Revisiting the “Perc Real-Time API”

Kelvin Nilsen, Chief Technology Officer Java, Atego Systems

The extended history of real-time Java

- Two research papers published in late 1995 and early 1996 represent the original birth of real-time Java
- Market response to these papers was overwhelmingly positive
  - Nearly 900 copies of the draft real-time Java API were downloaded in the 8 months following first publication in January 1996
  - Multiple RTOS vendors were “hearing from their customers” that they wanted this technology; I received multiple invitations to leave academia
  - Enthusiastic response motivated NIST to host “standardization” meetings on real-time Java
  - Got attention of Sun Microsystems, who did not want outsiders to be “defining” Java.
    - Their response was formation of the Java Community Process and formation of the JSR-1 expert group

atego™

Where are we today?

- Over ten years after the RTSJ became an official standard, real-time Java as defined by RTSJ is still primarily a “research topic”
  - Very difficult to find commercial deployments
  - There is talk of a few defense system deployments, but very limited “real data” on how well RTSJ has worked in these deployments
  - Plenty of opportunity for research projects
- Meanwhile, about six months ago, a well-known commercial avionics and defense technology supplier contacted Atego to request access to the original Perc API (as described in the paper first published in 1996)
- Would a different real-time Java “standard” have yielded different outcomes?
- Is it still possible to correct the course of real-time Java?

Before there was "Real-Time Java", there was real-time Java

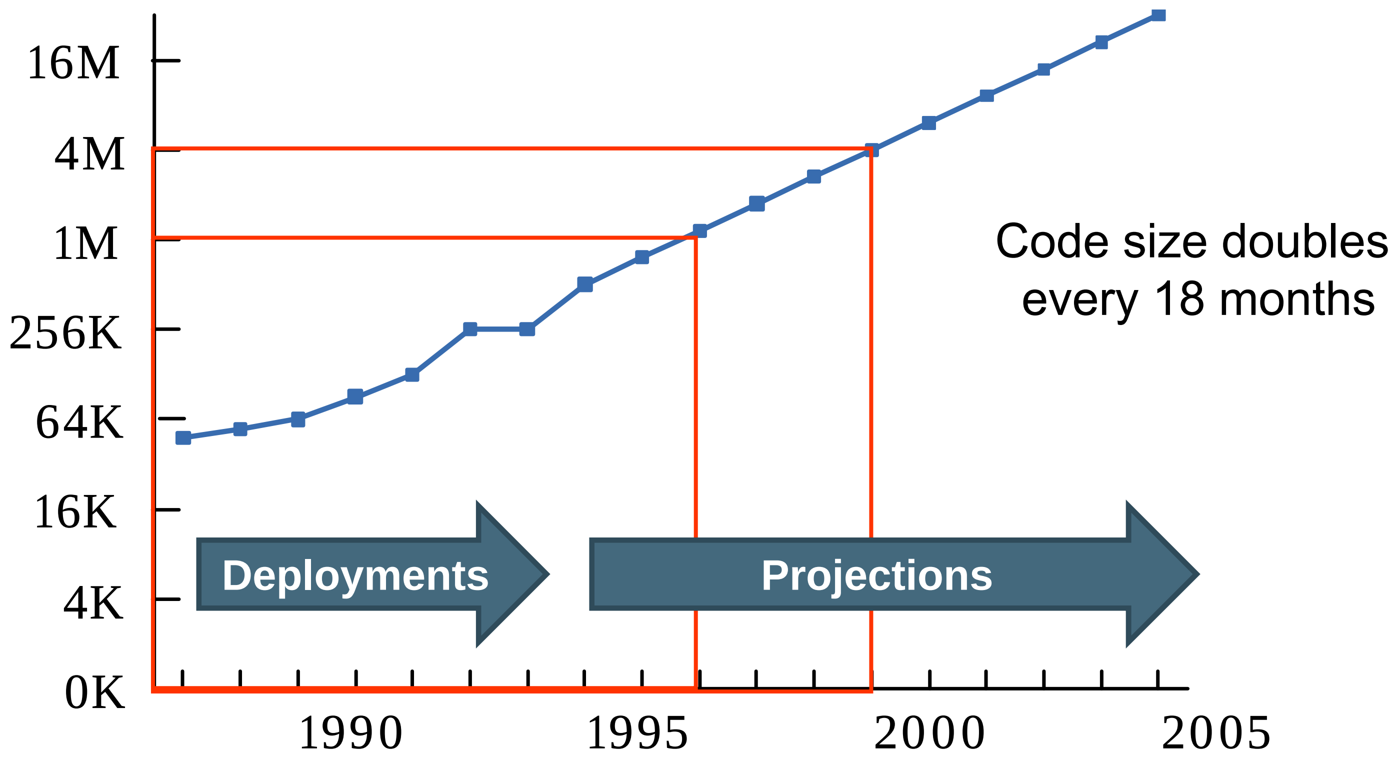
Origins

- 1980's: earned way through graduate school setting up 8-bit computers, soldering RS-232 cables, writing a hard disk device driver, and authoring the VersaCom interrupt-driven IBM PC telecommunications program
- Univ. of Arizona Computer Science convert from Physics undergraduate degree
  - Graduate programming language coursework placed strong emphasis on language design and programming language expressiveness
  - Gained strong appreciation for expressive power of programming languages
  - SR (Synchronizing Resources) language design with Greg Andrews
  - Icon (successor to SNOBOL4) language design with Ralph Griswold
  - Ph.D. topic: concurrent real-time version of Icon called "Conicon", including my early work on real-time garbage collection (1985-88).
- Subsequent NSF-funded research on real-time garbage collection for C++
  - I had a solution looking for a problem
  - But real-time programmers kept telling me they didn't "need" garbage collection, …
  - and they were right
- Transitioned work to Java beginning in late 1995
  - Backing from angel investor, DARPA, venture capitalists

Considerations in Design of Original PERC Real-Time API

- Target domain must be much broader than traditionalist "real-time"
  - Think "Star Wars", the movie
- Must address shortcomings of then-current "real-time practice":
  - Non-portable: real-time code is generally targeted, debugged, analyzed, tailored for a specific platform, assumptions are rarely documented
  - Non-scalable: every real-time programmer has to worry about what every other real-time programmer is doing (because of contention for CPU time, memory, network bandwidth, synchronized resources)
  - Non-modular: real-time programs are "monolithic"; there are no independent components, every part is aware of and dependent on every other part
  - Impractical: though scheduling theory is solid, analysis of execution times is overly conservative (by 100x), especially on modern processors, and interesting real-time workloads are not always predictable

Context: Moore's Law of Software Growth



Source: Philips Semiconductor data for high-end television receiver

Context: Moore's Law of Software Growth

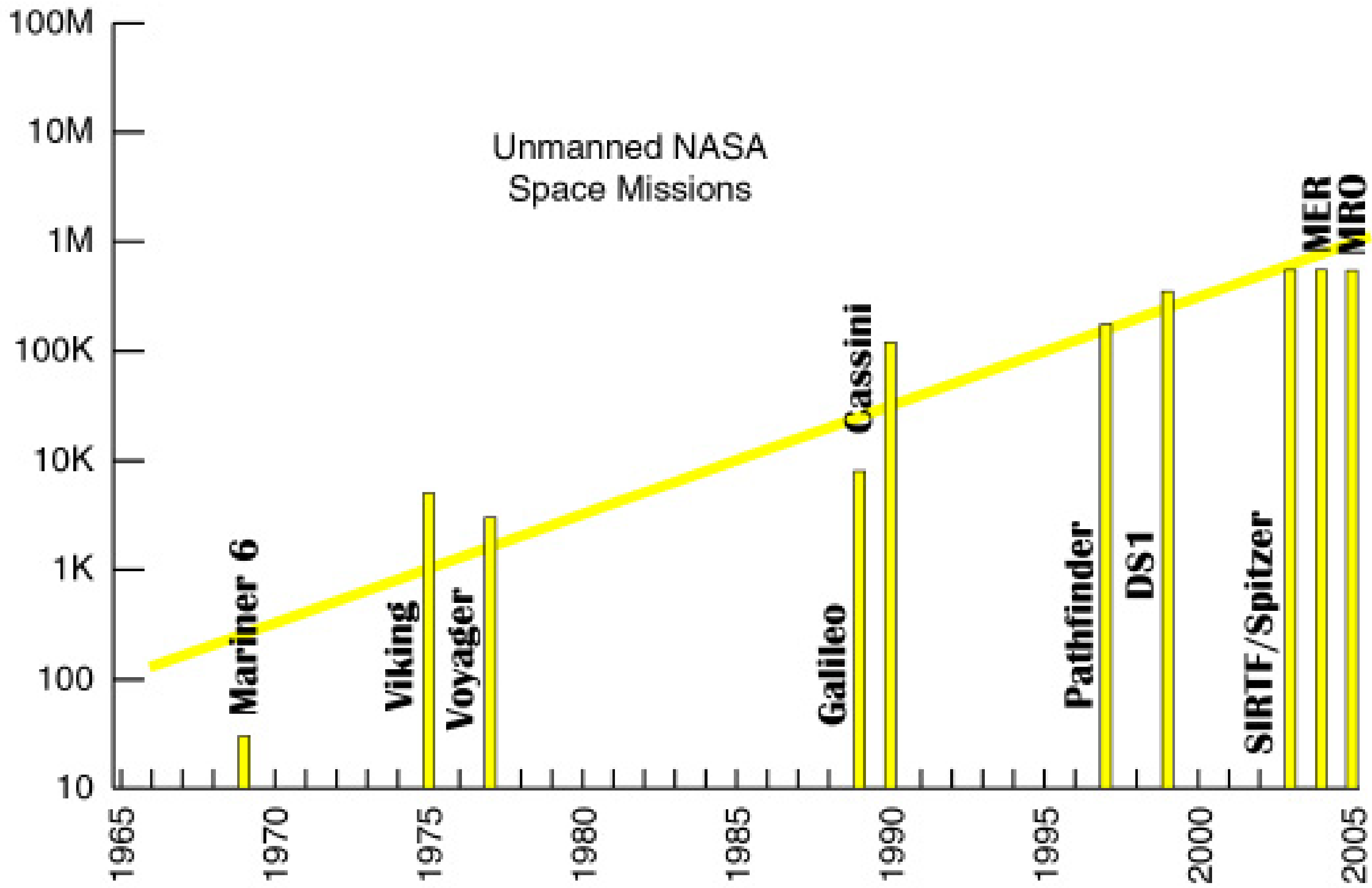
Relevance:

