Compiling Java for Real-Time Systems

Anders Nilsson

andersn@cs.lth.se

Department of Computer Science, Lund University, Sweden

LUND INSTITUTE OF TECHNOLOGY
Lund University

Outline

- Introduction
- Approach
- Real-Time Execution Platform
- A Compiler for Real-Time Java
- Experimental Verification
- Future Work
- Conclusions

Introduction

- Most computers in the world are embedded, with various RT demands
- Software complexity is increasing drastically
- Programming language problems:
  - Type casts, as in C
  - Pointer arithmetics
  - No array bounds checking
  - Manual memory management, malloc/free
  - Lack of encapsulation

Bugs are easily created, but hard to find

Hypothesis

Safe OO programming languages proved beneficial in other software development areas

- Encapsulation
- Strict type safety
- Many errors caught by compiler. Remaining errors caught and handled by run-time checks
- Automatic memory management

All possible results of the execution are expressed by the source code

Java or C#

- Both are safe (except explicit `unsafe` in C#) OO programming languages
- Built-in concurrency and synchronization
- Exception handling
- Platform "independent"

But, Java is more mature, available on more development platforms, and there is an open-source class library available

Problem Statement

Can standard Java be used as a programming language on arbitrary hardware platforms with varying degrees of real-time-, memory footprint-, and performance demands?

or

Write once run anywhere, for severely resource-constrained real-time systems?

Standards

Two RT Java standards; RTJ and JConsortium
None complies with Real (J2SE) Java:

- Assuming (hard) RTGC not feasible
- Numerous memory types: Immortal,Scoped, Raw, Heap
- Effectively return memory management to the programmer

Several Java benefits are lost.
There must be a better way!

Key Concepts

Considerations for Real Real-Time Java in embedded systems.

- Portability
- Scalability
- Real-time execution and performance
- Real-time communication
- Applicability

Utilize the language, adopt run-time to embedded needs

Approach

1. *Small* memory footprint and high performance

   $$\Downarrow$$

   Natively compiled Java (no JVM)

2. Any hardware comes with a C compiler

   $$\Downarrow$$

   Use ANSI C as intermediate (high-level assembly) language

External Code

Need to link with external (not GC-aware) code.

- Hardware device drivers.
- Code libraries.
- Legacy software.
- Automatically generated code from high-level tools (Matlab/Real-Time Workshop).

Typically, no (usable) source code available.

RT Memory Mangement

- Incremental GC in a medium priority thread.
- High priority threads pay no overhead penalty during allocation.
- Low priority threads pay overhead for themselves AND the high priority threads.

Compacting GC

$\Rightarrow$ Shorter maximum latencies than malloc/free.

Compacting GC

- Schedule so as not to disturb high-priority threads
- Read-barrier needed; objects relocate
- Calls to external functions become critical sections
- No fragmentation
- Average performance normally decreases

Non-Moving GC

- Schedule so as not to disturb high-priority threads
- Fixed size memory blocks $\Rightarrow$ large objects are split in several memory blocks. Problematic with external code
- No read barrier

Less overhead than Compacting GC

C/C++ compatibility option

Non-moving GC with variable memory block sizes and a good memory allocator

- Johnstone et al. 1998. Memory fragmentation is not a serious problem in real applications
- No read barrier
- As deterministic as using `malloc()` and `free()` in C
- Can call external code, that uses Java objects

Latency and Preemption

- Native preemption promotes short latency and allows external code, but may introduce (external) fragmentation
  $\Rightarrow$ deficient predictability (as in C++)
- 100% Java and appropriate run-time
  $\Rightarrow$ **Hard** RT Java.
  To improve average performance:
  - Preemption points $\Rightarrow$ higher latency
  - Block-based GC $\Rightarrow$ internal fragmentation

Hence a tradeoff: Latency $\leftrightarrow$ Fragmentation

Real-Time Execution Environment

Frenchmen, I die guiltless of the countless crimes imputed to me.
Pray God my blood fall not on France!

