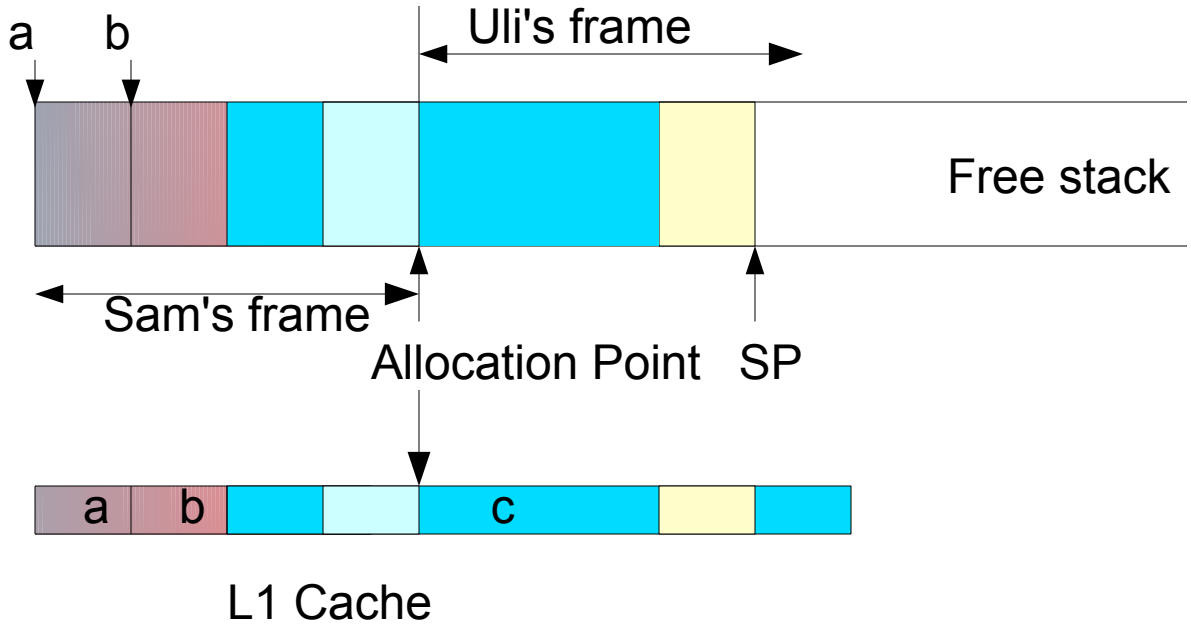
 

# Stack Based Allocation



AZUL SYSTEMS

## *Call Uli*




```
Sam {
   a = new;
   b = new;
   Ted();
   bad = Uli();
   bad.crash;
}
Uli {
   d = new;
   return d;
}
```