1. As application sizes grow, it is no longer practical to implement from scratch all new software for each new product configuration.
2. As ROM sizes grow, so do processor capabilities. More modern microcontrollers are often end-of-lifed 5-10 years after they are first commercially available.
3. When each new deployment more than doubles the code from previous configuration, there is not sufficient budget to "port" or "fully test" all of the new code on the integration platform.
4. Code reuse must be simple, effortless, and code integration must be seamless.

Subsequent Data Point

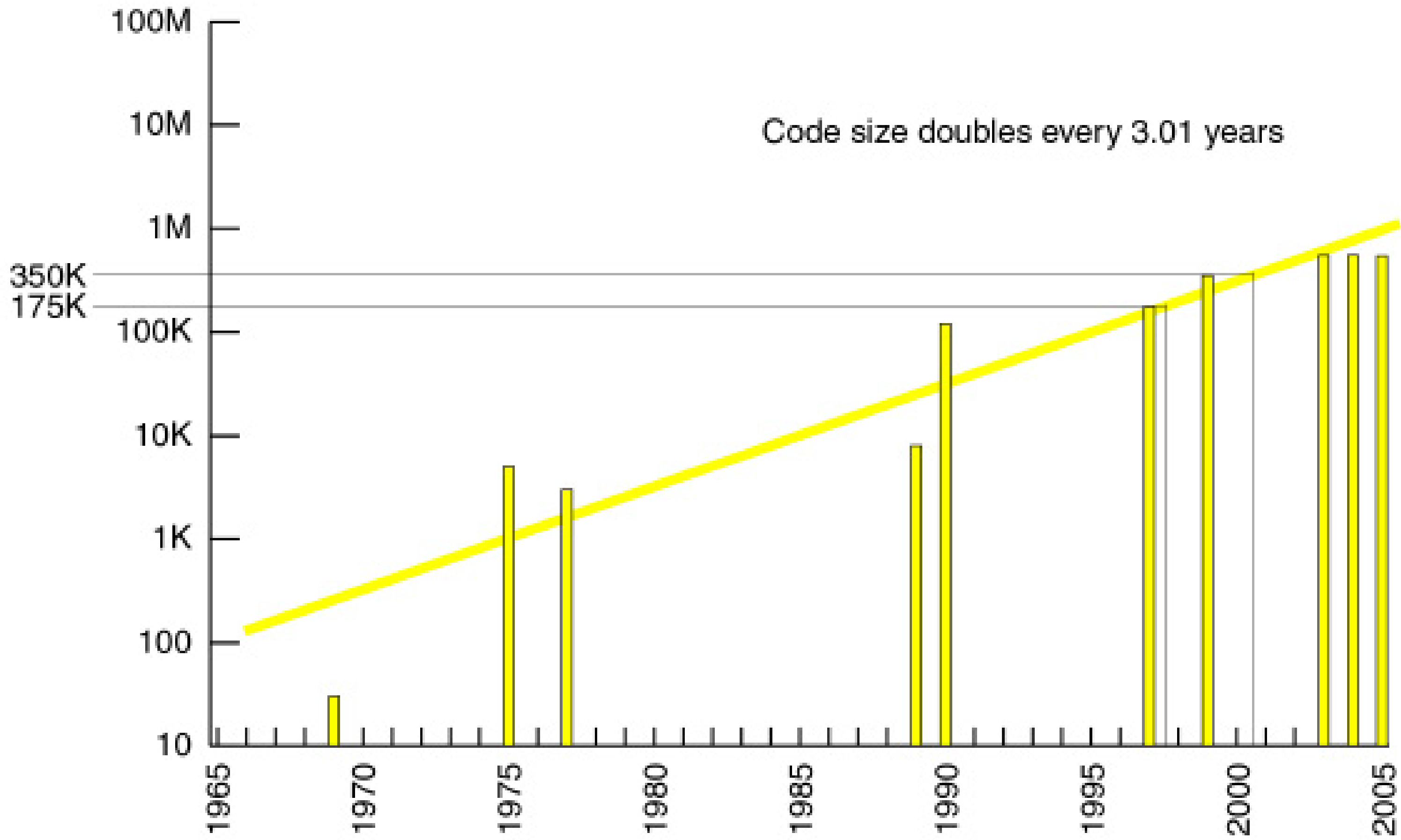
- GM CTO Anthony Scott (Oct. 2004):
  - “More than one-third of the cost of GM's automobiles now involves software and electronic components”
  - “Cars had approximately 1 million lines of software code in 1990, but this number will jump to 100 million by 2010”
  - Translation: code size doubles every 3 years

Further Corroboration:
Growth in Complexity of NASA Space Mission Software



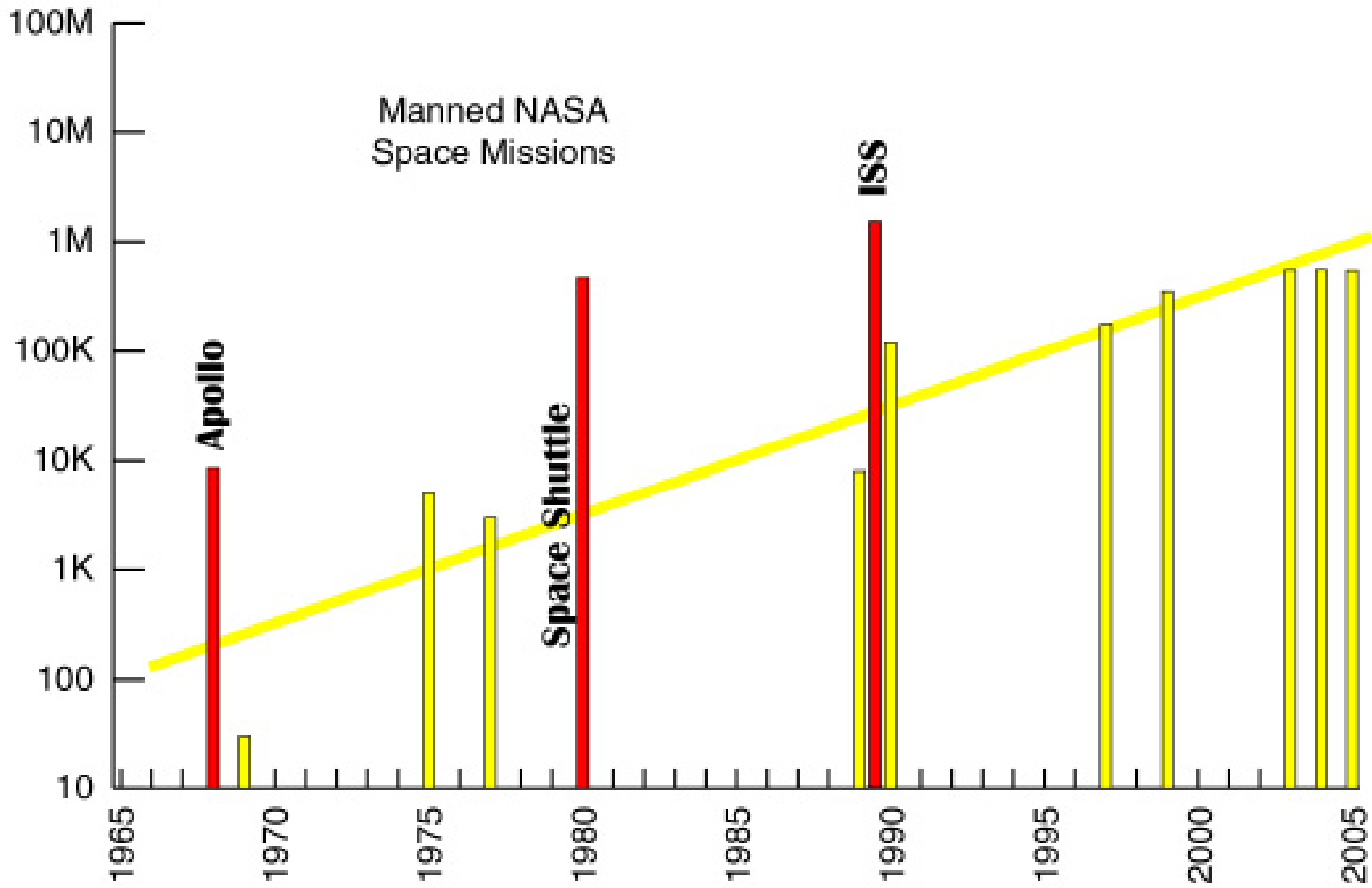
Source: D.L. Dvorak, ed., NASA Study on Flight Software Complexity, 3 March 2009

