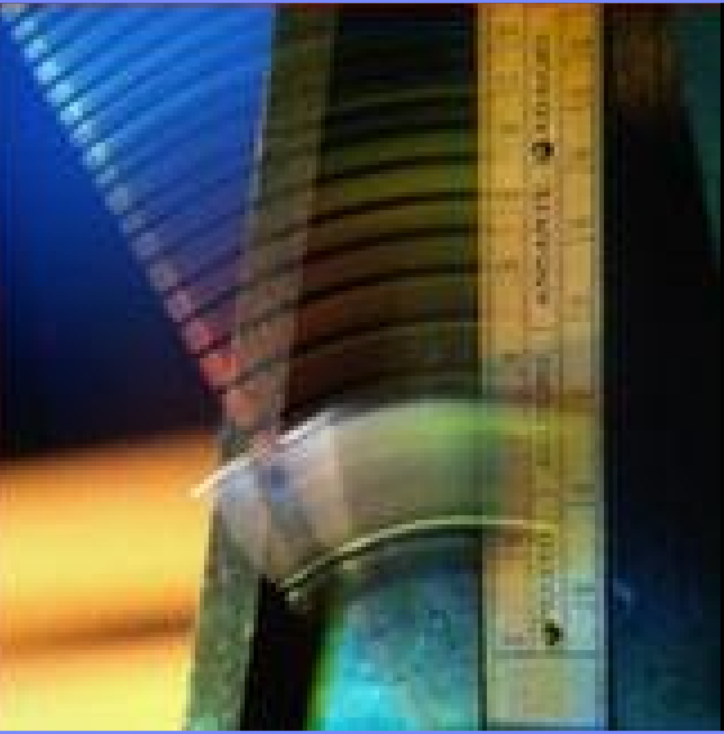


Real-time Music Synthesis in Java with the Metronome Garbage Collector

Joshua Auerbach
IBM Research

David F. Bacon
IBM Research

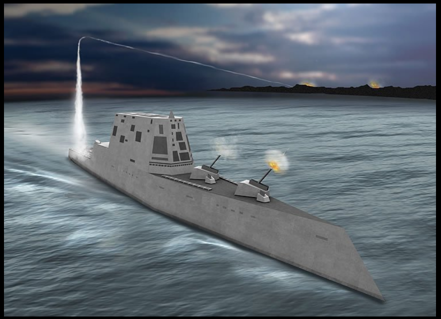
Florian Bomers
Bome Software

Perry Cheng
IBM Research

It's a Real-time World

30 MLOC
1ms

80 MLOC
10us - 100ms



100 MLOC
10us - 10ms

50 MLOC
5ms

60 MLOC
10ms

Why Real-Time Java??

- Traditional methodologies
  - Highly restricted programming models with verifiable properties
  - And/Or low-level languages for explicit control
  - "ad-hoc low-level methods with validation by simulation and prototyping"
- But: these methodologies do not scale
  - Halting problem
  - Low productivity (low-level languages, hand-optimization)
- And: complexity of real-time systems are growing extremely fast
  - From isolated devices to integrated multi-level networked systems
  - Traditional methodologies break down

Why Not Real-time Java?

- Garbage Collection
  - Non-deterministic pauses from 100 ms to 1 second
  - Requirement for real-time behavior is 100 us to 10 ms
- Dynamic (JIT) Compilation
  - Unpredictable interruptions
  - Large variation in speed (10x)
- Dynamic Loading and Resolution
  - Semantics determined by run-time ordering
- Optimization technology optimizes average case
  - Thin locks, speculative in-lining, value prediction, etc.
  - Sometimes cause non-deterministic slowdowns
- …

Demo: Synthesizer on Non-RT Java

Garbage Collection: Motivation & History

- Invented in 1960 by McCarthy for Lisp
  - Objects are reclaimed automatically when no longer in use
- Huge advantages:
  - No bugs due to freeing of memory still in use
  - Simpler interfaces since lifetime management not required
  - Type safety
  - Security
- Used in:
  - Lisp, Smalltalk, ML, Java, C#, Lua, Python, …
- But not in:
  - C, C++, Pascal, Ada, Fortran, …