Lois XVI, 1793

Real-Time Execution Environment

- Garbage Collector Interface
  - Different strategies require different code
- Class Library
  - Native methods using GCI. Domain-specific I/O
- Threads and Synchronization
  - Map thread primitives on native OS
  - RTThread classes @CS
- Exceptions
  - Only one active exception per thread
  - Implemented with `setjmp/longjmp`

Garbage Collector Interface

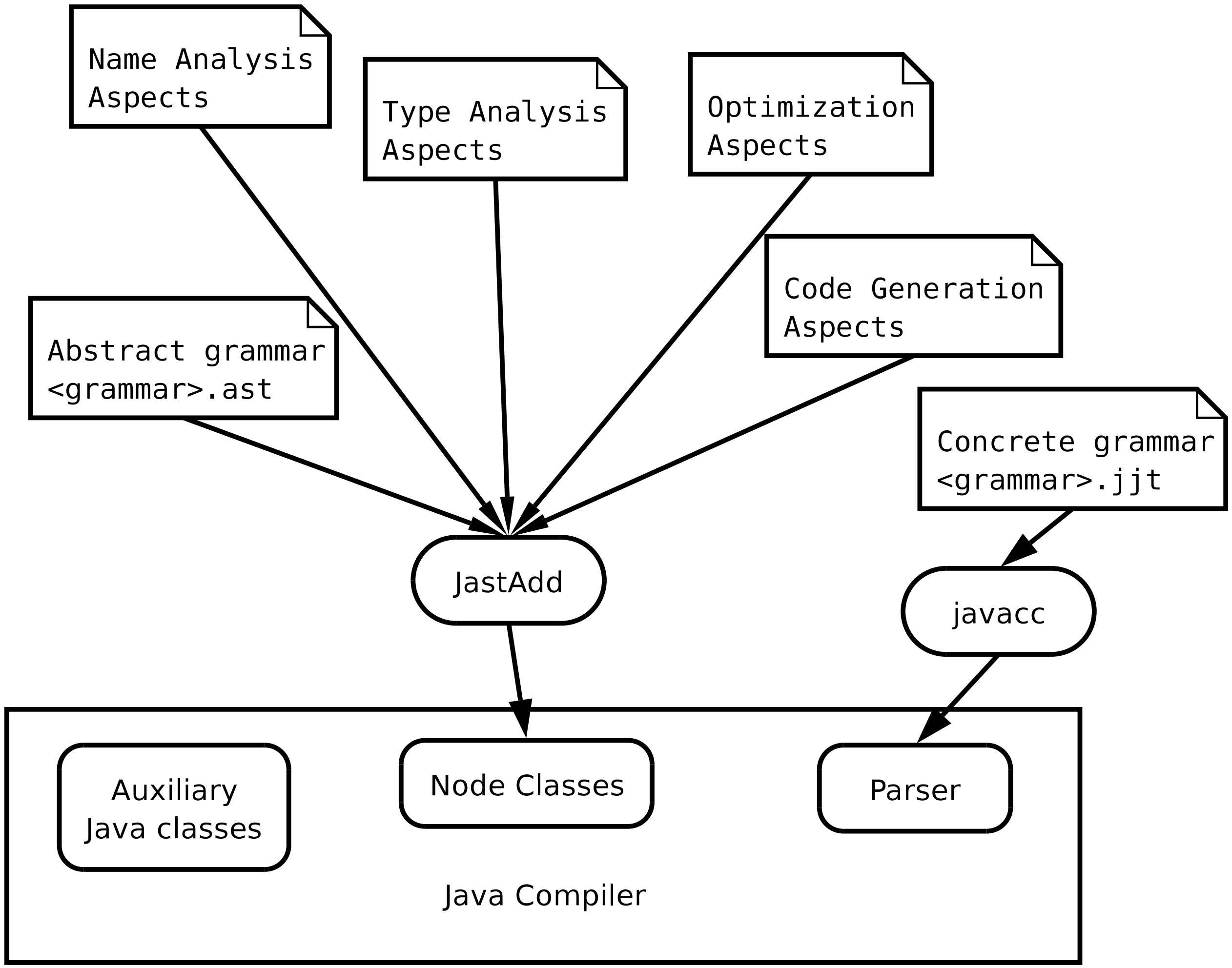
```
class MyClass {
  void foo() {
    String foo = new
        String("Hello World!");
    System.out.println(foo);
  }
}
```

```
GC_PROC_BEGIN(_MyClass_foo,
              GC_PARAM(MyClass,this))
    GC_PARAM_REF(MyClass,this);
    GC_PUSH_PARAM(this);
    GC_ENTER
    GC_REF(String,foo); GC_PUSH_ROOT(foo
    GC_NEW(String,foo,"Hello World!");
    GC_PROC_CALL(System_out_println,foo)
    GC_POP_ROOT(foo);
    GC_LEAVE
    GC_POP_PARAM(this);
GC_PROC_END(_MyClass_foo)
```

