Towards Automatic Specialization of Java Programs

ECOOP ’99, June 17

Ulrik Pagh Schultz, Julia L. Lawall, Charles Consel, Gilles Muller

Compose Project, IRISA/INRIA

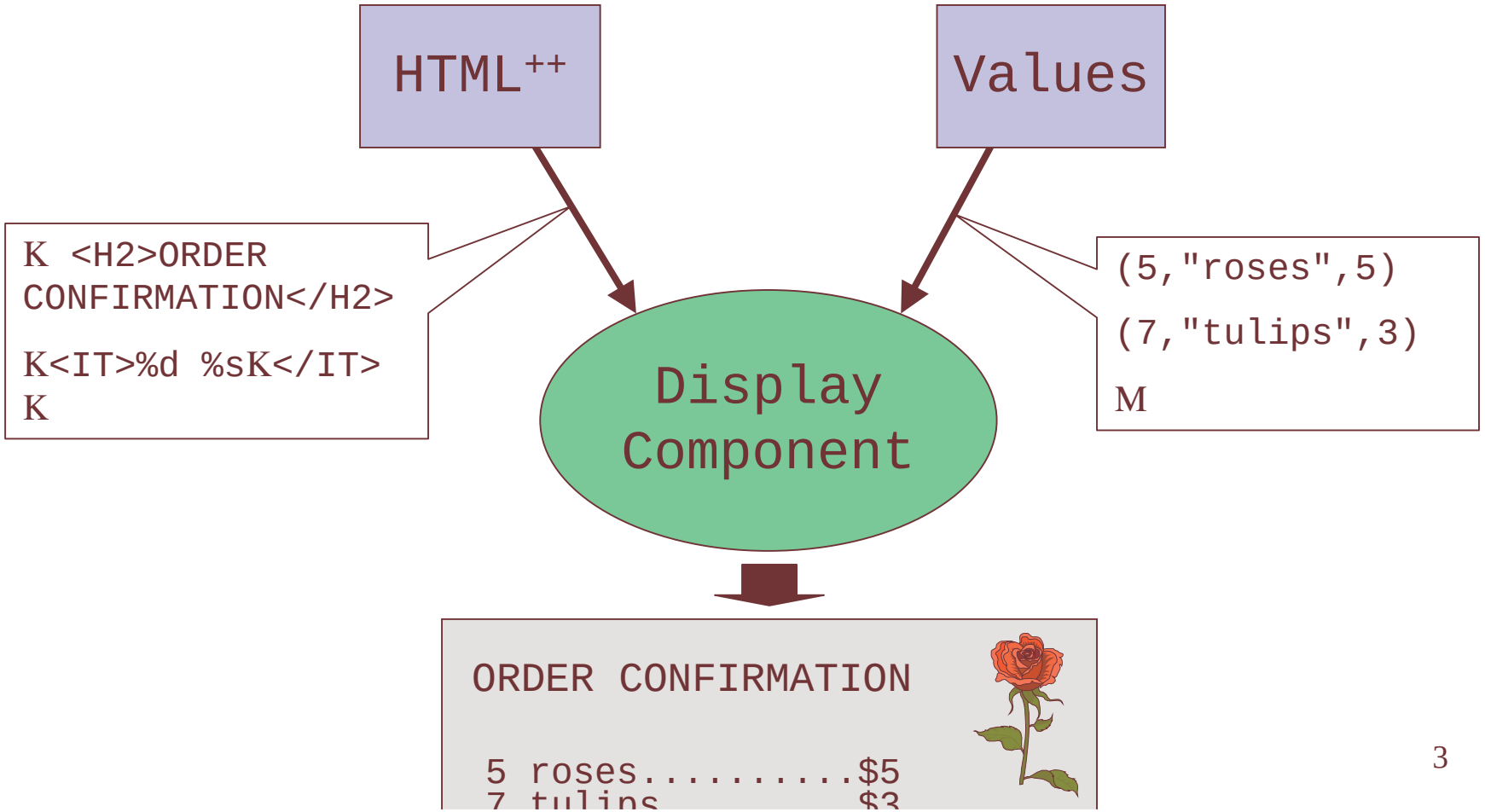
France