Further Corroboration:
Growth in Complexity of NASA Space Mission Software



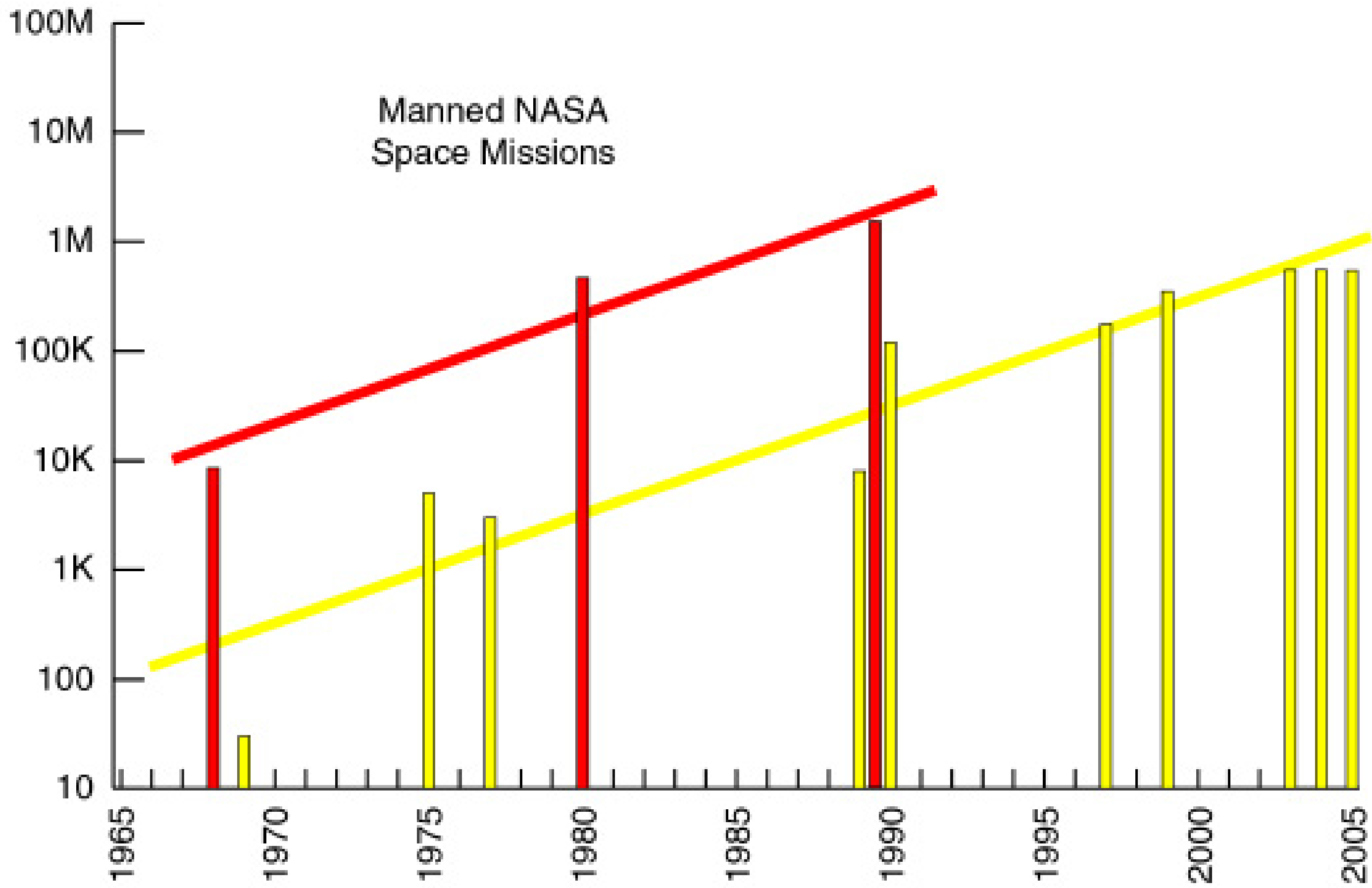
Source: D.L. Dvorak, ed., NASA Study on Flight Software Complexity, 3 March 2009

Further Corroboration: Growth in Complexity of NASA Space Mission Software



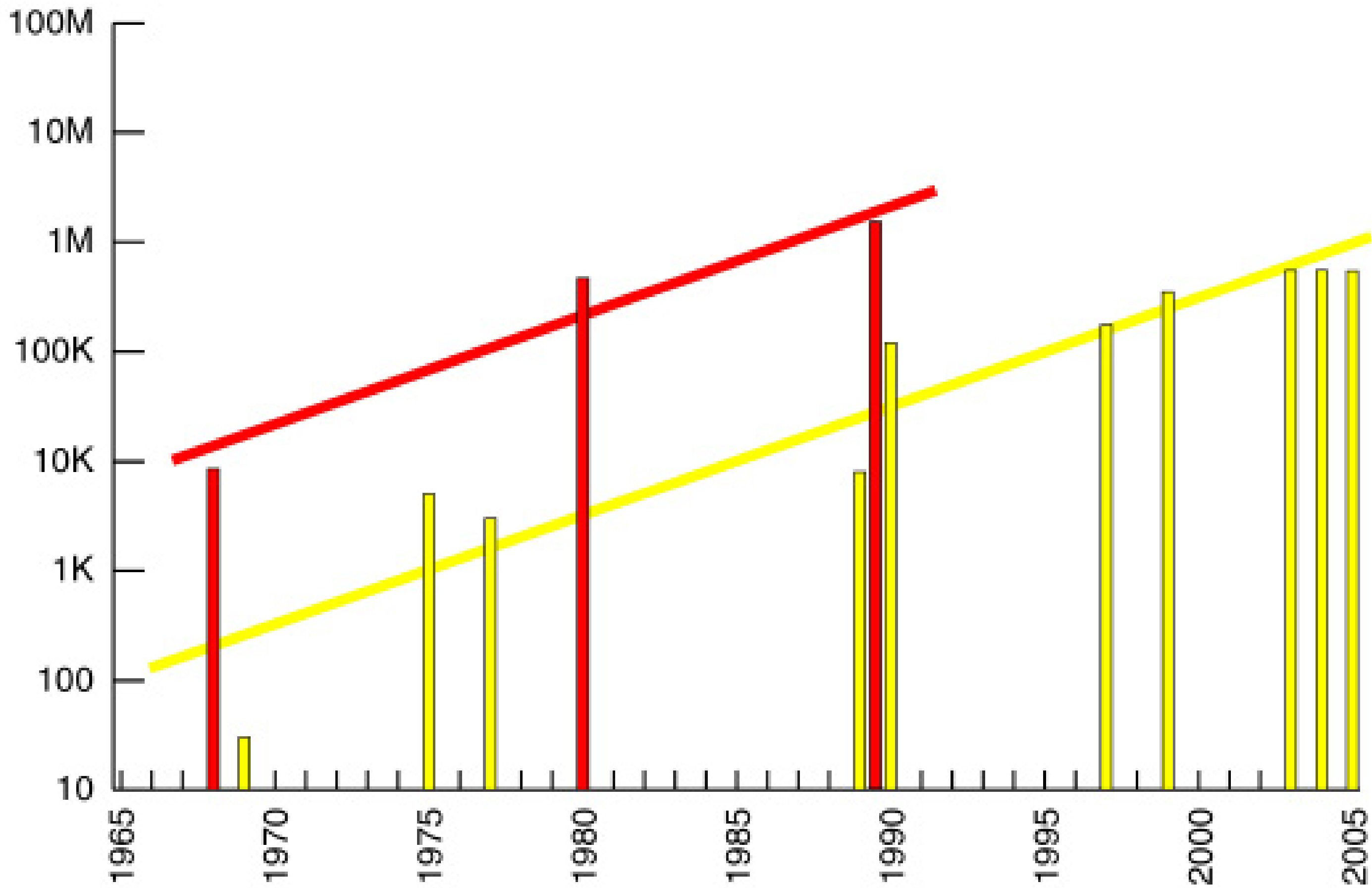
Source: D.L. Dvorak, ed., NASA Study on Flight Software Complexity, 3 March 2009