Java Compiler

- Java based compiler-compiler, generating Java (to C) translator, in Java
- Based on Reference Attributed Grammars, JastAdd
- AOP for modular semantics, optimization and code generating

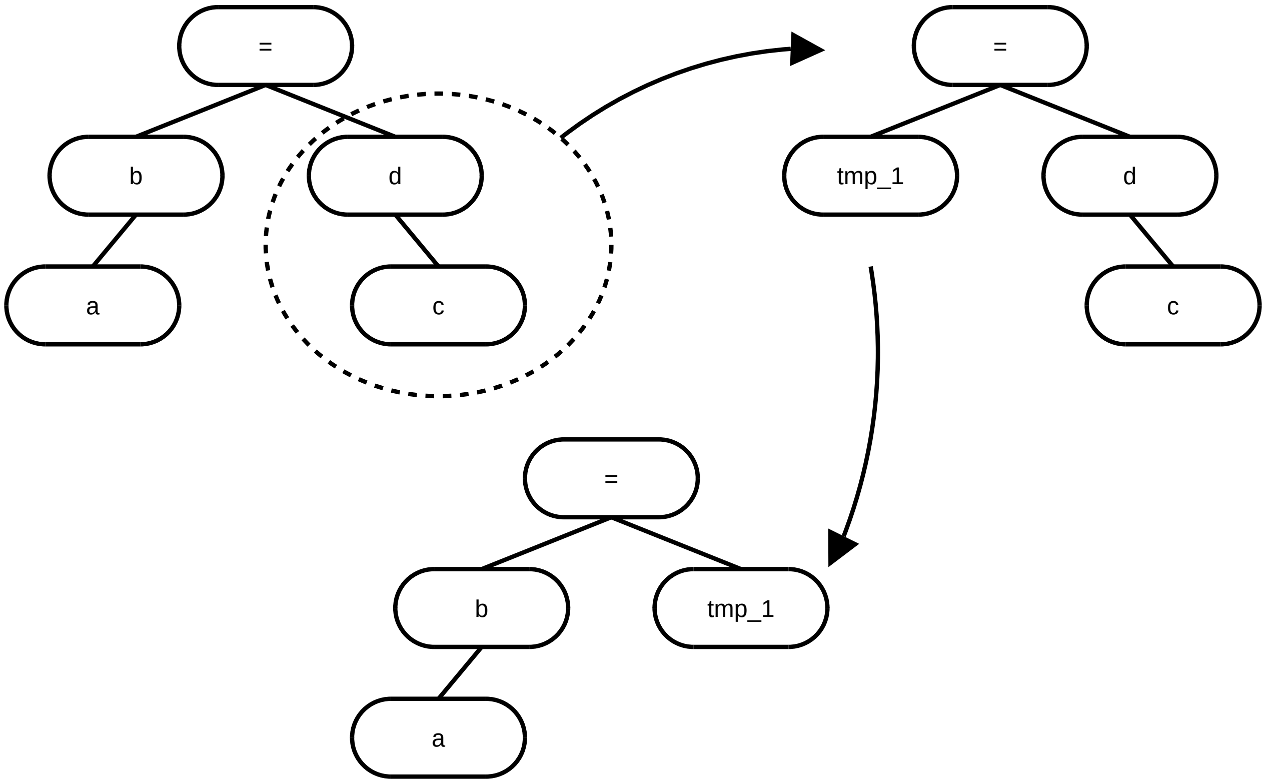
Compiler Overview



Name Transformations

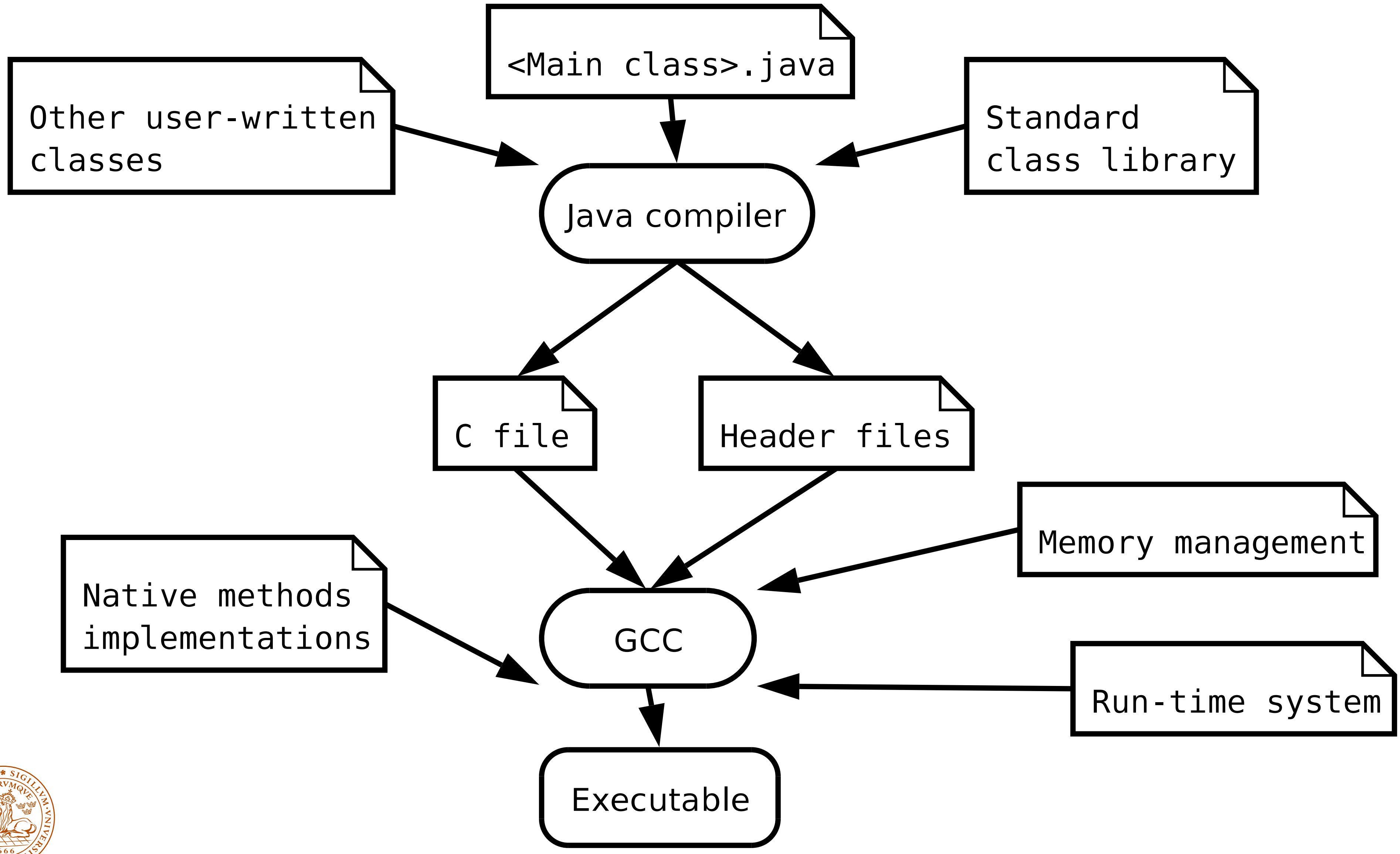
```
a.b = c        ⇒        GC_SET(a,b,c)
```

```
a.b = c.d;
     ⇓
tmp_1 = c.d;
a.b = tmp_1;
```



LUND INSTITUTE OF TECHNOLOGY
Lund University

Code Generation



Evaluation

	Lines of code
Parser and AST	
Abstract Grammar	181
Concrete Grammar	1044
Semantic Analysis	
Name- and Type Analysis	1458
Transformations and Optimizations	
Simplifications	901
Dead Code Optimization	154
Code Generation	
C code generation	5745
TOTAL	9473

Compiler Performance

	Our compiler	gcj	javac
HelloWorld			
Memory usage (MB)	14	<5	21
Time (s)	26	0.65	3
RobotController			
Memory usage (MB)	34	-	30
Time (s)	160	-	9

LUND INSTITUTE OF TECHNOLOGY
Lund University

Experimental Verification

To which extent are the key concept requirements fulfilled?