Previous Partial Solutions to GC Problems

- Two main types
  - Generational Collection (Ungar)
  - Incremental Collection (Dijkstra, Yuasa)
- Many pathologies:
  - High nursery survival rate (1ms -> 40ms collection)
  - Atomic root snapshot (no thread scaling)
  - Unpredictable termination ("last" pointer problem, 100s of ms)
  - Inability to handle large objects in real-time
  - Uneven utilization (driven by allocation or pointer access)
  - Subject to fragmentation
  - High (sometimes unbounded) memory overhead
  - Failure to incrementalize weak reference, finalizers, strings, …

Java for Real-time: Current Practice

- Avoid allocation after setup
  - Low-level programming, vulnerable to allocation by libraries
- Allocate from object pools
  - Only works for homogeneous objects, suffers from "free" bugs
- Use Scoped memory constructs of RTSJ
  - Manual, suffers from unpredictable run-time exceptions
- Use a generational collector
  - Puts off the inevitable, slow when survival rate is high
- Use an incremental collector
  - Often works but subject to numerous failure modes
- Use reference counting (automatic or manual)
  - Does not collect cycles (at least not predictably)

Metronome: RT GC without Pathologies

- All phases of collection incrementalized
- All collector work deferrable to next desired quantum
- Scheduling regular and guaranteed by metric (MMU)
- Threads processed independently
- Internal fragmentation bounded (parameter, use 1/8)
- External fragmentation prevented (on-demand compact)
- Large objects broken into pieces (“arraylets”)
- Constant-time allocation
- Single-quantum termination
- Simple and provable feasibility: live memory, allocation rate

- Result: application allocating 10 MB/s, with 1000 threads, 1 GB heap, 10 MB objects, and many phase changes can run for weeks with *zero* violations, 2ms worst-case latency

IBM Real-Time Java (J9 Virtual Machine)

- Metronome Real-time Garbage Collection
  - Provides real-time without changing the programming model
- RTSJ (Real-Time Specification for Java) – existing standard
  - Scheduling
  - Scopes
- Ahead-of-Time Compilation
  - Ahead-of-time (AOT) compilation and JXE Linking
  - Removes JIT non-determinism, allows code to be moved into ROM
  - Class pre-loading

- Real-time Linux
  - Maximize use of existing patches; stabilize; add needed features
  - Contribute to open-source community

- Status
  - Shipping product since 8/06, over $100M contract revenue
  - In use in telecom, military, and financial industries

