Aonix

Der Einsatz von Java für "harte Echtzeit"-Anforderungen

Frank Lippert, Aonix GmbH

- Vor 10 oder 15 Jahren war es noch möglich:
  - Die gesamte Software für jedes Embedded Gerät konnte von Grund auf selbst entwickelt werden
  - Mit einem Team von einem oder zwei Entwicklern
  - in sechs bis zwölf Monaten

- Nichts davon ist heute noch machbar:
  - Moderne eingebettete Systeme bestehen aus Hunderttausenden oder gar Millionen Zeilen Code
  - Sogar inkrementelle neue Funktionen, die mit jedem Produktzyklus kommen sind mehr als das Entwicklerteam innerhalb von zwölf Monaten bewältigen kann

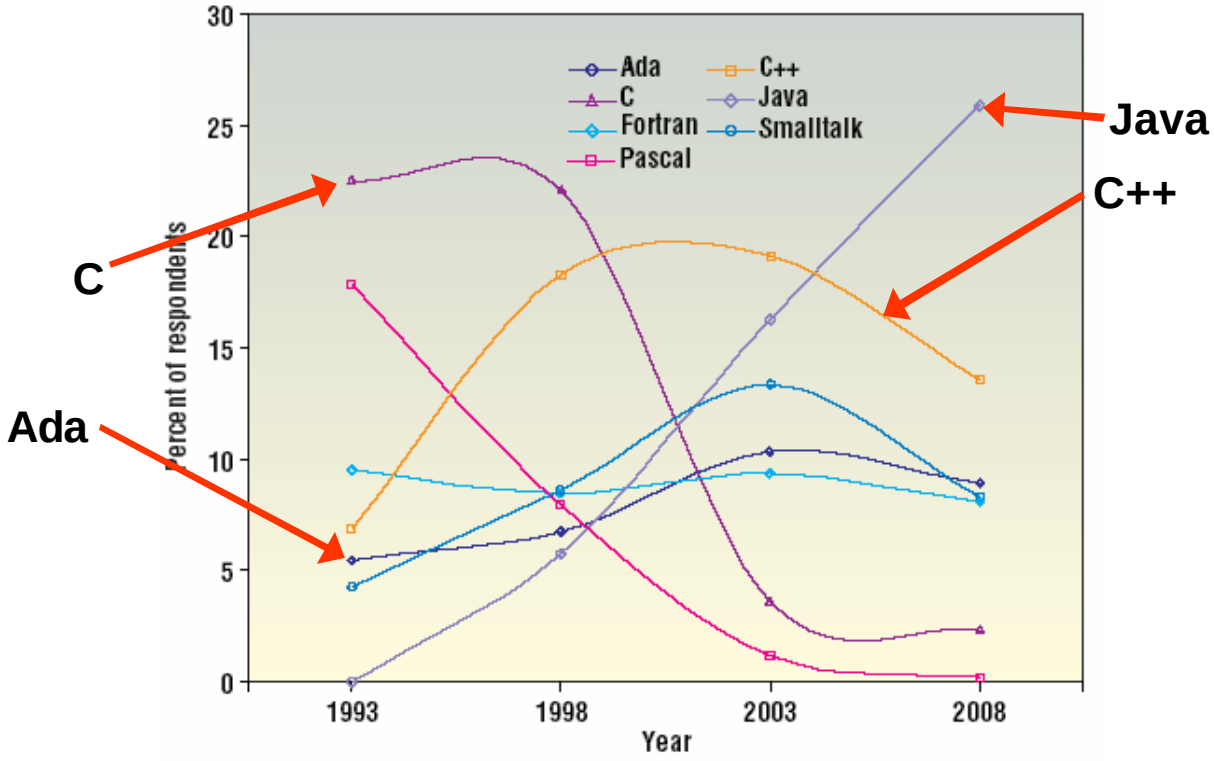
- Heute suchen Softwareentwickler im Embedded-bereich nach
  - Einfacheren und günstigeren Wegen zur Softwarewartung
    - Wartung bedeutet Portierung auf neue Plattformen, Fehlerkorrekturen, Performanceverbesserungen, das inkrementelle Hinzufügen neuer Funktionen und das Kombinieren einer Software mit anderen, komplementären Systemen.
  - Leichtere Integration von unabhängig entwickelten Komponenten
  - Höhere Entwicklerproduktivität während der Implementierung neuer Fähigkeiten

- Viele Experimente zeigen, dass Java-Entwickler mindestens doppelt so produktiv sind wie C und C++ Entwickler. - Warum?
  - Strengere Typprüfung reduziert Programmierfehler
  - Objektorientierung vermindert Störungen zwischen unabhängig entwickelten Komponenten
  - Erzwungene Ausnahmebehandlung reduziert Programmierfehler
  - Automatische Speicherbereinigung vereinfacht das Speichermanagement und lässt weniger Fehler zu
  - Die einfachere Sprache reduziert Missverständnisse beim Programmieren
  - Die erhöhte Portabilität erlaubt es Programmieren die Sprache zu meistern anstatt einer bestimmten Tool-kette und RTOS-Diensten

- Viele Entwickler sind in Java 5 bis 10 mal produktiver bei der Integration unabhängig entwickelter Komponenten als mit C oder C++. Warum?
  - Verbesserte Portabilität macht das "Portieren" zu jeder neuen Hardware oder OS unnötig.
  - Objektorientierte Abstraktionen vermindern Namenskollisionen und andere Störungen zwischen Komponenten
  - Automatische Speicherbereinigung vereinfacht Integration weil keine Verantwortung für das Wiedergewinnung von Speicher, der durch eine Komponente benutzt, aber durch eine andere angefordert wurde, zugewiesen werden muss
  - Ergebnis: Komponenten arbeiten "out of the box"

- Intel-Projekt:
  ausfallsicherer, verteilter Java-Demonstrator in 3 Tagen! Intel's Aussage: “Es hätte dreier Mannmonate bedurft um diese Demo ohne Java zu entwickeln [d.h. VxWorks und C]”
- Kundenerfahrungsbericht von Calix:

  Verglichen mit C erhöhte die Echtzeit-VM PERC die Entwicklerproduktivität um den Faktor zwei (einschließlich der Zeit für das Erlernen von Java!), und reduzierte die Codegröße auf ein Fünftel.
- Nortel:
  Java-Entwickler sind produktiver und ihr Code problemloser als C++-Entwicklern.