- Portability
- Scalability
- Real-time execution and performance
- Real-time communication
- Applicability

Portability

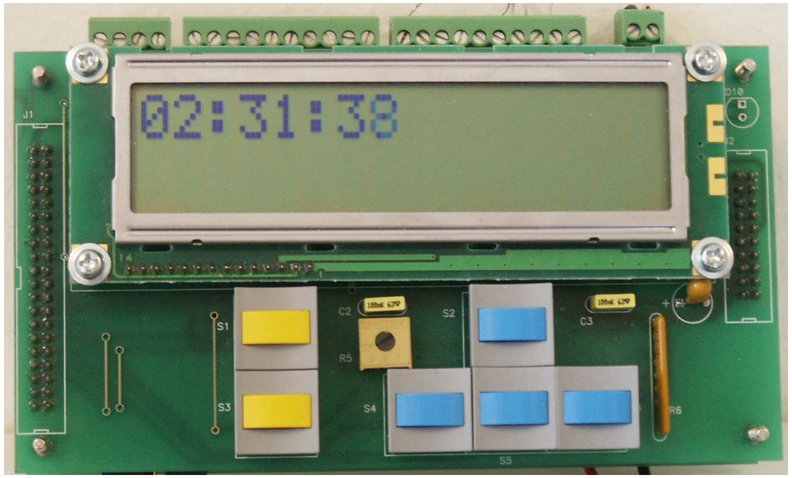
Current supported platforms

	AVR	PPC	i386	SPARC
CSRTK	X			
STORK		X		
Linux RTAI(k)		X	X	
Linux RTAI(u)		X	X	
Posix		X	X	X

Scalability

Low end prototype

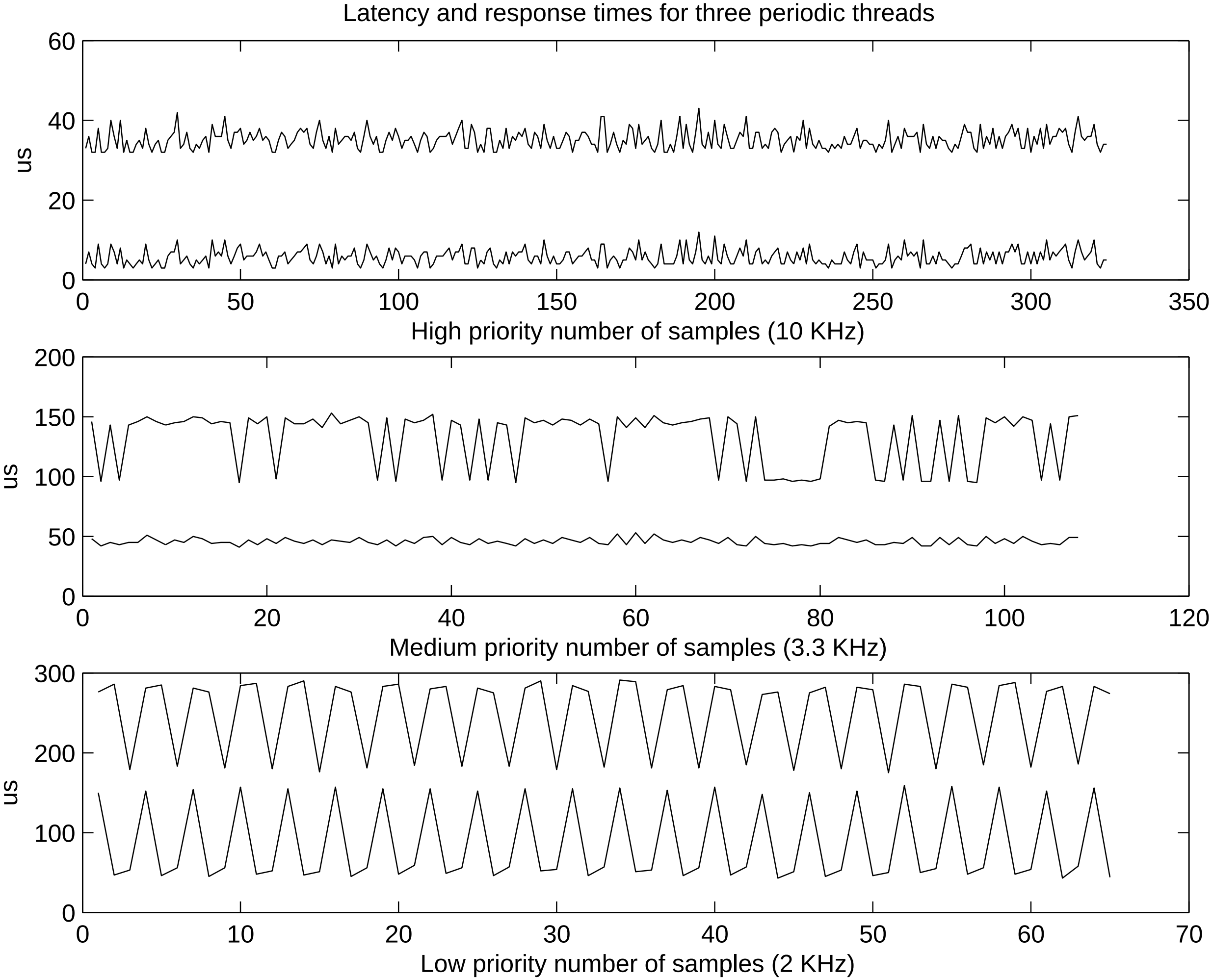
- Atmel AVR ATmega 103
  - 8 bit RISC Architecture, 6 MHz $\Rightarrow$ 6 MIPS
  - 32 Registers, 128 KB Flash, 4KB RAM
  - Real-Time Clock, UART, Timers, 8-channel 10-bit ADC
- LCD Display, Summer, 6 buttons
- Tiny in-house RTOS