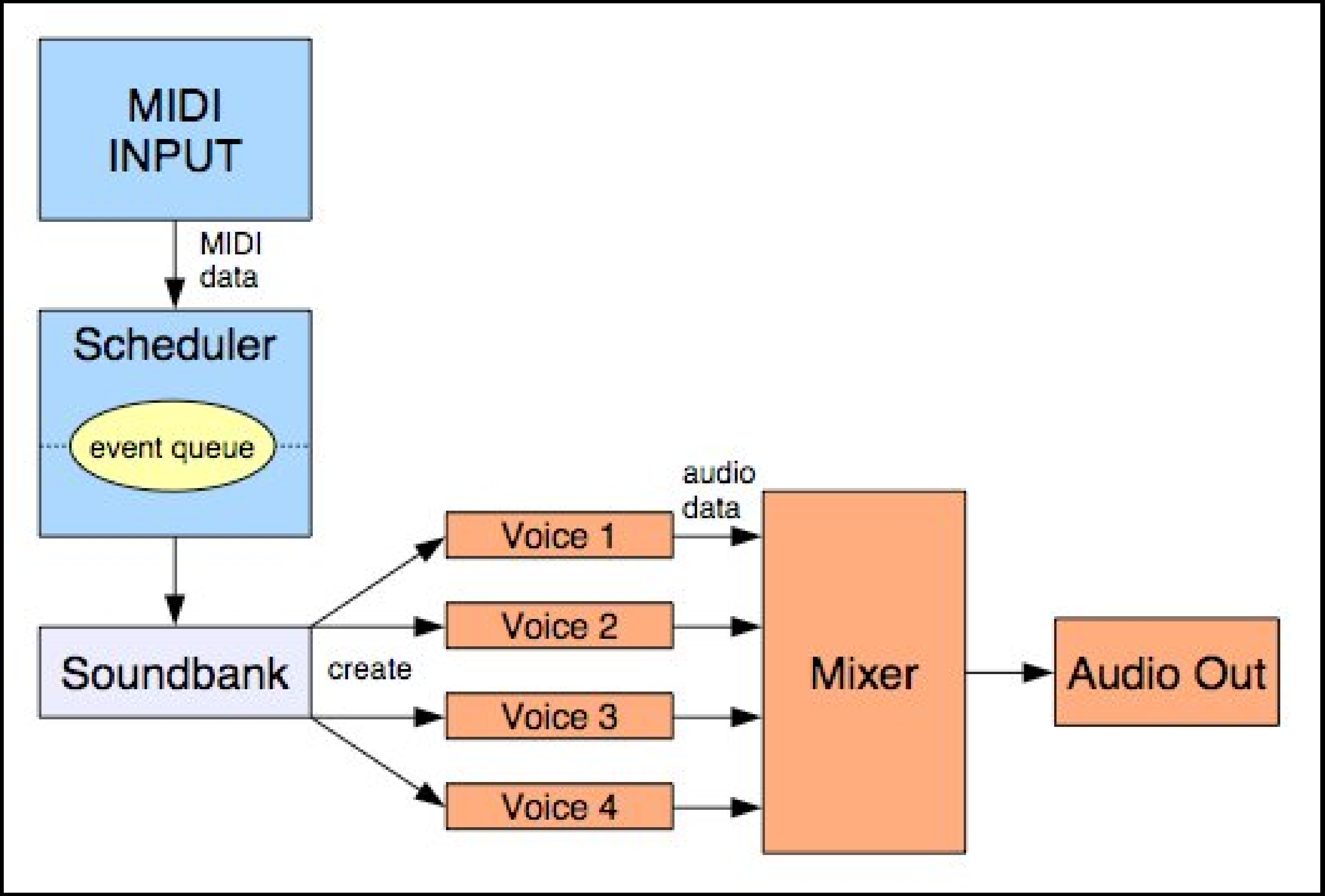
Harmonicon Java Synthesizer

Java for Real-time MIDI Synthesis

- Typical real-time music application
- Requires max 5-10ms latency, 1-2ms jitter

- Harmonicon: all-Java synthesizer
  - SoundFont-2 wavetable synthesizer
  - 64-bit sample precision
  - Arbitrary polyphony (500 voices on current hardware)
  - Concurrent (multiprocessor) rendering
  - Modular, flexible, high-level design
  - Extensive use of object-orientation and dynamic allocation