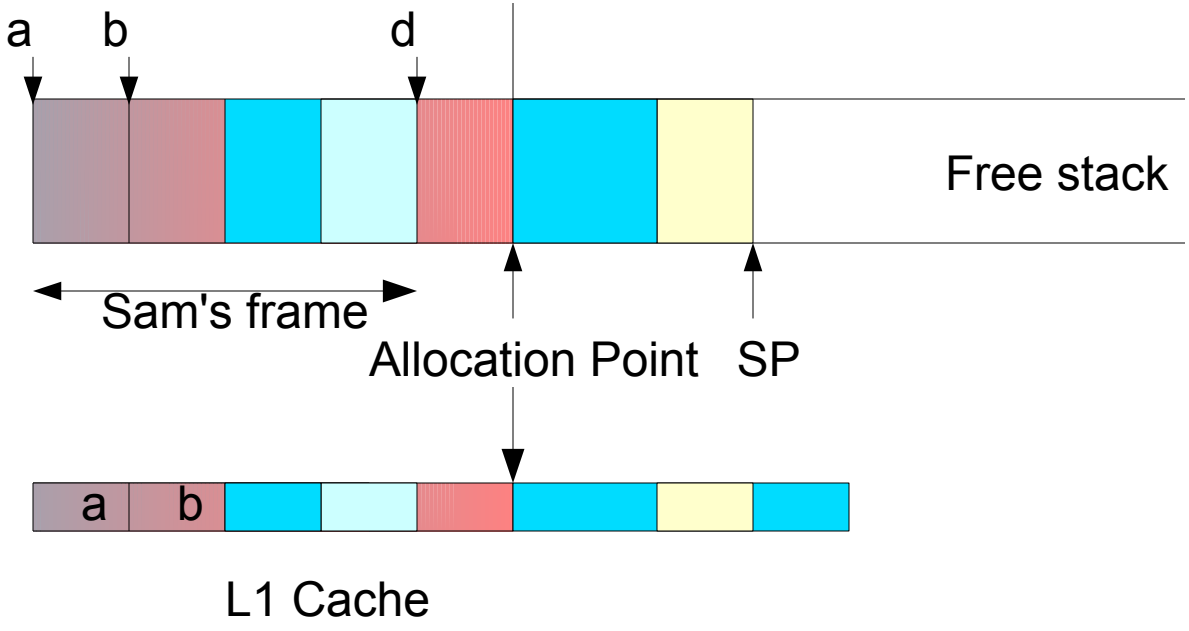
 

# Stack Based Allocation



AZUL SYSTEMS

## *Allocate object d*




```
Sam {
   a = new;
   b = new;
   Ted();
   bad = Uli();
   bad.crash;
}
Uli {
   d = new;
   return d;
}
```

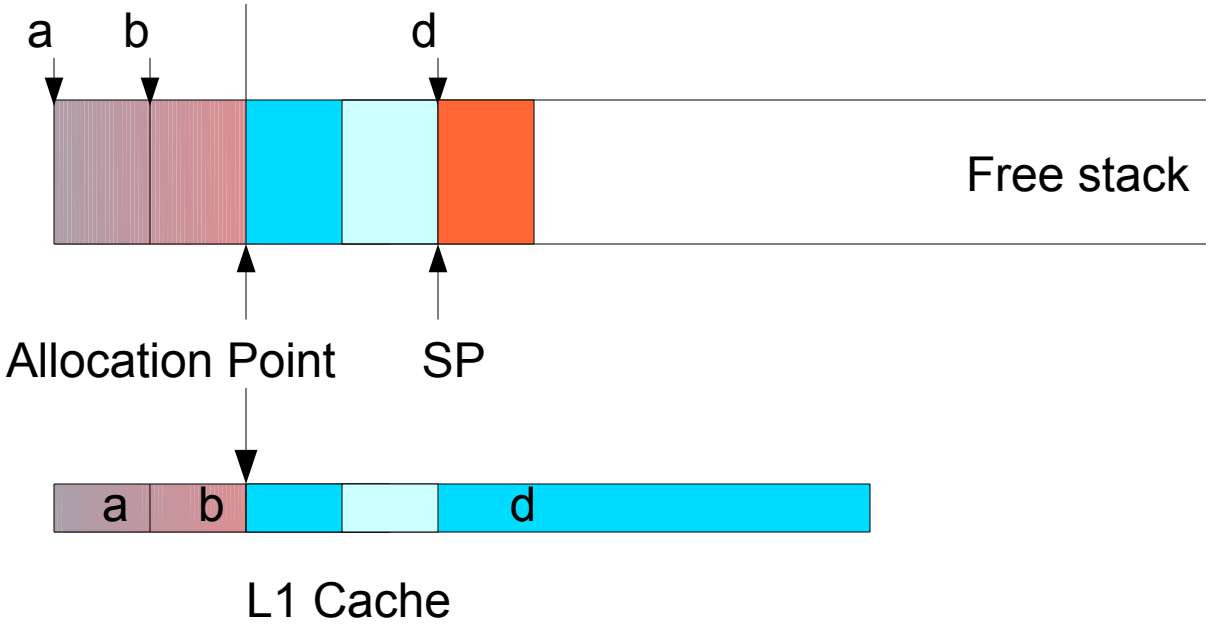
 

# Stack Based Allocation



AZUL SYSTEMS

## *Exit Uli, return d*




```
Sam {
   a = new;
   b = new;
   Ted();
   bad = Uli();
   bad.crash;
}
Uli {
   d = new;
   return d;
}
```