Further Corroboration:
Growth in Complexity of NASA Space Mission Software



Source: D.L. Dvorak, ed., NASA Study on Flight Software Complexity, 3 March 2009

Further Corroboration:
Growth in Complexity of NASA Space Mission Software

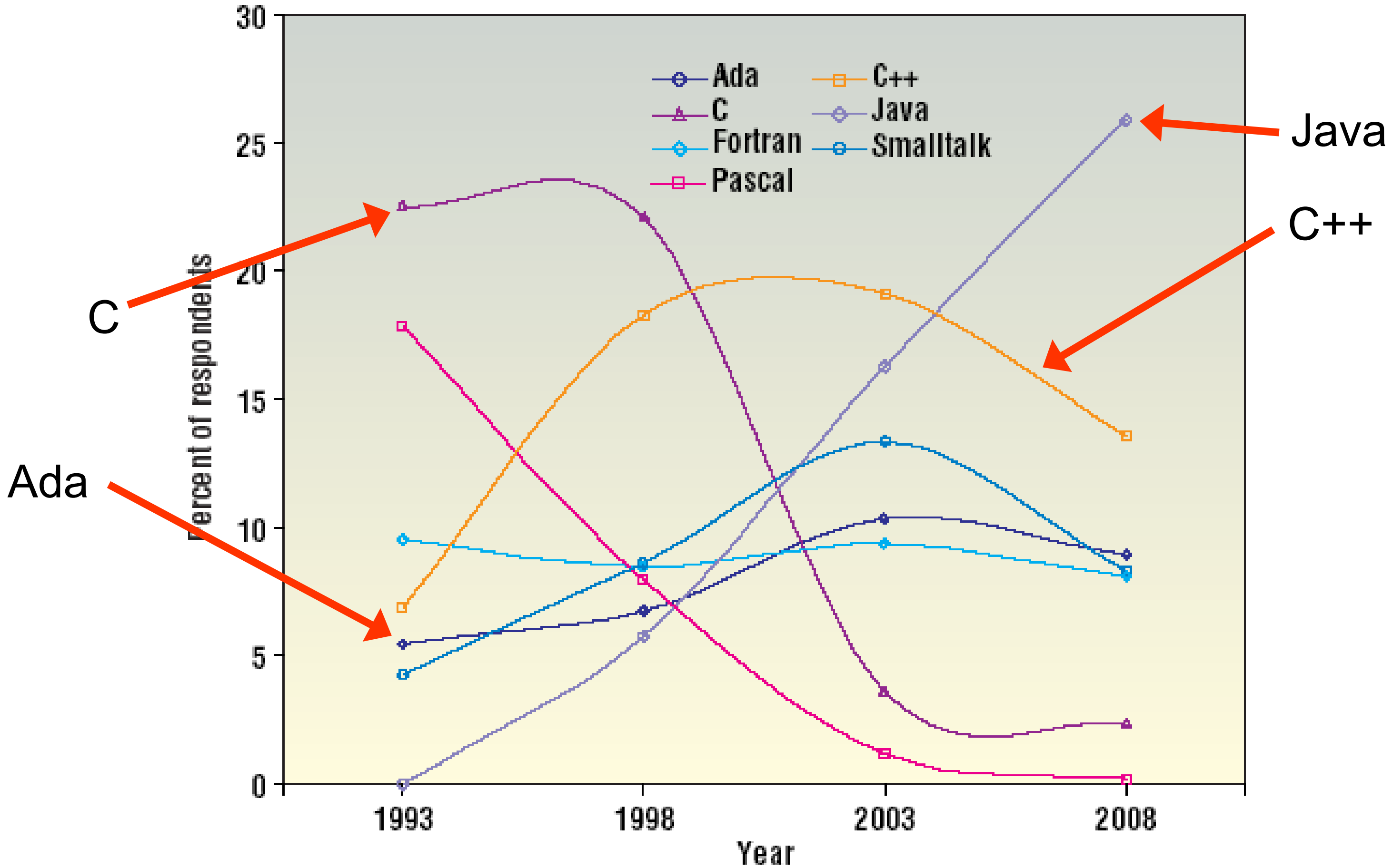


Source: D.L. Dvorak, ed., NASA Study on Flight Software Complexity, 3 March 2009

Initial Impressions of Java (1995)

- My sense at the time was that Java provided a strong foundation on which to build a technology that could address the emerging needs of real-time developers
  - Superior portability
  - Strong encapsulation and object-oriented abstraction to support scalable expansion of large software systems from independently developed software components
- But a variety of issues would need to be addressed in order to deliver the benefits of traditional Java to the real-time domain

Statistical Study of Java Language Adoption



Usage Trends of the 8 Most Popular Programming Languages
"An Empirical Study of Programming Language Trends", Chen, Dios, Mili, Wu, Wang
IEEE Software, May/June 2005

What makes one language more popular than another?

- Top Intrinsic Factors (with statistical correlations)
  - Machine independence (portability) (0.8876)
  - Extensibility (scalability) (0.7625)
  - Generality (scalability) (0.6913)
  - Simplicity (-0.4703)
  - Implementability (-0.3390)
  - Reliability (scalability) (0.3199)

Extrinsic Factors Correlations

■ Extrinsic Factors: Support from

- Institutions (e.g. university curricula)
- Industry (corporate endorsements, guidelines, adoption)
- Government (research funding, procurement guidelines)
- Organizations (e.g. JUG)
- Grass roots (how many count this as “primary or favorite language?”)
- Technology (vendor support, 3rd party involvement)

The Perc API Vision

- Real-time programmers design, implement, and debug real-time software components independent of the deployment platform and execution context
- Individual real-time components can choose whether to budget CPU and memory resources conservatively or aggressively
- The same real-time components run reliably on
  - Faster and slower computers
  - Yesterday's, today's, and tomorrow's computers
  - As standalone applications, and as individual contributors to complex systems
  - On systems with abundant excess resources, and systems that are oversubscribed
  - With independently managed variable service quality

Overview of the Original PERC Real-Time API

- Depends on real-time garbage collection, predates NoHeapRealtimeThread concepts
- Real-time software components are structured as *real-time activities*, each comprised of
  - Multiple tasks
  - A CPU-time budget
  - A live-memory budget
  - A memory allocation rate budget
  - Resource budgets consist of a guaranteed allotment and an expected allotment
  - A configure method that is used to determine the activity's resource needs on the current platform
  - A negotiate method that allows activity to approve proposed budgets

How does configure() determine resource needs?

- Running benchmarks, using Perc services to understand the benchmark behavior:
  - “Time” consumed by benchmark thread(s)
  - Memory allocated by benchmark thread(s)
- Remembering results of previous runs on this or similar platforms using the same Perc services to monitor resource consumption
- Static analysis services are provided as part of the dynamic Perc API run-time environment
  - WCET and EET for code written in very restrictive style (facilitated by compile-time analysis)
  - WC memory consumption of particular classes and objects

Measures of "Time"

- Activity time budgets are expressed in terms of *Execution Time*, which is the combination of:
  - CPU Time
  - Block Time
- *CPU Time* is the combination of
  - *Delay Time* (spin loops, even if an implementation uses blocking)
  - Time executing instructions
- *Block Time* includes
  - *Endowed Time* (when priority inheritance mechanism endows this thread's CPU time allotment to another thread, including GC thread(s))
  - Time waiting to acquire synchronization lock, wait()ing, or suspended in I/O
- *Inherited Time* is time spent by this thread running on behalf of other threads

How does a real-time task manage its budgeted resources?