Harmonicon Synthesizer Architecture



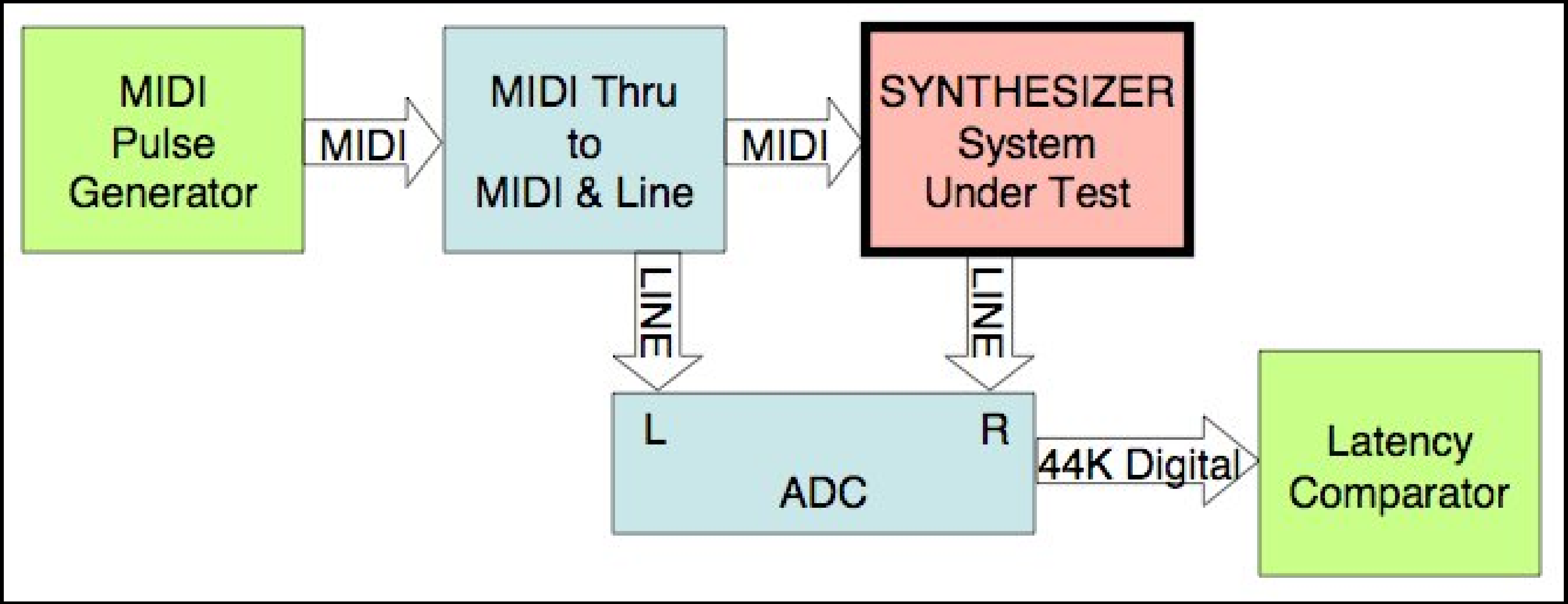
Experimental Evaluation

- Experimental Environment
  - Dual Opteron 250 CPUs (2.4 GHz, 1MB L2 cache)
  - M-Audio 2496 sound card (MIDI in, RCA out)
  - IBM Real-time Linux (RHEL 4 U2, 2.6.16 based)
  - IBM Websphere Real-time Java V1 SR1
  - Debussy's *Doctor Gradus*, Piano 1 instrument, max polyphony 13
  - 44.1 KHz 32-bit stereo