Trends in der Benutzung der 8 populärsten Programmiersprachen

Ergebnis: warum wird Java so gu

- Einfachere Entwicklung flexiblerer Software-systeme (Flexibilität ermöglicht Langlebigkeit)
- Enorme Einsparungen bei der Softwareintegration und -wartung (Der Anteil der Wartungskosten wächst proportional zu der Grösse und Komplexität von Systemen)
- Es existiert eine Fülle an Standardsoftware-komponenten und Entwicklungstools
- Kompetenten Java Entwickler sind ausreichend im Stellenmarkt vorhanden

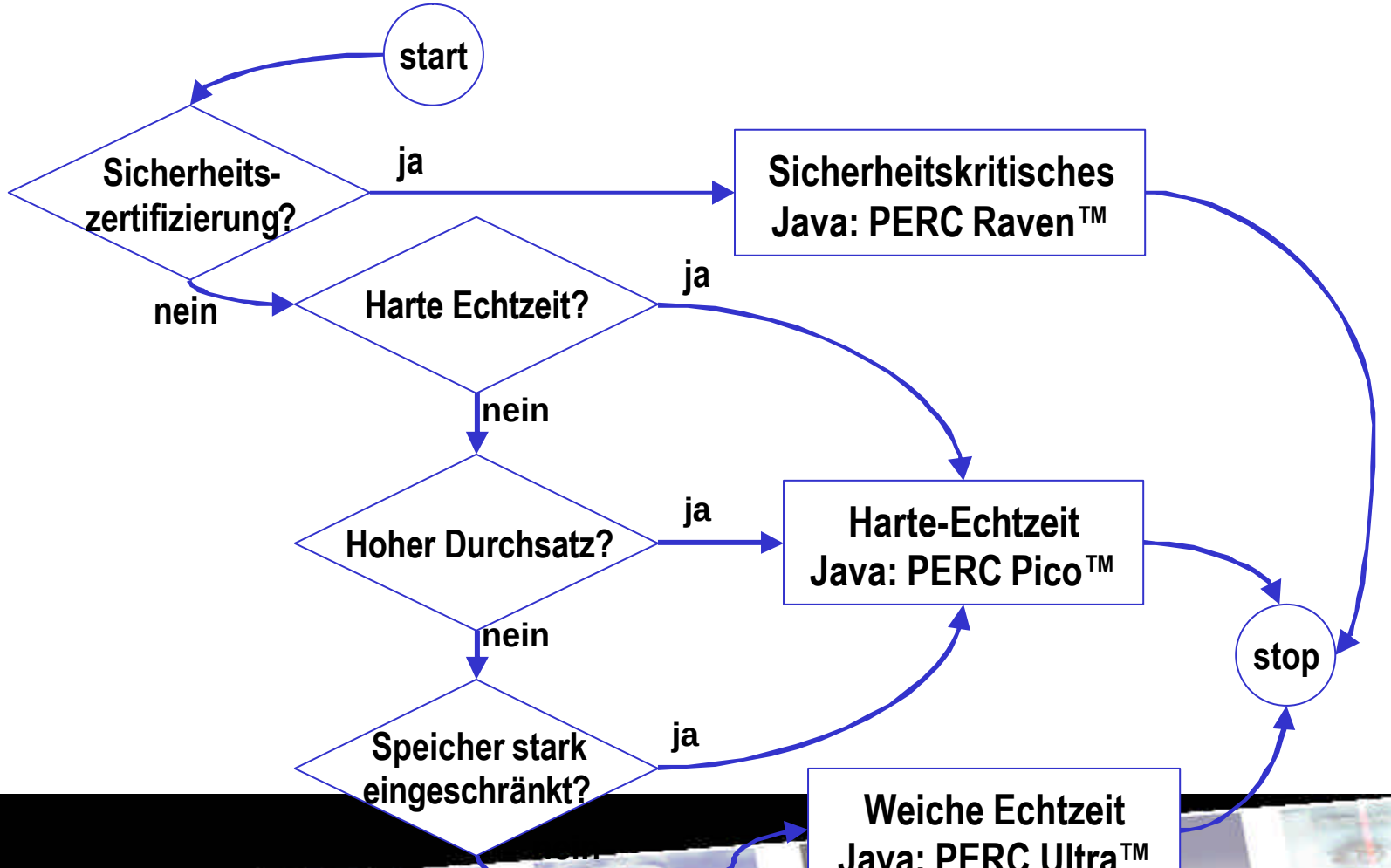
Zwei Arten von Echtzeit

- Weiche Echtzeit:
  - Resourcenverbrauch wird durch empirische Techniken bestimmt
  - Ressourcenbudgets werden mittels Heuristiken durchgesetzt

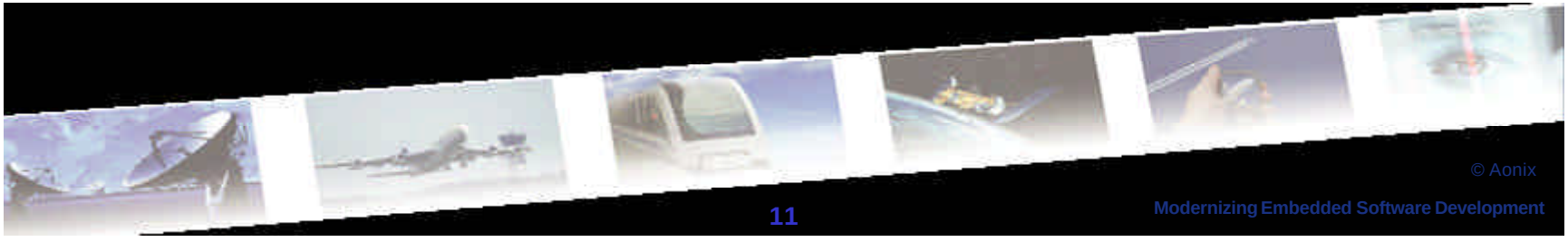
- Harte Echtzeit:
  - Absolute Grenzen von worst-case Ressourcenverbrauch durch analytische Methoden bestimmt
  - Deterministische Algorithmen erzwingen Ressourcenbudgets