Multi(3)-threaded application in less than 62 KB ROM and 32 KB RAM, including run-time

Real-Time Execution

	T_1	T_2	T_3	GC_1
Period (μs)	100	300	500	NA
Workload (μs)	30	50	90	NA



General Performance

	fibonacci (virtual)	fibonacci (static)	scalar
Our compiler (ms)			
mark-compact GC	10050	7012	146400
mark-sweep GC	7002	6904	7760
no GC	753	586	5402
Other (ms)			
Sun JVM	271	251	5085
Sun JVM -server	270	245	3910
Sun JVM -Xint	3302	3120	52500
GCJ	360	567	10098
GCJ -O3	328	504	2249
Hand-written C			
GCC	NA	280	6810
GCC -O3	NA	293	761

Real-Time Communication

- Real-time network protocol available: ThrottleNet (`@control.lth.se`)
- Successful experiments with compiled Java and RTAI: Patrycja Grudziecka and Daniel Nyberg (2004)

Applicability

Tested on many platforms with different levels of real-time requirements.

- Atmel AVR, Hard real-time
- Motorola PPC G4, Hard real-time
- RTAI Linux, Hard real-time
- Posix, No real-time

Conclusions – General

- Java (safe & portable) highly desirable for flexible RT systems
- Use the language (no JVM) for embedded systems!
- Real (based on J2SE) Real-Time Java is feasible
- Standard memory model to be kept (RTGC is ok)

Conclusions – tradeoffs

Non-moving GC

- \+ Can link with external binary code that can use Java objects
- \+ Latency as good as C++
- \- Predictability as bad as with C++

Compacting GC

- \- 100% Java (or open source) required (no ext. code)
- \- Decreased average performance
- \+ Hard RT Java!

Trade latency for:

- predictability (using compacting GC)
- average performance (using compacting GC and preemption points)

Contributions – Real-Time Java

- A prototype implementation of hard Real-Time Java
- The Garbage Collector Interface
- A Real-Time Exception implementation
- Latency $\Leftrightarrow$ Predictability tradeoff

Conclusions – CC

- OO AST, RAGs and AOP renders a very compact, yet clear compiler implementation
- Code analysis, refactorings and optimizations are conveniently described as aspects, possibly performing transformations, on the AST
- Current implementation is substantially slower than other compilers, but still fast enough.

Contributions – CC

- A compiler for a complete real-world OO language, based on RAGs and AOP
- A new way of implementing high-level optimizations as a set of AST transformations
- Use AST transformations to simplify expressions makes code generation easier

Future Work

- Full-scale applications
- Improve general performance
  - GC synchronization, memory allocation and OO optimizations
- Networking
- Dynamic class loading
  - Klas Nilsson et al. 1998
- WCET analysis
  - Patrik Persson 2000
- Hybrid execution environment
  - Compiled Java $\Leftrightarrow$ JVM

The End

- Portability – OK
- Scalability – OK
- Real-time execution – OK
- Real-time communication – OK
- Applicability – OK
- Performance – Needs more work

Write once run anywhere, for severely resource-constrained real-time systems!