  - Additional 8 MB/s memory load thread executing at all times
- Measurements
  - Evaluation of base MIDI latency/jitter
  - Absolute measurements vs. Kurzweil K2000R
  - Comparison of 4 garbage collectors

Demo: Synthesis with RT Java

1ms buffer,
AOT compilation,
class preloading

Absolute Latency Measurements



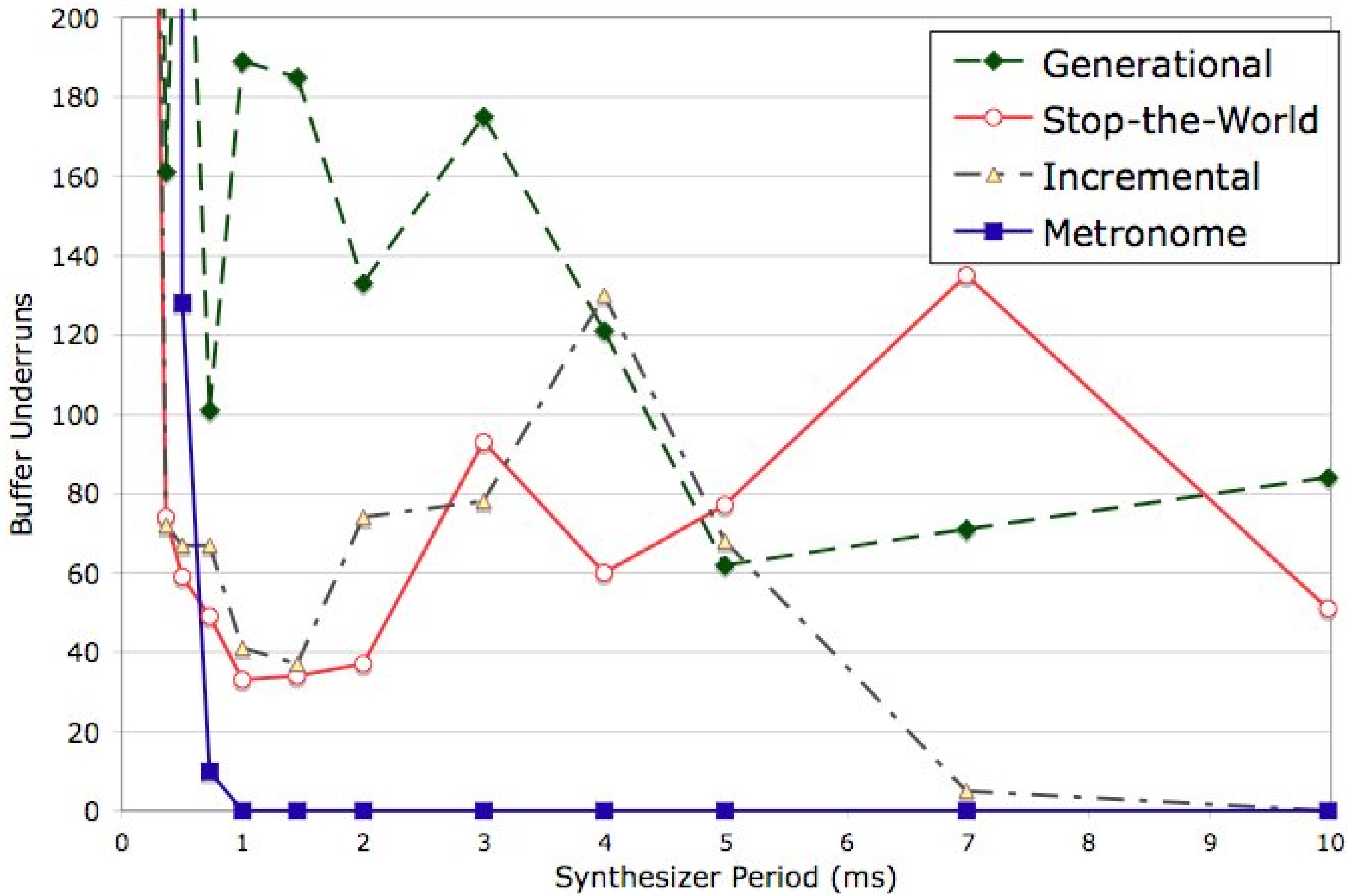
Base MIDI Latency (milliseconds)