Das richtige Tool auswählen

Productivity In Real–Time



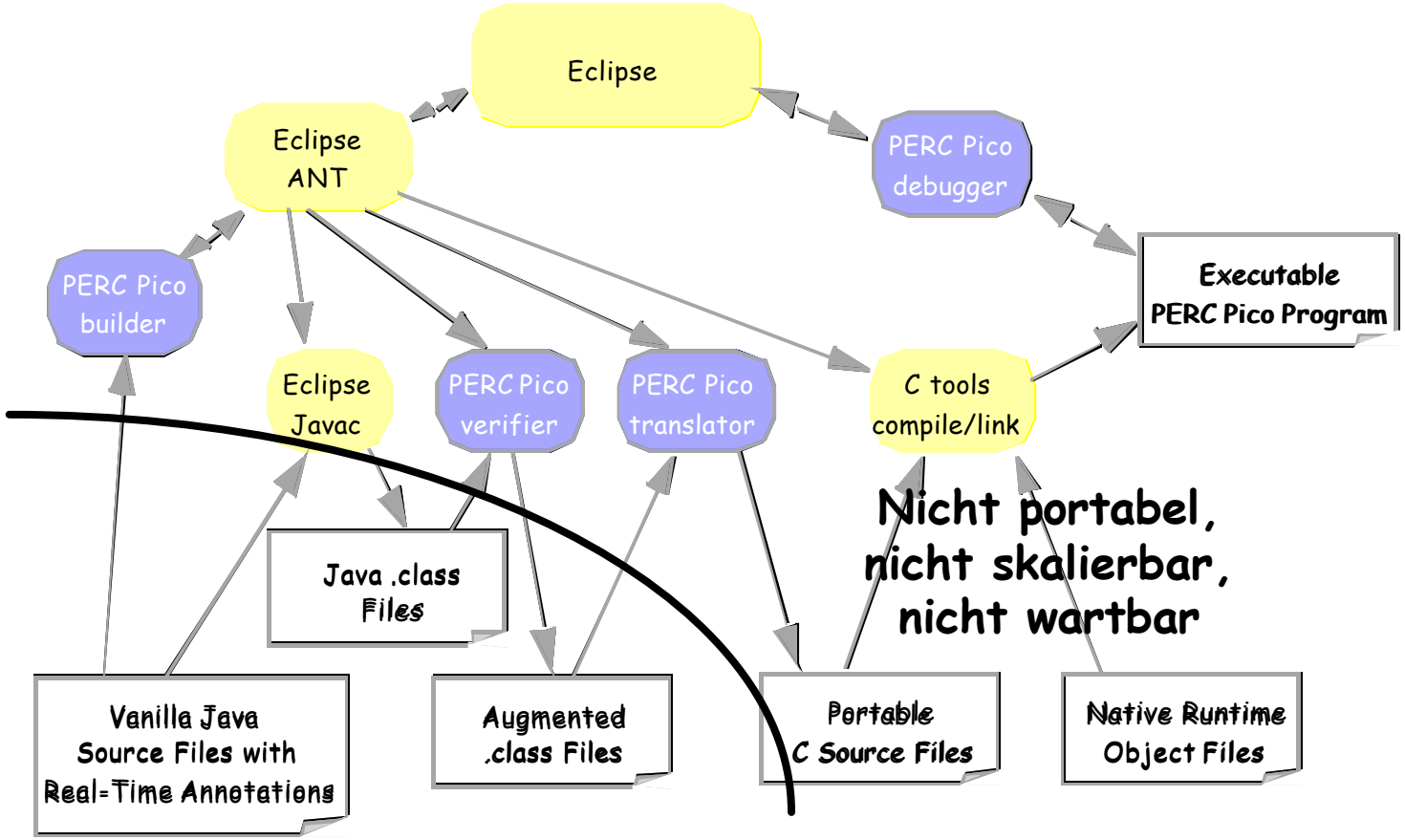
- Standardbibliotheken sind exakt definiert bezüglich worst-case CPU-Zeit und Speicherverbrauch
- Die Programmiersprache hat syntaktische Mittel worst-case CPU-Zeit und Speicherverbrauch zu charakterisieren
- Geräte-E/A und Echtzeitrandbedingungen können abgedeckt werden
- Es gibt Werkzeuge, die bei der Analyse von CPU-Zeit und Speicheranforderungen helfen



- Keine automatische Speicherbereinigung (Garbage Collector)
  - Stattdessen Benutzung sicherer scope-basierter Speicheranforderung
- Standarduntermenge von RTSJ and J2SE (siehe JSR-302)
  - Verbesserte Portabilität
  - Verbesserte Effizienz (vermeidet kostspielige Features die nicht relevant sind für die harte-Echtzeit-Nische)
- Erweiterungen der Standardbibliothek
  - Geräte-E/A und Interrupthandling
  - Passive und aktive Echtzeituhr
- Java 5.0 Standard-Metadaten-Annotationen machen Absichten der Programmierer klar und ermöglichen modulares Zusammen-stellen von Komponenten
  - Weniger Programmier- und Integrationsfehler, erhöhte Verlässlichkeit
  - Höhere Performance und geringerer Speicherverbrauch
  - Geradlinige und verlässliche Integration von Komponenten

Die PERC Pico 1.1 Toolkette

Productivity In Real–Time



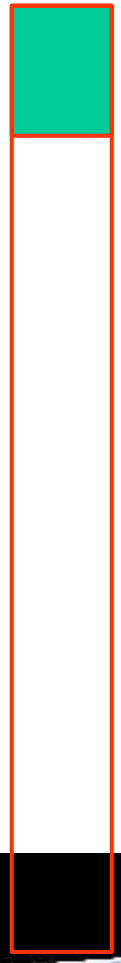
Beispielannotationen für Re


```
[1]  import com.percpico.util.sc.StaticAnalyzable;
[2]  import com.percpico.util.sc.StaticLimit;
[3]
[4]  public class BubbleSort {
[5]      public enum AnalysisModes {
[6]        UNBOUNDED,          // can't analyze the most general case
[7]        PRESORTED,          // only one element out of order
[8]        UNSORTED,           // no ordering precondition
[9]      }
[10]     @StaticAnalyzable(
[11]         enforce_memory_analysis = { true, true, true },
[12]         enforce_time_analysis = { false, true, true },
[13]         modes = AnalysisModes.class, parameters = {"", "n", "n"})
[14]     public static void sort(@Scoped int a[]) {
[15]         int i, j, k, t;
[16]         int len = a.length;
[17]         boolean sorted = false;
```