- With judicious application of **timed** and **atomic** statements:

```
atomic {
  approximation = initial_approximation;
  initializeGlobalState(approximation);
}
timed (Time.us(250)) {
  while (refinementDesirable(approximation)) {
    approximation = refineApproximation(approximation);
    atomic {
      updateGlobalState(approximation);
    }
  }
}
```

How does a real-time task manage its budgeted resources?

- With judicious application of **timed** and **atomic** statements:

```
atomic {
  approximation = initial_approximation;
  initializeGlobalState(approximation);
}
timed (Time.u
  while (refin
    approxim
    atomic {
      updateGlobalState(approximation);
    }
  }
}
```

- o Atomic statements always execute all or nothing, are abort deferred
- o Must use analyzable subset of full Java
- o To avoid overrunning time budget, may check timing compliance on entry

How does a real-time task manage its budgeted resources?

- With judicious application of **timed** and **atomic** statements:

```
atomic {
  approximation = initial_approximation;
  initializeGlobalState(approximation);
}
timed (Time.us(250)) {
  while (refinementDesirable(approximation)) {
    approximation = refineApproximation(approximation);
    atomic {
      updateGlobalState(approximation);
    }
  }
}
```

- o Timed statements are parameterized with "execution time"

How does a real-time task manage its budgeted resources?

- o Atomic uses priority ceiling emulation to avoid blocking, localizing analysis of execution time

■ With ju

```
atomic {
   approximation = initial_approximation;
   initializeGlobalState(approximation);
}
timed (Time.us(250)) {
   while (refinementDesirable(approximation)) {
      approximation = refineApproximation(approximation);
      atomic {
         updateGlobalState(approximation);
      }
   }
}
```

Types of Tasks

- PeriodicTask repeatedly performs the event handling sequence

```
task.startup();        // execution-time bounded
timed(work_budget) {
   task.work();        // stylized Java enables variable service quality
}
task.finish();         // execution-time bounded
```

- SporadicTask event handler is triggered by asynchronous events with a maximum trigger frequency
- SpontaneousTask is a one-shot event handler, often activated in response to an unanticipated "opportunity"
- OngoingTask is a background task, running with a fair share of CPU resources; may consist of traditional Java code; may use atomic statements to share data with other tasks

Spontaneous Activities

- A SpontaneousTask is only allowed within a SpontaneousActivity
- Only SpontaneousTasks are allowed in SpontaneousActivities
- An application introduces a spontaneous activity to the real-time executive, and specifies an upper bound on the time allowed for configuration and negotiation.
  - If the real-time executive is able to accept the proposed spontaneous activity's workload and begin its execution within the specified time bound, it is added to the workload
  - If not, the application is told that there are insufficient resources to perform the spontaneous activity at this time

Issues with the Bodies of Work Tasks

- Timing out traditional Java code is problematic
  - Aborting a Java thread may leave shared data in an incoherent state
    - Need a way to defer abortion during certain critical sections of code
    - Synchronized is not the same as abort-deferred
  - If a timed out thread defers its abort request too long, it will consume more than its budgeted time
  - Executing catch and finally clauses associated with aborted try statements will also delay abortion beyond the budgeted time
  - If real-time threads are allowed to consume more time than was budgeted, other real-time threads are pushed off their schedule
    - Local concerns become global concerns

Style Restrictions on Bodies of Work Tasks

- Not allowed to catch a TimeoutException
- Atomic statements must be execution-time analyzable; before entering the abort-deferred body of the atomic statement, confirm that there is enough time to execute to completion
  - Only atomic statements defer abortion; synchronized statements do not
- Catch and finally clauses must be execution-time analyzable
- Upon entry into a try-clause, the timeout clock is skewed forward to account for the time that might subsequently be required to cleanup the context.
  - Suppose catch and finally clauses require 20 μs and the budget for execution of the try-statement is 10 ms, deliver the timeout request at 10 ms – 20 μs

Some thoughts on real-time scalability

- Priorities are not scalable
  - Speak of deadlines instead
- Priorities don't span multiple cores, but deadlines do; SMP priority inheritance based on deadlines works
- In periodic tasks, it is unnatural to "wait" for data or conditions.
  - If a periodic task is expected to process data supplied by other tasks, the design should assure that the data is produced on "schedule"; use atomic statements to safely transfer data between threads
- If waiting is required, it may be more appropriate to use a SporadicTask than a PeriodicTask.

So why didn't we finish what we started?

- Overly ambitious
- Not compatible with most real-time operating systems
- Sun Microsystems, the JCP, and JSR-1
- Microsoft and the J Consortium
- Too messy
- The market became confused, scared, catatonic
- Investors got cold feet
- NewMonics redirected to a more conservative path

What was Plan B?

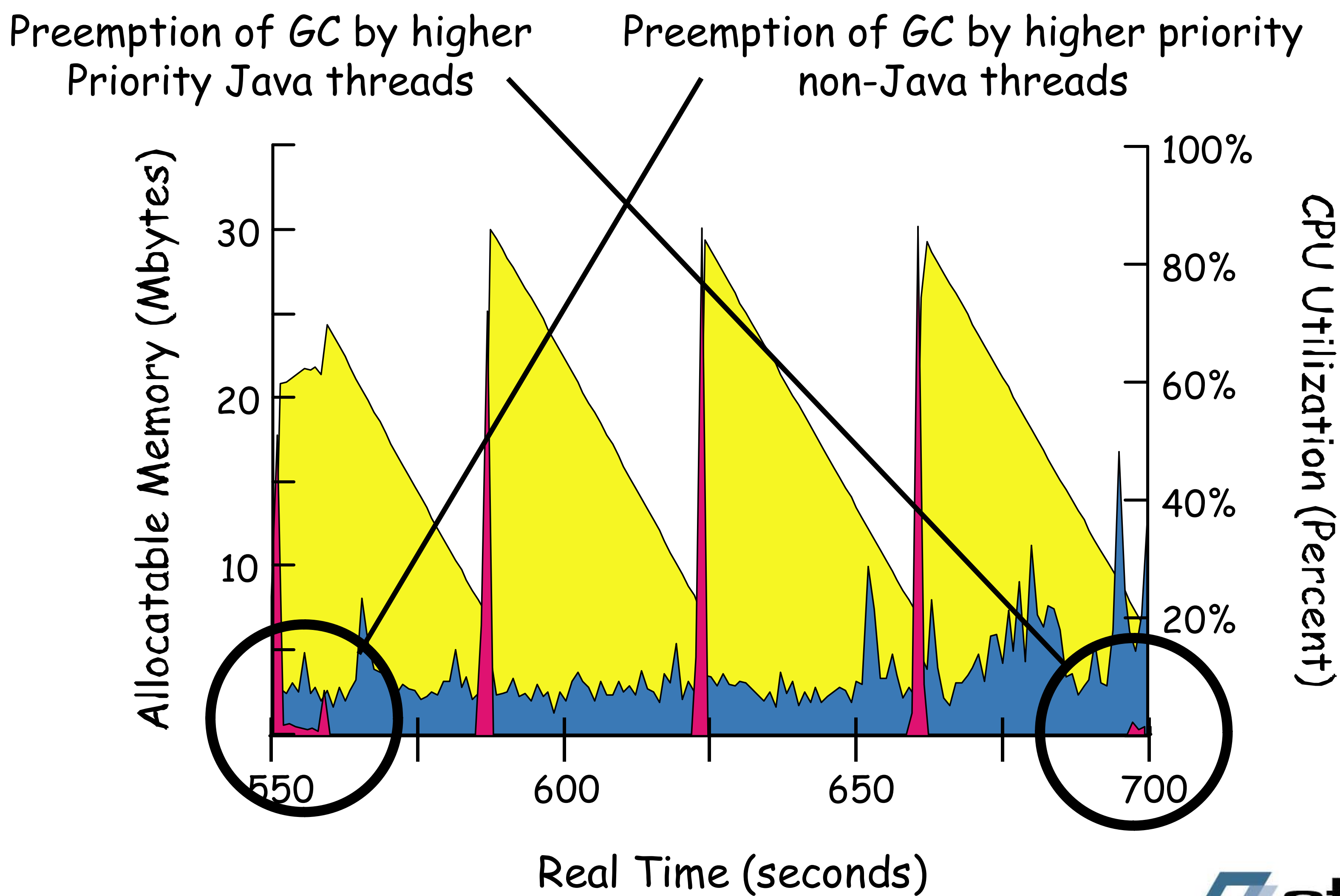
- Java Standard Edition with Real-Time Garbage Collection
  - Typical applications enforce time constraints of 1-100 ms
- Portable Real-Time Scheduling and Synchronization
  - Global dispatching always runs the N highest priority ready Java threads on N available cores
  - Implements priority inheritance on all Java synchronization locks
  - Maintains all thread queues in priority order
  - Optional use of extended priority range (1-32)
- Embedded integrations (RTOSes, processors, ROM)
- Improved Timing Services
  - Monotonically increasing clock for global synchronization
- VM Management Services (monitor, control resource utilization)