	Min	Mean	Max	StDev
ALSA via C	0.340	0.347	0.362	0.011
Java Sound	0.385	1.455	3.197	0.701
ALSA via Java/JNI	0.385	0.406	0.430	0.011

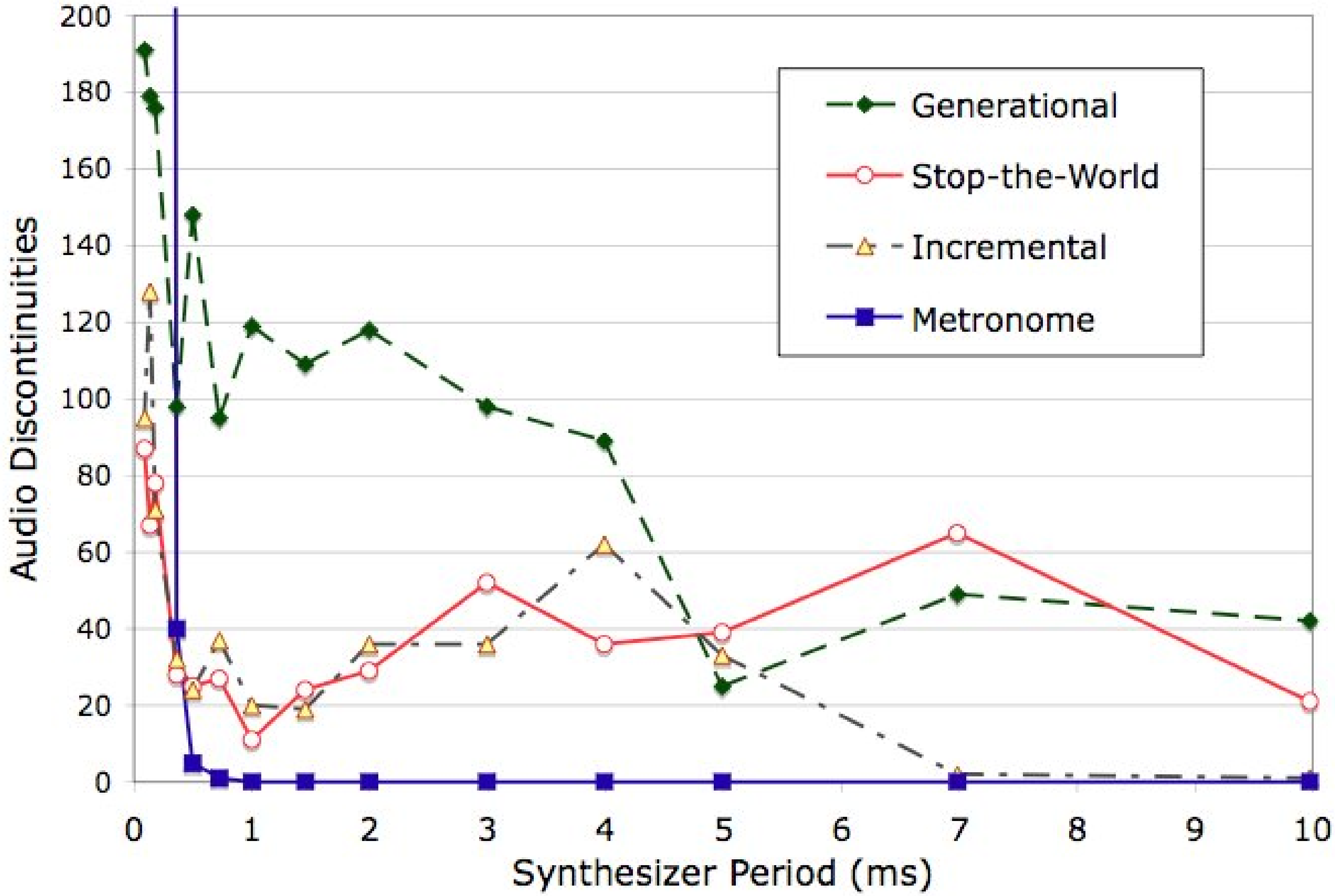
Harmonicon vs Kurzweil K2000R

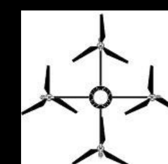
	Min	Mean	Max	StDev
Kurzweil K2000R	2.925	3.909	4.897	0.570
Harmonicon (1ms buffer)	4.240	4.959	5.736	0.317
Harmonicon (365us buffer, no GC)	2.947	3.120	3.310	0.109