Beispielannotationen für Re

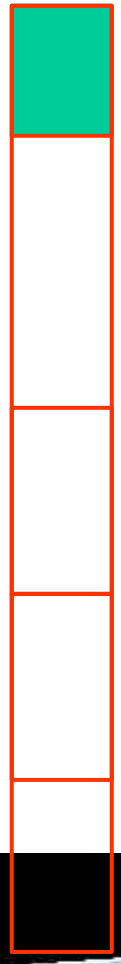

```
[18]      for (i = 0, k = len; !sorted && (i < len); i++) {
[19]          assert StaticLimit.IterationBound(AnalysisModes.UNSORTED, “n”);
[20]          assert StaticLimit.IterationBound(AnalysisModes.PRESORTED, “2”);
[21]          k--;
[22]          sorted = true;// assume array is sorted
[23]          for (j = 0; j < k; j++) {
[24]              assert StaticLimit.IterationBound(
[25]                  AnalysisModes.UNSORTED, “((n-1)* n)/2)”);
[26]              assert StaticLimit.IterationBound(
[27]                  AnalysisModes.PRESORTED, “(n-1) + (n-2)”);
[28]              if (a[i] < a[j]) {
[29]              assert StaticLimit.IterationBound(
[30]                  AnalysisModes.PRESORTED, “n-1”);
[31]              t = a[i]; a[i] = a[j]; a[j] = t; sorted = false;
[32]          }
[33]      }
[34] }
```

Das Thread Stack Modell



Anfangs repräsentiert der Laufzeitstack des Hauptthreads (wächst nach unten) den gesamten nicht-*immortal* Speicher

Das Thread Stack Modell



Der Hauptthread erzeugt eventuell zusätzliche Threads, wobei er Teile seines eigenen Stacks reserviert um den Stackspeicher der erzeugten Threads zu repräsentieren

Jeder der erzeugten Threads kann wiederum seinen Stack aufteilen um „Enkel"-Threads zu erzeugen

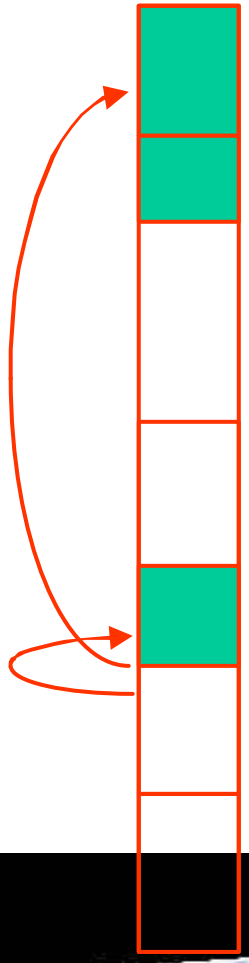
Speicher für den Stack des dritten erzeugten Threads

Speicher für den Stack des zweiten erzeugten Threads

Speicher für den Stack des dritten erzeugten Threads

Das Thread Stack Modell

Productivity In Real–Time



Aber Objekte im Stack des Elternthreads unterhalb des Punkts von dem aus der Kindthread erzeugt wurde sind nicht für den Kindthread sichtbar. Und äußere Objekte können keine Objekte sehen die in einem weiter innen liegenden Bereich allokiert wurden.

Ein Kindthread kann Bereichsobjekte im Stack des Eltern-threads oberhalb des Punktes, von dem aus der Kindthread erzeugt wurde sehen.

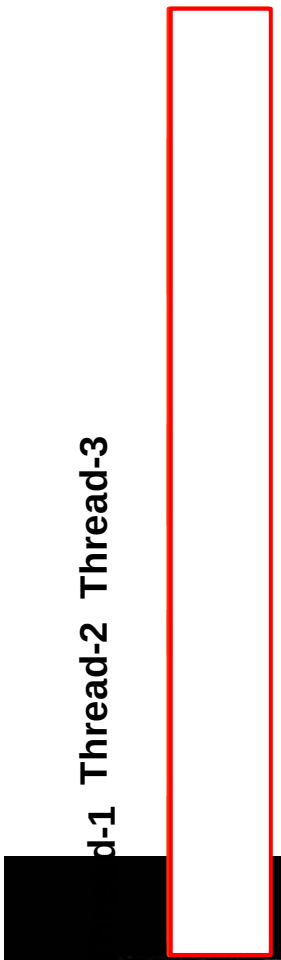
Einzelne Threads füllen ihre Laufzeitstacks nach Bedarf

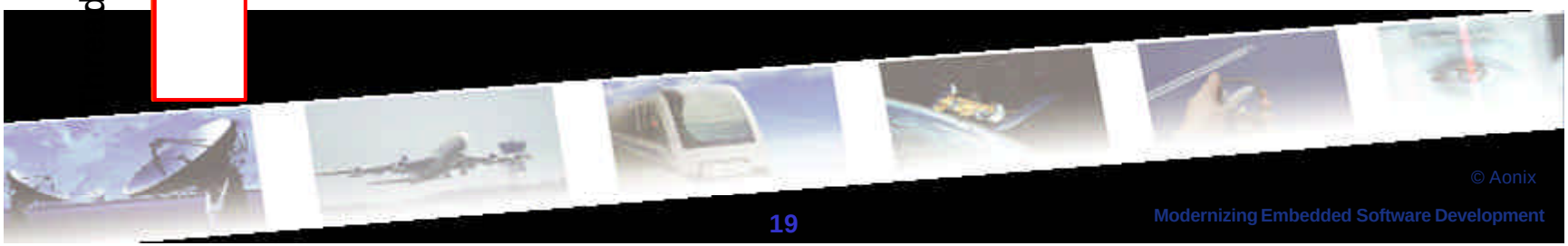
Die Bereichsobjekte jedes Threads können die Bereichsobjekte allokiert in einem weiter außen liegenden Bereich desselben Threads sehen.

Das Thread Stack Modell

Aonix

Der Elternthread muss auf seine erzeugten Threads warten bevor er aus dem Kontext, von dem aus er diese Threads erzeugt hat, zurückkehren kann.





Beispiel für Stackallokierung


```
[1]   package samples;
[2]   import javax.realtime.util.sc.*;
[3]
[4]   public class Complex {
[5]         public float real, imaginary;
[6]         public @ScopedThis Complex(float r, float i) {
[7]               real = r;
[8]               imaginary = i;
[9]         }
[10]
[11]        public @CallerAllocatedResult @ScopedPure Complex add(Complex arg)
[12]        {
[13]              float r, i;
[14]              r = this.real + arg.real;
[15]              i = this.imaginary + arg.imaginary;
```

Beispiel für Stackallokierung


```
[19]        public @CallerAllocatedResult @ScopedPure Complex multiply(Complex arg)
[20]        {
[21]            float r, i;
[22]
[23]            r = this.real * arg.real - this.imaginary * arg.imaginary;
[24]            i = this.real * arg.imaginary + this.imaginary * arg.real;
[25]            return new Complex(r, i);
[26]        }
[27]    }
```

Beispiel für Stackallokierung


```
Complex a, b, i;

a = new Complex(3.5, 2.4);
i = new Complex(0, 1);
b = i.multiply(i);

// Argument von undisciplinedCode hat kein @Scoped
undisciplinedCode(a);
```