Perc Ultra SMP Real-Time GC

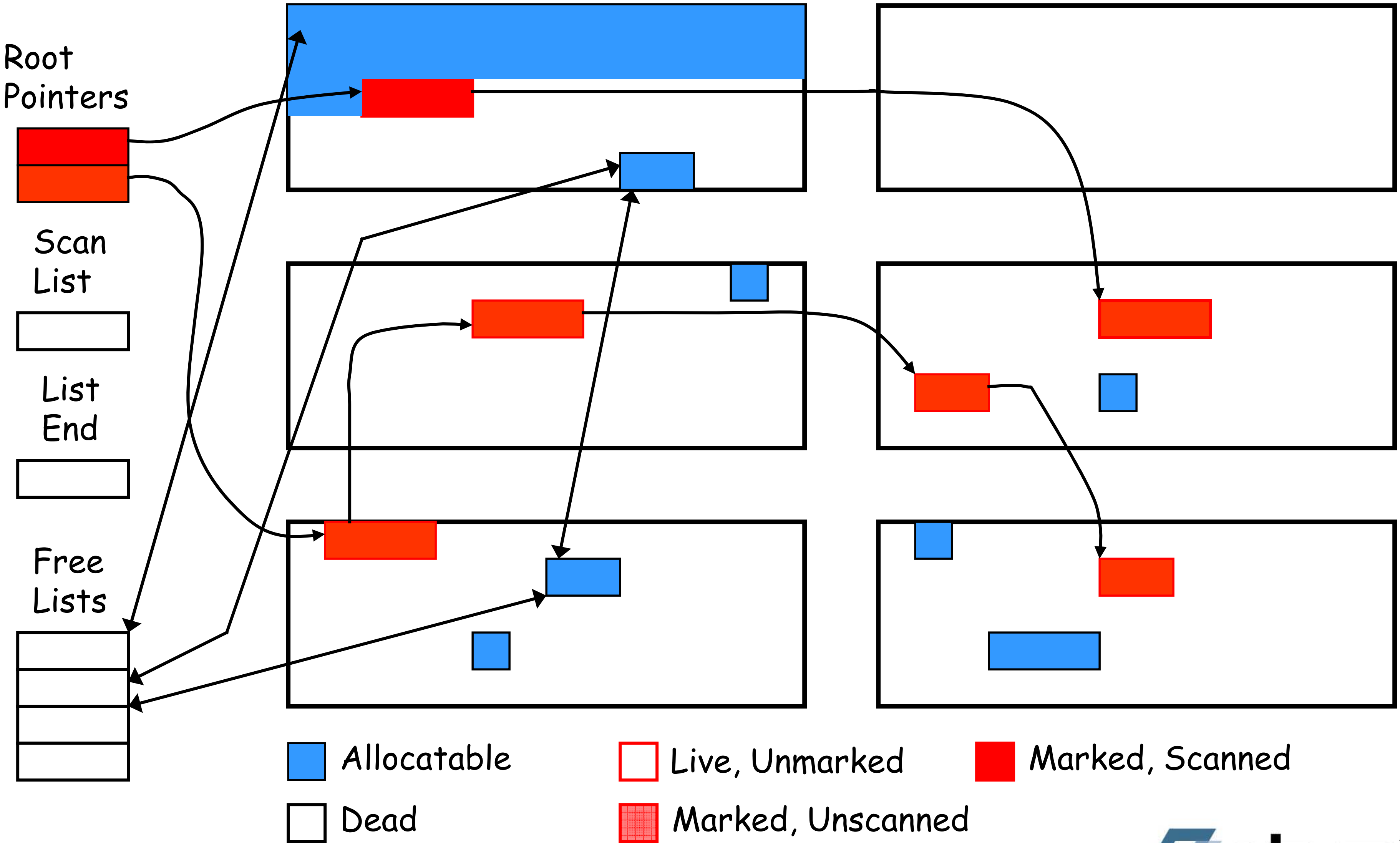
- Key attributes of Perc Ultra real-time GC for SMP Java:
  - Preemptive
  - Incremental
  - Accurate
  - Defragmenting
  - Paced
  - Parallel and Concurrent