GC Comparisons: ALSA Underruns



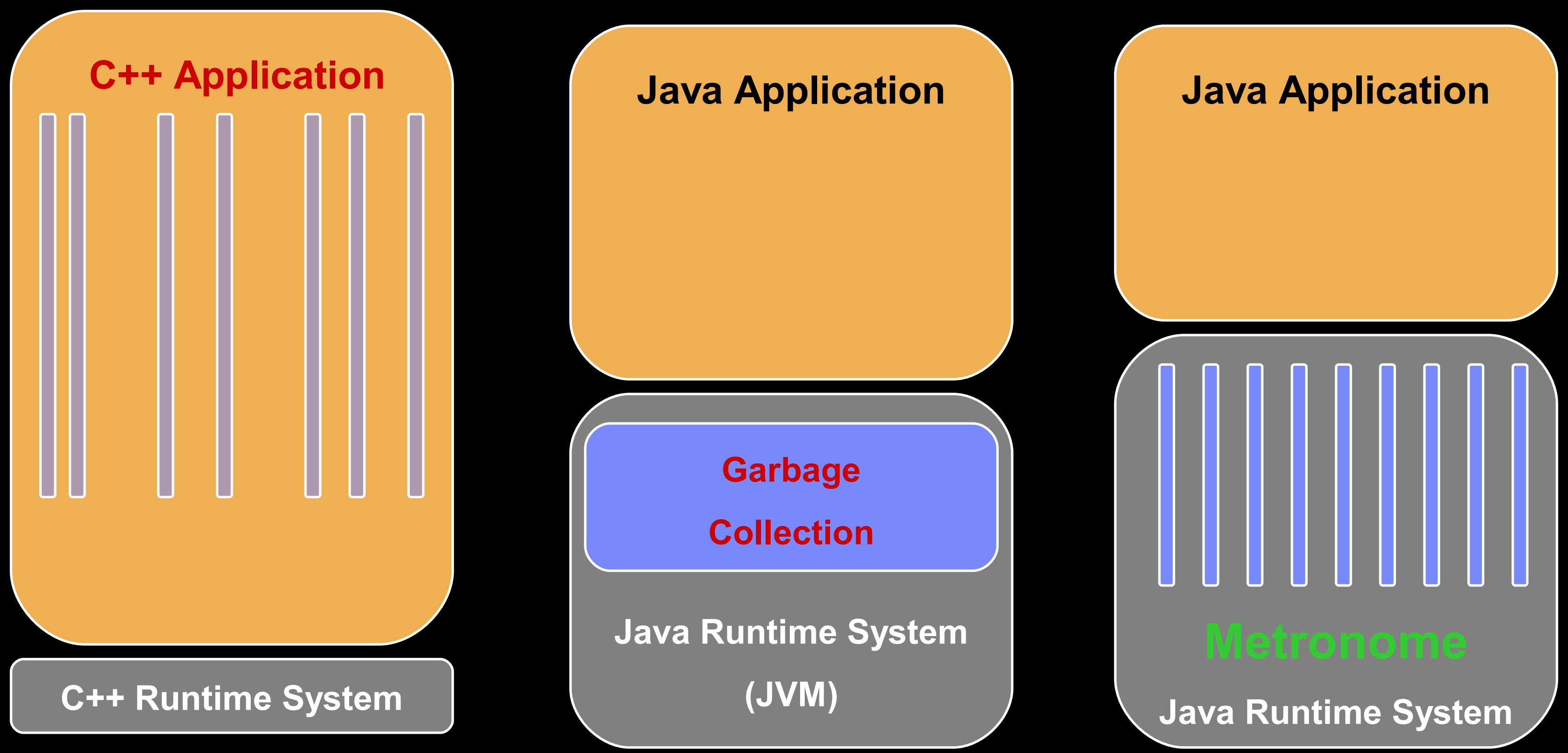
GC Comparisons: Audio Discontinuities





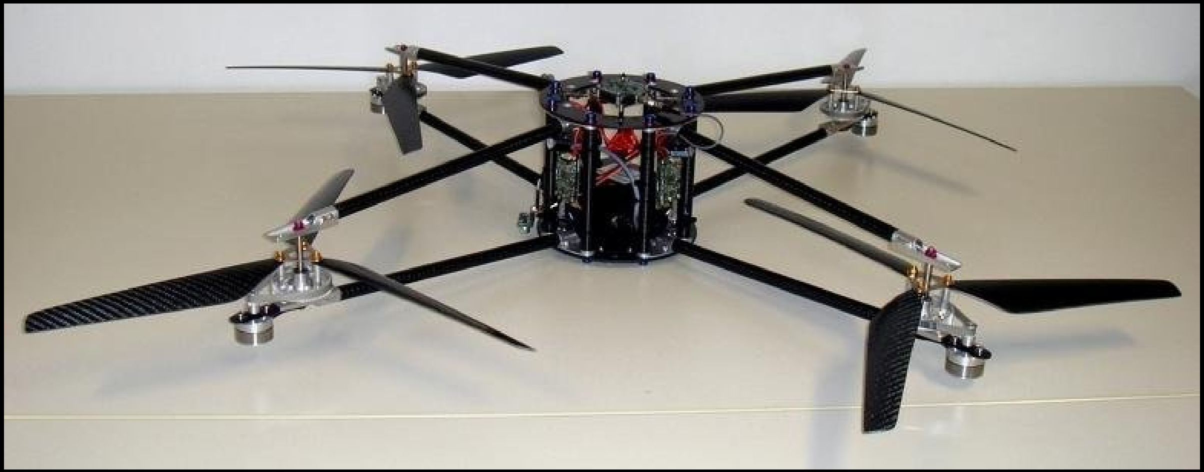
http://www.research.ibm.com/metronome

Metronome: *Transparent* Real-time Java



Manual, Unsafe	Automatic, Safe	Automatic, Safe
Predictable	Unpredictable	Predictable

Testbed 1: Autonomous Quad-rotor Helicopter



- Single-helicopter control
  - Fully custom design
  - Completely Java-based
  - 3 ms control loop period
- Key Goals
  - Validate with most critical physical control systems
  - Time-portable real-time software
  - Compositional real-time
    - Dynamic upload of other RT systems

[with Christoph Kirsch, University of Salzburg]