Programm legt drei *Complex* Objekte an

Ziel des Verifier: prüfe, ob alle Objekte sicher vom lokalen Frame allokiert werden können

Problem: “a” kann nicht sicher im Stack-frame allokiert werden, weil *undisciplined* Code eine Referenz darauf behalten könnte

Lösung: entweder den Code so annotieren, daß “a” in *ImmortalMemory* angelegt werden kann oder das Argument von *undisciplinedCode* als @Scoped annotieren und mögliche Kompilierfehler nachträglich fixen

```
class MyInterruptHandler extends InterruptHandler implements Atomic {

    static final int INTERRUPT_PRIORITY = 128;
    static final int InterruptNo = 5;
    static final int DataPortAddr = 0x3f0;
    static final int StatusPortAddr = 0x3f4;
    static final int ControlPortAddr = 0x3f8;
    static final byte RcvReady = (byte) 0x08;
    static final byte ResetPort = (byte) 0xff;
    static final int BUFFER_LENGTH = 1024;
    final public byte shared_buffers[][];

    IOPort8IO data_xfer;
    IOPort8I status;
    IOPort8O control;

    int write_buffer_count;
    int read_buffer_count;
    int write_buffer_index;
    boolean reader_waiting;
```

```
public MyInterruptHandler() throws EmbeddedConflictException {
        super(new PriorityParameters(INTERRUPT_PRIORITY), true);

        // The arguments to createIOPort are address, memory-mapped?,
        //  port width, readable?, writeable?
        data_xfer = (IOPort8IO) IOPort.createIOPort(DataPortAddr, true, 8, true, true);
        status = (IOPort8I) IOPort.createIOPort(StatusPortAddr, true, 8, true, false);
        control = (IOPort8O) IOPort.createIOPort(ControlPortAddr, true, 8, false, true);

        write_buffer_count = read_buffer_count = write_buffer_index = 0;
        shared_buffers = new byte [2][];
        shared_buffers[0] = new byte[BUFFER_LENGTH];
        shared_buffers[1] = new byte[BUFFER_LENGTH];
}
```

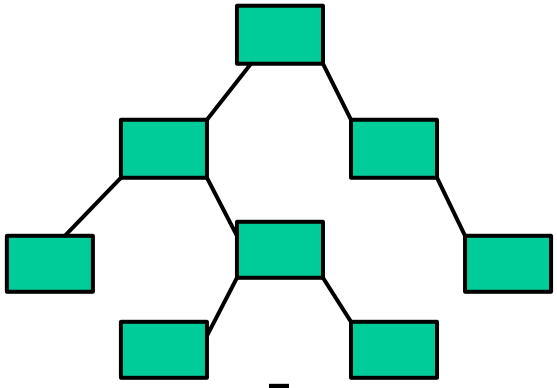
Der Interrupthandler-Code

```
@StaticAnalyzable public synchronized void handleAsyncEvent() {
        byte b;

        if (status.readByte() & 0x01 != 0) {
            b = data_xfer.readByte();
            if (write_buffer_count < BUFFER_LENGTH)
                shared_buffers[write_buffer_index][write_buffer_count++] = b;
            // else, buffer overflow is ignored

            if (reader_waiting)
                this.notify();
        }
    }
}
```

- Historische Relevanz:
  - Ein strategisch wichtiger Kunde aus dem Wehr- und Luftfahrtmarkt zeigte uns diese Applikation vor einigen Jahren
  - Der Kunde evaluierte die damals verfügbaren Technologien und entschied sich dann für C anstatt Java
  - Die hier gezeigte Version des Algorithmus ist vom Original abstrahiert um Betriebsgeheimnisse zu schützen
  - Der Vergleich zwischen den Implementierungen den wir jetzt zeigen ist unser eigener und reflektiert nicht unbedingt die Prioritäten oder Auswahlkriterien des Kunden

Region of Interest-Suchalgorithm

Phase 1



Datenbank mit
'Regions of Interest'

(@ImmortalAllocation)

Phase 2

Suchkriterium: finde die ersten *n* Einträge innerhalb des Bereichs *lower* - *upper*

N Iterationen

Ergebnis der Linked List

(@Scoped)

Linked List von passenden Regionen

Effizienz und Determinismus

Productivity In Real–Time

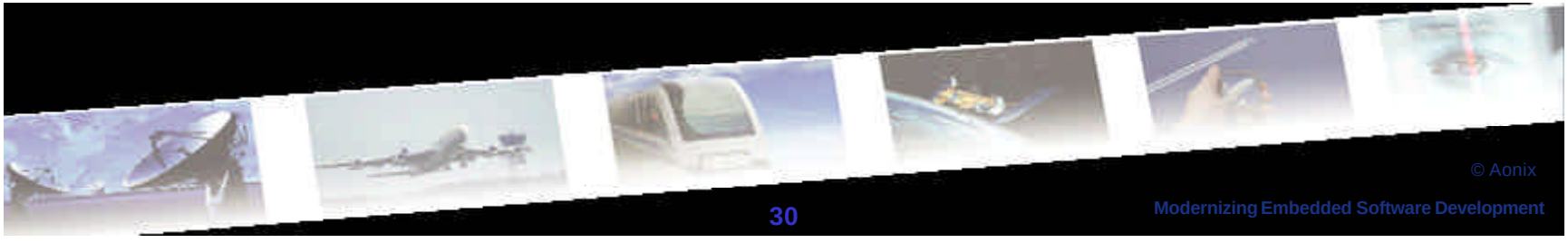
	HotSpot Java	C	PERC Ultra	PERC Pico
Min (ns)	173	280	519	
Max (ns)	8,089	571	1,060	
Mittel (ns)	294	360	639	
Std.abweich. (ns)	228.7896	25.52465	48.66415	
Ges. Virtual Memory (MB)	253	2.32	24	