# Stack Based Allocation



AZUL SYSTEMS

- Fast allocate and free
- Better cache behavior
- Only works for objects with stack lifetime
- How do we know object lifetime?
- Escape Analysis
  - Precise, conservative
  - Expensive at compile time
- Escape Detection
  - Imprecise, optimistic
  - Expensive at run time (requires write barrier)

# Escape Detection



AZUL SYSTEMS

- Escape Analysis
  - Conservatively correct
  - Objects known to never escape
  - Allows objects to be en-registered
  - Fairly expensive (for a JIT) except in trivial cases
- Escape Detection
  - Optimistic; some objects may escape
  - Requires a complex write barrier and fix-up
- Write barrier can be done in hardware
- Requires minor hardware change
  - Azul is implementing a Java™ VM with custom hardware

# Escape Detection Hardware



AZUL SYSTEMS

- Allocate objects in stack frames
- Steal some address bits for frame depth (FID)
- FID bits stored in object pointer
- Hardware ignores FID on a load (getfield)
- Hardware does range check on a store (putfield)
  - Storing old FID into new FID object is OK
  - Storing new FID into old FID object TRAPS
- Trap requires expensive fixup
  - Crawl only new frames on this thread's stack
  - Move object, adjust FIDs

# Fixup is Expensive



AZUL SYSTEMS

- Fixup must be rare
  - *Object factories always escape object!*
- Tag allocation sites with relative allocation depth
- Allocate object in frames with longer lifetimes
  - Or directly in heap if needed
- Start small
- Adjust tag upwards with each failure
- *Rapidly converges in practice!*
- No expensive analysis

# How Big are Frames?



AZUL SYSTEMS

- Check for frame overflow on allocation
  - Same as test-and-bump allocation
- If it fits, fine.  If not...
- Allocate an overflow area on the side
  - Stamp same FID into objects
  - Hardware checks FID, not actual stack address
- Adjust call prolog code
  - Make a larger frame for next time
- *Rapidly converges in practice!*

# Never-Exit Frames



AZUL SYSTEMS

- Some top-level frames never exit
  - Also loops, and JIT may inline
- But will accumulate dead objects
  - Objects only freed on a frame exit!
- Periodically need a thread-local GC
  - Toss out dead objects
  - Compact overflow areas
  - Can adjust frame sizes

# Inlining helps Site Tagging



AZUL SYSTEMS

- Allocation site behavior may depend on call path
- Same site may want different depth tags depending on caller
- Hot code gets compiled, with inlining
- Inlining clones allocation sites
- Each inlined clone gets its own depth tag

# Reverse: Tags Guide Inlining



AZUL SYSTEMS

- Depth tags tell empirical lifetime
  - Of objects allocated here
- Likely can prove actual=empirical with Escape Analysis
- Inline up to tagged depth
- Run Escape Analysis on inlined code
  - E.A. is cheap in a single compilation unit
  - E.A., if successful, is better than Detection
  - Can en-register whole objects
- Avoid E.A. If allocation site escapes

# Prelimary Results



AZUL SYSTEMS

- Modest sized runs of SpecJVM98, SpecJBB, SJAS
- About ½ of all objects stack allocated
- With 4meg overflow areas
  - Typical stack GC needed every 6-8sec
  - Stack GC takes <1msec
- Some anomalies observed
- Lots of stack-allocated 2K and 4K arrays
  - May be faster to heap allocate
    - Because of Azul fast zero'ing hardware
  - Frames forced too large too quick
- JIT inlines allocation into loops

# Software Escape Barrier



AZUL SYSTEMS

- Azul has precise per-frame E.D. hardware
- Possible to make an software E.D. barrier
- Compare raw stack addresses & trap
  - Requires all stacks be on same side of heap
- Store FID just prior to object header in stack
- Trap checks FID before declaring an escape
  - load, load, compare, branch
- Trap can fail repeatedly for non-escaping
  - Two objects in same frame but wrong order
  - No good overflow solution

# Summary



AZUL SYSTEMS

- First GC-specific hardware in a long time
- Looks good on paper
- Looks good in prelimary results
- Real Thing is a little ways off yet
  - Needs tuning
  - Performance-proof in large runs