- Traditional Java virtual machines fail in one or more of these aspects

Pacing of Garbage Collection

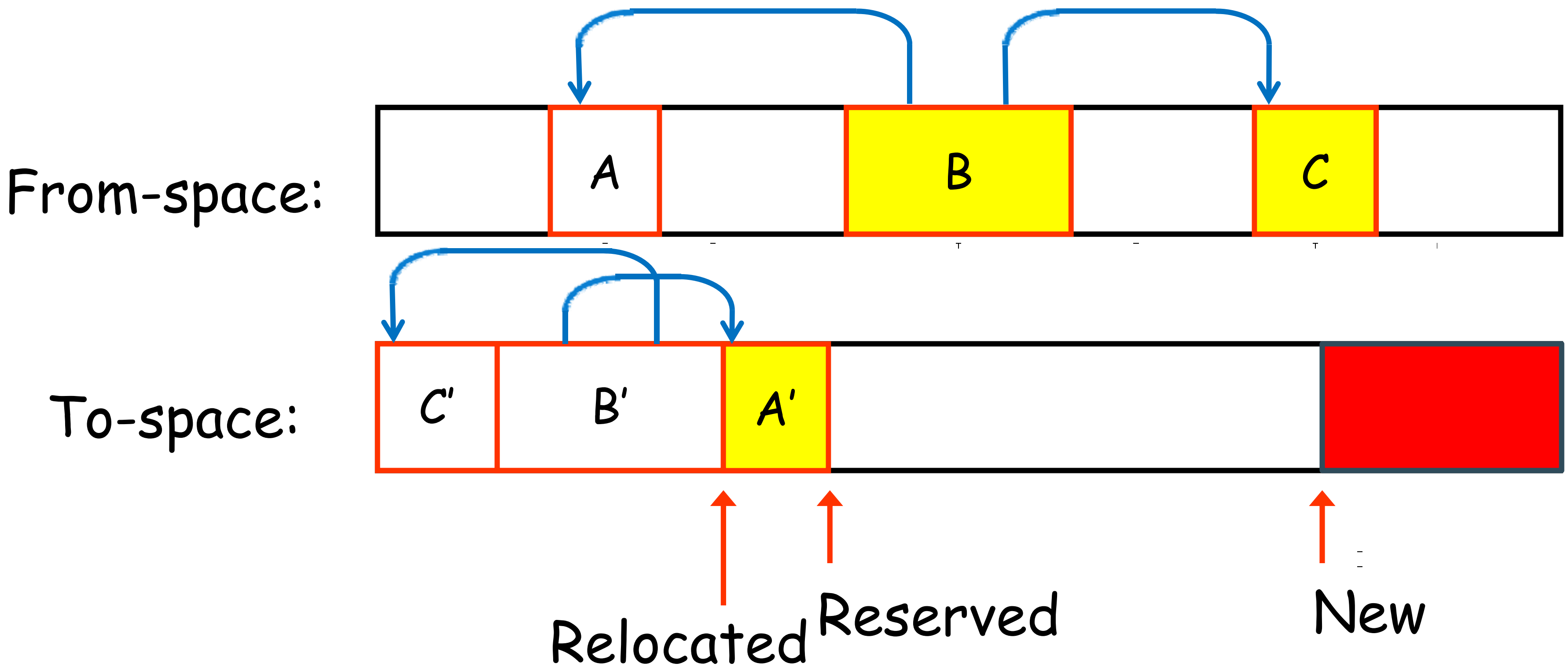


Incremental Mark and Sweep GC



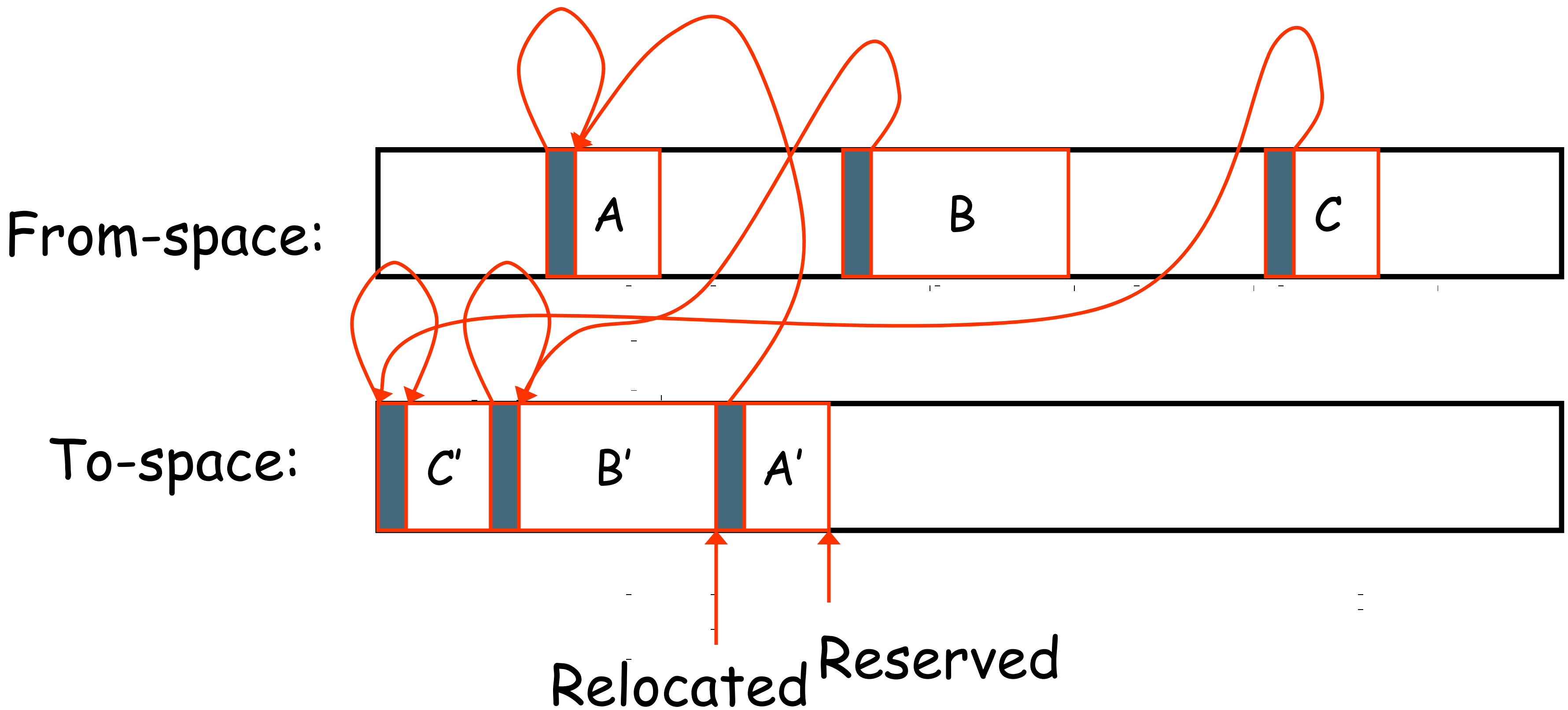
Fully Copying Garbage Collection

- Incrementally copy objects from from-space into to-space
- Redirect memory accesses between Relocated and Reserved