Vergleich zwischen PERC Pico u

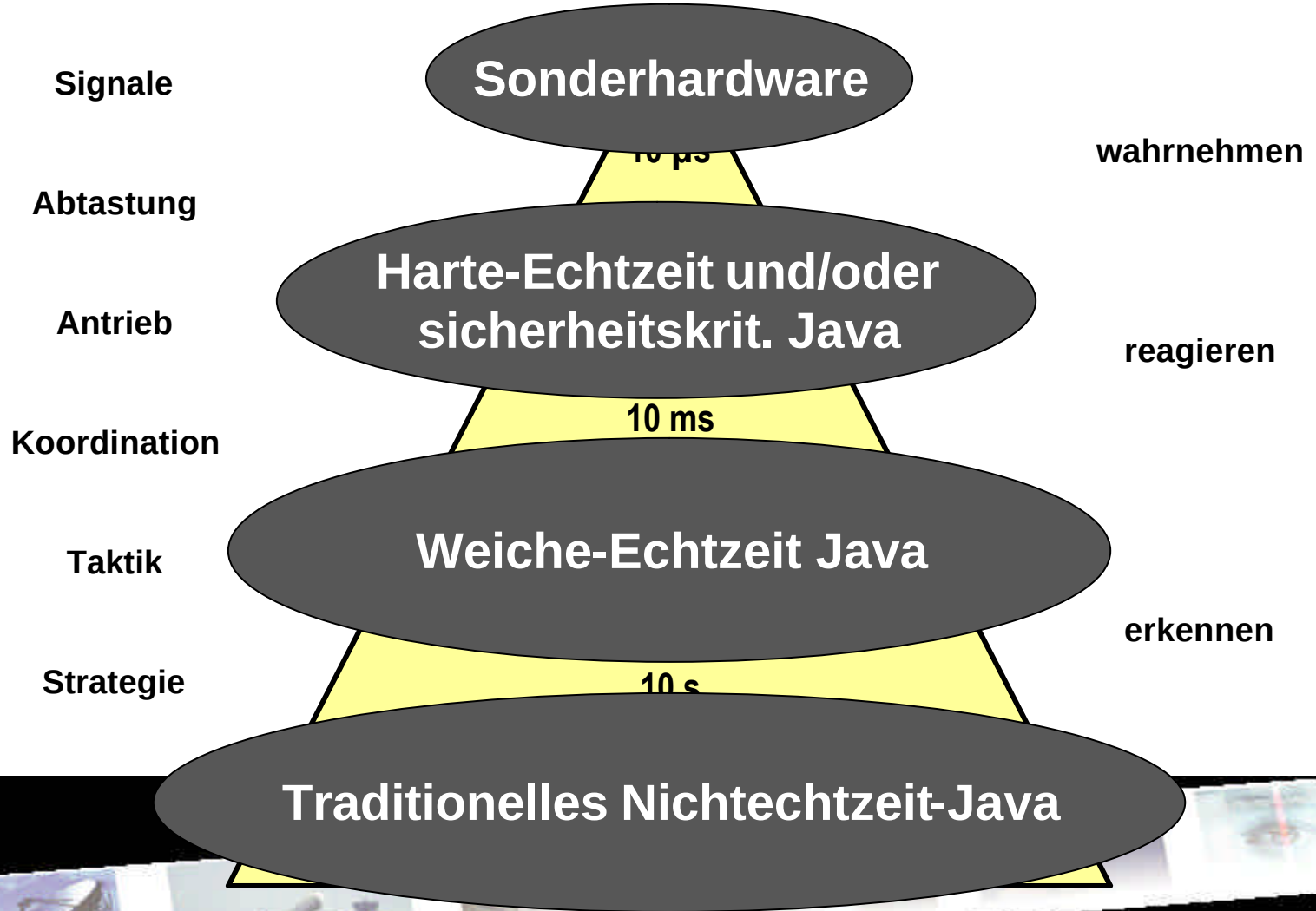
	C	PERC Pico	Comparison
Min (ns)	280	314	+12%
Max (ns)	571	633	+11%
Mittel (ns)	360	392	+9%
Std. abweich. (ns)	25.52465	29.843	
Ges. Memory (MB)	2.32	2.5	

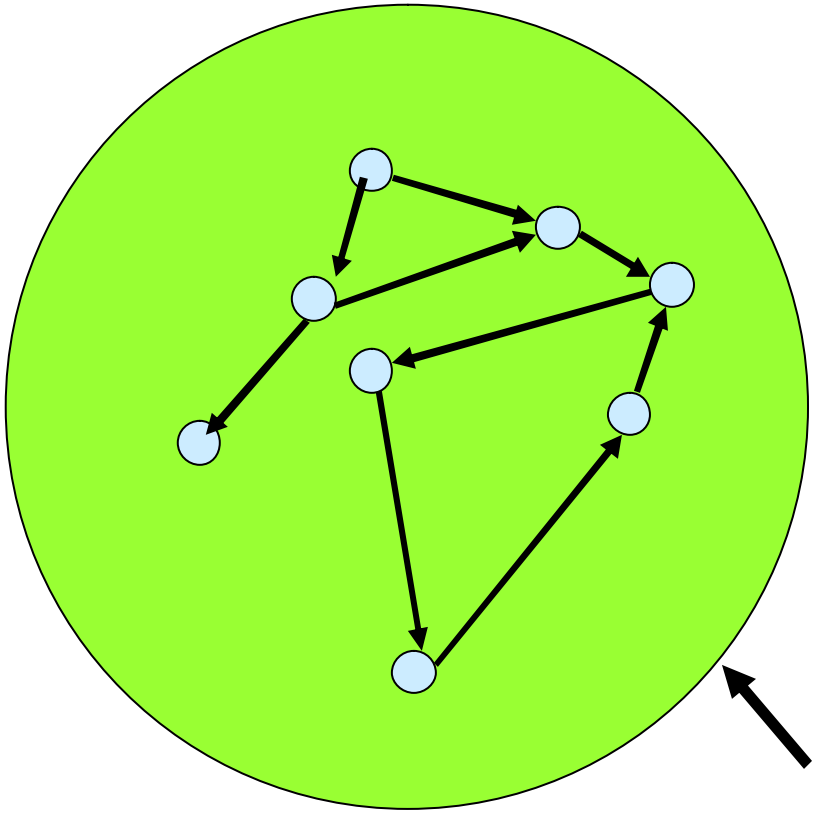
Verschiedene Optimierungen sind für PERC Pico 1.1 geplant (Ende 2. Quartal 2008)

- Die meisten tatsächlich eingesetzten Systeme bestehen aus vielen verschiedenen Technologien, von denen jede andere Bedürfnisse adressiert und andere Kompromisse eingeht.
- Typische Echtzeitsysteme bedienen eine Kombination von Richtlinien für harte und weiche Echtzeit
- C für hardwarenahe harte Echtzeit und Java für weiche Echtzeit auf höhere Ebene zu benutzen ist ungünstig:
  - Geschwindigkeitseinbußen durch Versenden von Daten
  - Probleme in Verlässlichkeit und Wartung weil C-Code die Integrität der Java-Sicherheitsmechanismen kompromittiert.

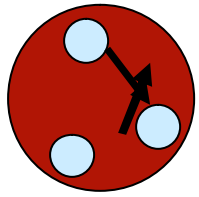


Echtzeit-Pyramide (Hierarchien u





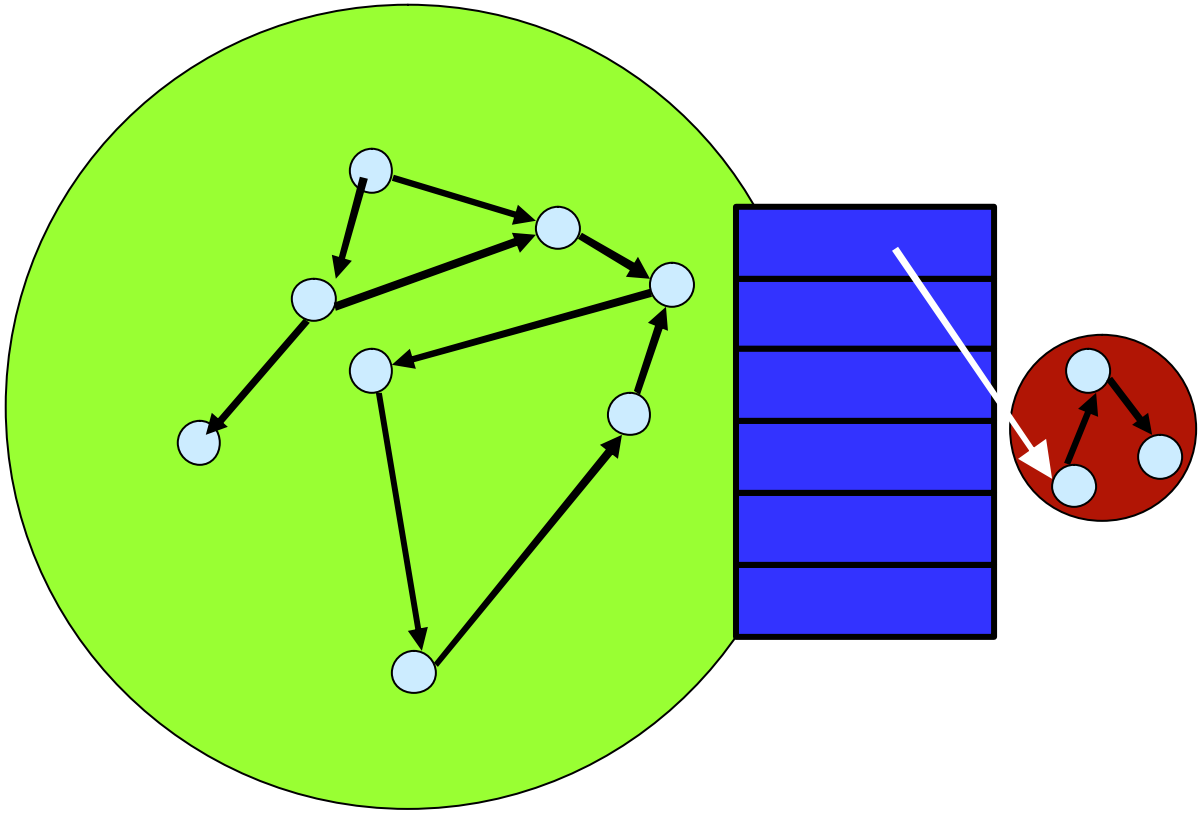
Pico Hard
Real-Time
Execution Engine



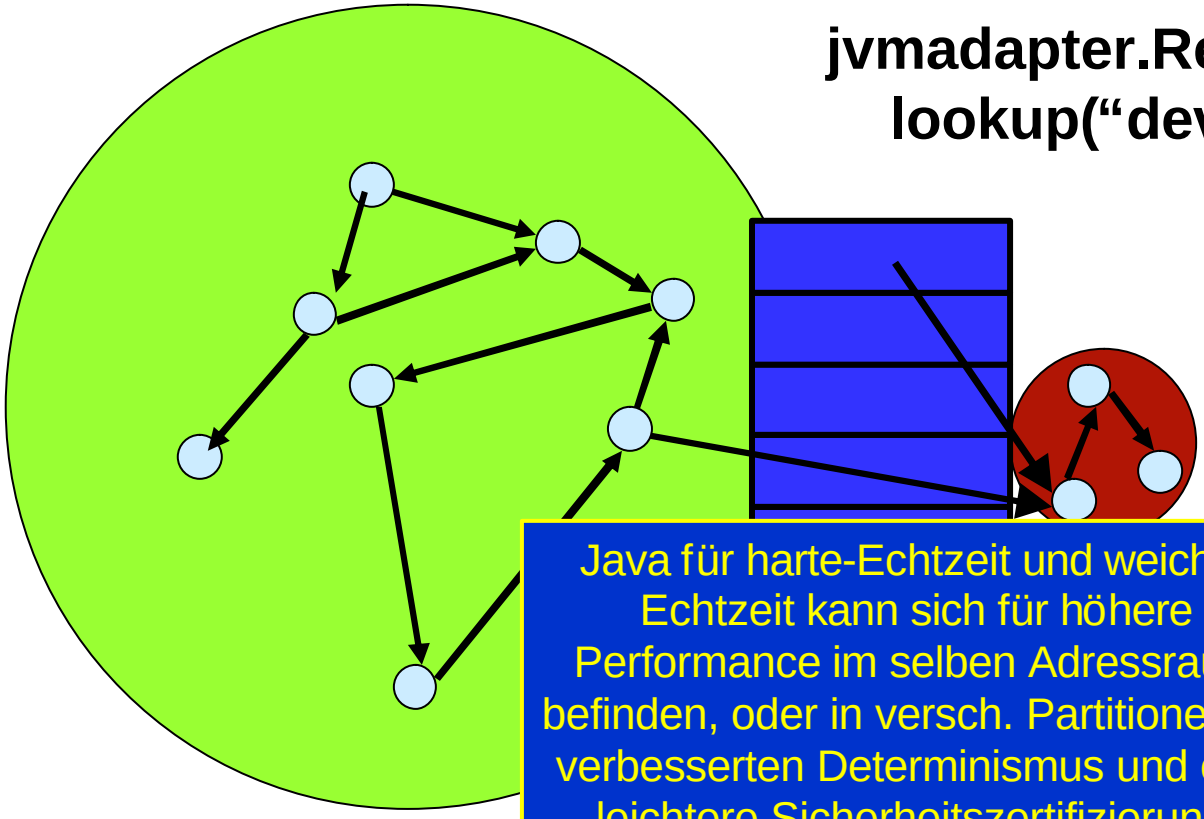
Ultra Virtual Machine

TraditionalJava Registry

Productivity In Real–Time



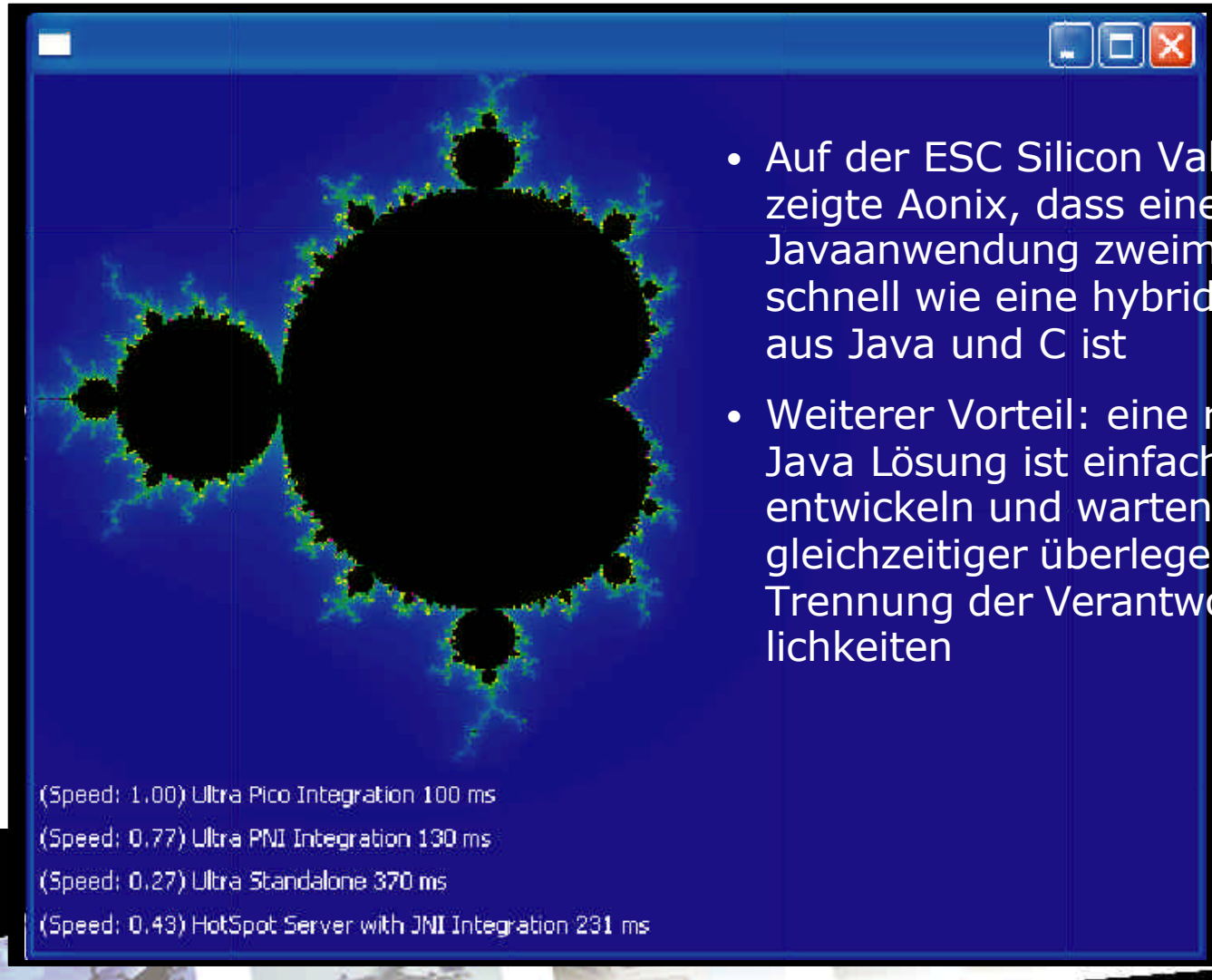
TraditionalJava Registry



jvmadapter.Registry.
lookup(“deviceXYZ”)

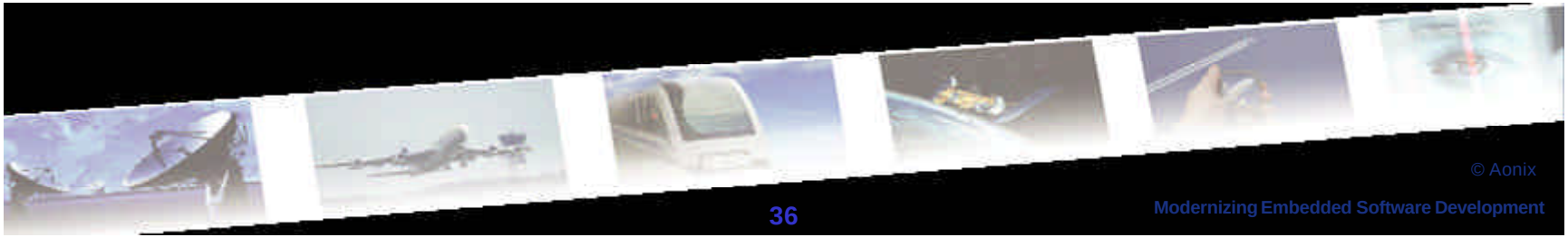
Java für harte-Echtzeit und weiche-Echtzeit kann sich für höhere Performance im selben Adressraum befinden, oder in versch. Partitionen für verbesserten Determinismus und eine leichtere Sicherheitszertifizierung.

Reine Javalösung 2x so schnell



- Auf der ESC Silicon Valley 2007 zeigte Aonix, dass eine reine Javaanwendung zweimal so schnell wie eine hybride Lösung aus Java und C ist
- Weiterer Vorteil: eine reine Java Lösung ist einfacher zu entwickeln und warten, bei gleichzeitiger überlegener Trennung der Verantwort-lichkeiten

- Verglichen mit den althergebrachten Sprachen für Echtzeit, hat Java das Potential für:
  - eine Verdopplung der Entwicklerproduktivität
  - eine zehnfach effizientere Wartungsphase
  - die Benutzung von Standardsoftwarekomponenten
  - leichte Anwerbung kompetenter Entwickler
- Java für weiche Echtzeit hat sich bereits in vielen geschäftskritischen Installationen bewährt
- Java für harte Echtzeit ist jetzt verfügbar!



DANKE!