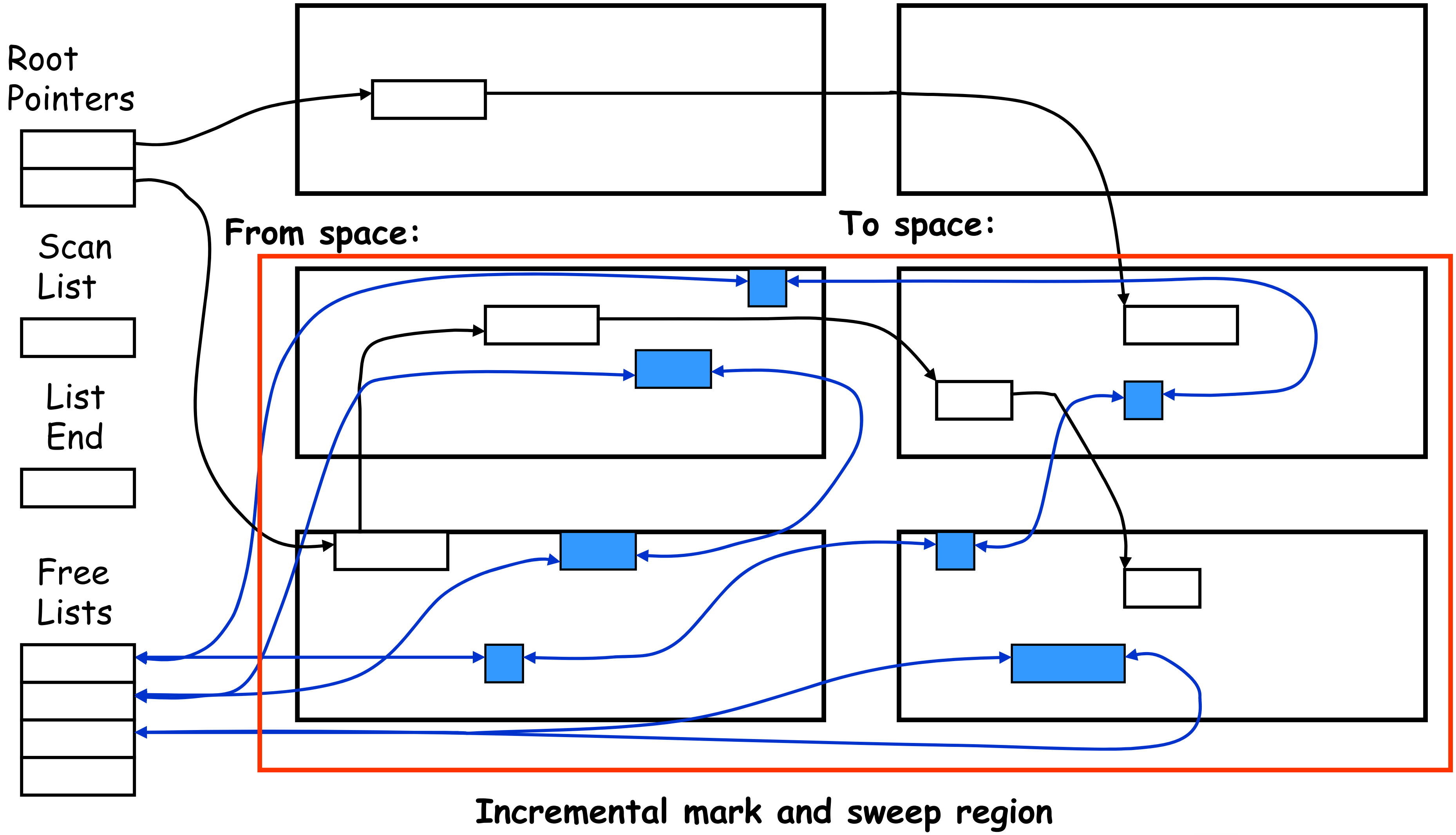


Implementation Approach

- Every object has a valid-copy pointer contained within its header



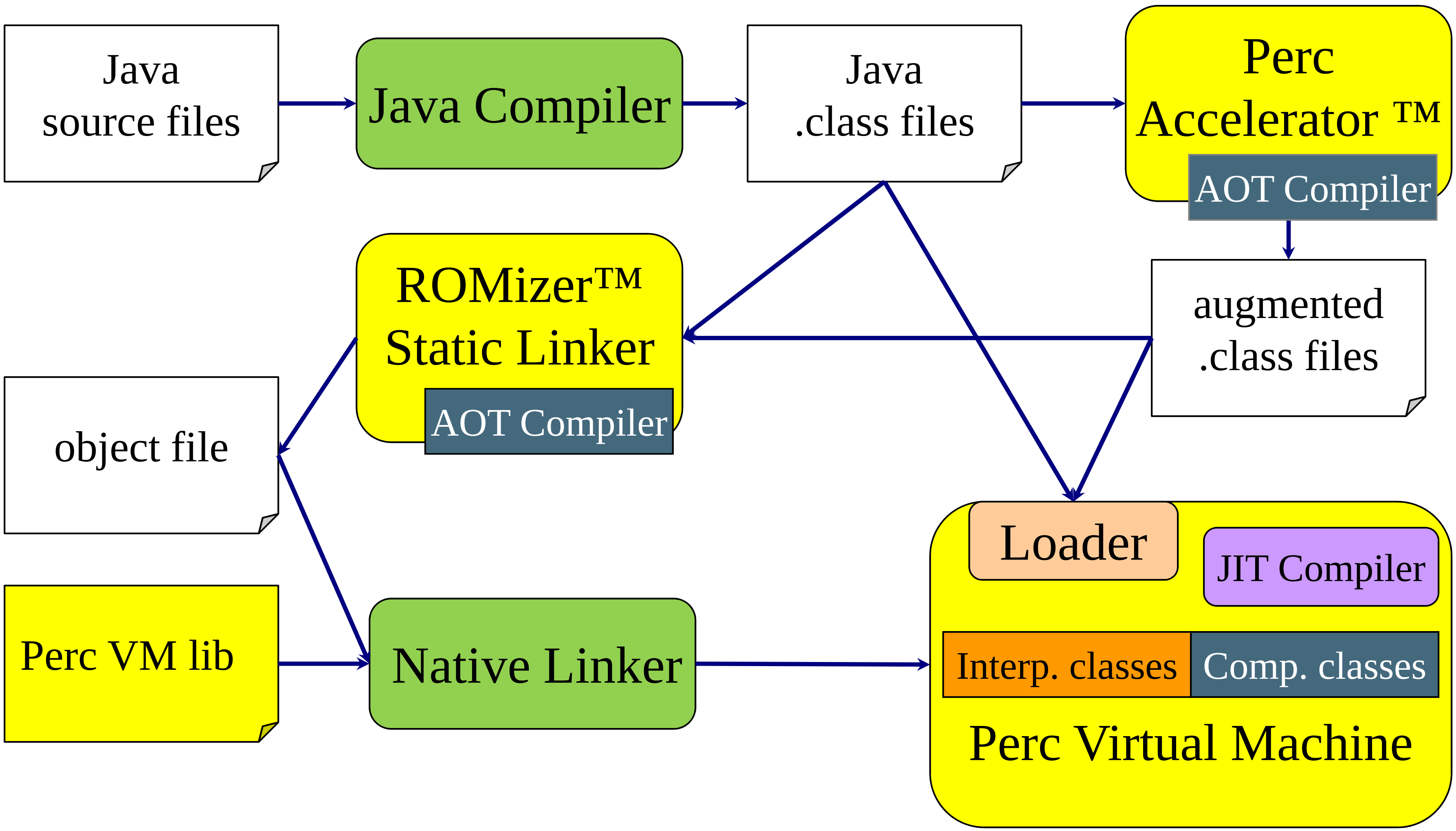
Mostly Stationary Garbage Collection



VM Management Services

- Status inquiries reveal:
  - CPU time consumed by each thread
  - Memory allocation rates
  - Total heap memory usage
  - Length of finalization queue
  - CPU time spent in garbage collection
  - Memory reclaimed by garbage collection
  - Amount of CPU time consumed at each priority level
- Management API controls:
  - Priorities for garbage collection and finalization
  - Size of the heap (enlarge and shrink)

Perc Development Flow



Anecdotal Results

- Calix: Rewrote management plane for C7 broadband loop carrier in half the time of previous C effort, while learning Java, correcting bugs in original software, and adding new functionality and scalability. (2 fold improvement)
- Lockheed Martin Aegis project: added support for "Standard Missile 6" in only 3 months. Before Java, this effort would have required at least a full year. (4 fold improvement)
- Lockheed Martin verified 3,500 requirements for a portion of Aegis Weapons System software in only 5 months. Previous expectation was 3-4 requirements per day. (9 fold improvement)
- Intel built a fault-tolerant demo of new hardware by integrating existing Java components in only 3 days. Prior similar efforts with C++ required 3 solid months! (20 fold improvement)

What have we learned in the past 15 years?

- Non-standard Java syntax is a non-starter
- Real-time Java programs should run on traditional Java VMs
- The market for traditional Java is much larger than for real-time Java – leverage traditional Java economies of scale
  - Structure “real-time Java” as libraries to avoid gratuitous incompatibility
- Traditional Java software should run on a real-time Java VM
- With proliferation of multicore processors, real-time Java needs to incorporate support for processor affinities, SMP scheduling, SMP priority inheritance

What have we learned in the past 15 years?

- Multiple real-time clocks and user-defined clocks are important
- Asynchronous transfer of control should be more general than just supporting timeouts
- With all due respect to my esteemed colleagues, programming language design by committee is not very effective
- Ten-fold improvement needed to disrupt the status quo

Some thoughts on modernizing the Perc Real-Time API

- Restructure the Perc Real-Time API entirely as a library
  - Provide an open-source library implementation to run on Standard Edition Java
  - Allow varying quality of implementation; some platforms will lack:
    - Real-time garbage collection
    - Prioritization of thread scheduling
    - Priority inheritance and priority ordered thread queues
    - Priority ceiling emulation for implementation of atomic statements
    - Precise time accounting for running threads
    - Ability to analyze worst-case execution times and approximate expected execution times
    - Precise time-driven alarms
  - Provide a tool chain to enforce restrictive styles in specific real-time contexts

Benefits of running Perc API on non-real-time VM

- Testing and debugging of functional behavior can exploit mainstream economies of scale
- Even in absence of "full compliance" with API requirements, real-time semantics can be approximated
  - timed and atomic statements, activity configuration, resource negotiation
- Real-time software running with approximate semantics on non-real-time VM will be more real-time than code that is not structured according to real-time API
- Providing an incremental (and painless) step towards disciplined real-time execution of Java for the mainstream Java market shows that community a manageable path forward
  - They will make incremental quality of implementation improvements as motivated by free-market dynamics

Enforcing style restrictions on interruptible code

- The @Responsive annotation marks code that can be timed out
- Open issue: Can we supply libraries to provide network and console I/O that would be considered @Responsive?
- The catch and finally clauses within @Responsive methods must be execution-time analyzable
- Attributes of the @Responsive annotation specify upper bounds on the responsiveness to an asynchronous signal
- A @Responsive method can only invoke other @Responsive methods with compatible responsiveness bounds
- Application developers are required to insert invocations of Perc.checkForSignal() within @Responsive code at “appropriate” intervals (consistent with declared responsiveness bounds)

Two-Phase Analysis

```
@Responsive(latency_ms = 0, latency_ns = 100000)
void method(int arg1, int arg2) throws InterruptedException {
  Perc.checkForSignal();
  for (int i = 0; i < 10000; i++) {
    arg1 += arg2;
  }
  Perc.checkForSignal();
}
```

Two-Phase Analysis

```
@Responsive(latency_ms = 0, latency_ns = 100000)
void method(int arg1, int arg2) throws InterruptedException {
   Perc.checkForSignal();
   for (int i = 0; i < 10000; i++) {
      arg1 += arg2
   }
   Perc.checkFor
}
```

Phase 1 (target independent) analysis assures

1. **That the first and last statements in this @Responsive method invoke checkForSignal()**
2. **That every path between the two checkForSignal() invocations is execution-time analyzable**

Two-Phase Analysis

```
@Responsive(latency_ms = 0, latency_ns = 100000)
void method(int arg
   Perc.checkForSig
   for (int i = 0; i < 10
      arg1 += arg2;
   }
   Perc.checkForSig
}
```

Phase 2 (target dependent) analysis assures

1. **That the maximum execution time between the first and last invocations of checkForSignal() is less than or equal to 100 μs**

Representing atomic statements

- Add the @Atomic annotation to a synchronized method to denote that the object uses atomic PCP locking
  - The body of @Atomic synchronized method must be WCET analyzable
  - If one method is @Atomic synchronized, then all synchronized methods must be marked @Atomic
  - For any class that has @Atomic synchronizers, all synchronized methods in super- and sub-classes must be @Atomic synchronized
  - If a class has @Atomic synchronizers, then the class must implement the PriorityCeilingEmulation interface (a refinement from the paper), which has two methods:
    - int msCeiling() and int nsCeiling()
    - These are invoked by a real-time virtual machine each time an @Atomic synchronized method is entered
    - A non-real-time virtual machine will not enforce priority ceilings

Implementing the timed statement

- The Perc.timed() library service expects two arguments:
  - A RelativeTime object that is associated with the system's ExecutionTime clock, and
  - An Interruptible object – Interruptible implements Runnable and adds the @Responsive annotation to its run() method

Summary

- Different versions of real-time Java offer different benefits to different audiences
- Having failed to achieve widespread market acceptance, many anticipated benefits of RTSJ "standardization" are not being realized
  - Multiple suppliers of tool chains, compilers, libraries
  - Abundance of off-the-shelf reusable software components
  - Widespread adoption allowing multiple end users to share the costs of technology development
  - Free market competition to drive innovation and product improvement
- An alternative approach to real-time Java may achieve better acceptance and deliver greater benefits, with or without standardization