Object-oriented languages encourage

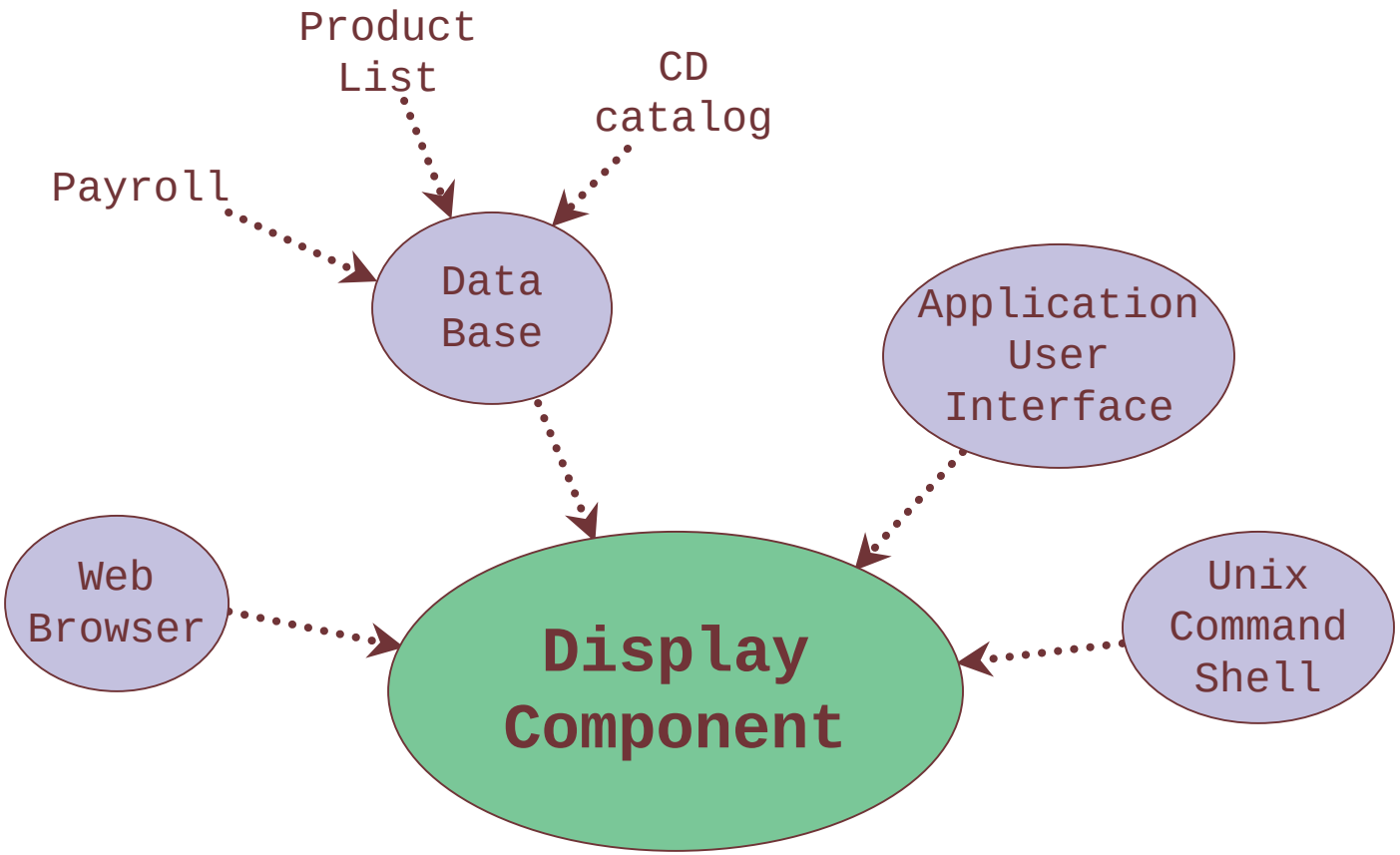
- modularity through objects and classes
- flexibility through virtual dispatching
- reusability through encapsulation

But what about time and space efficiency?

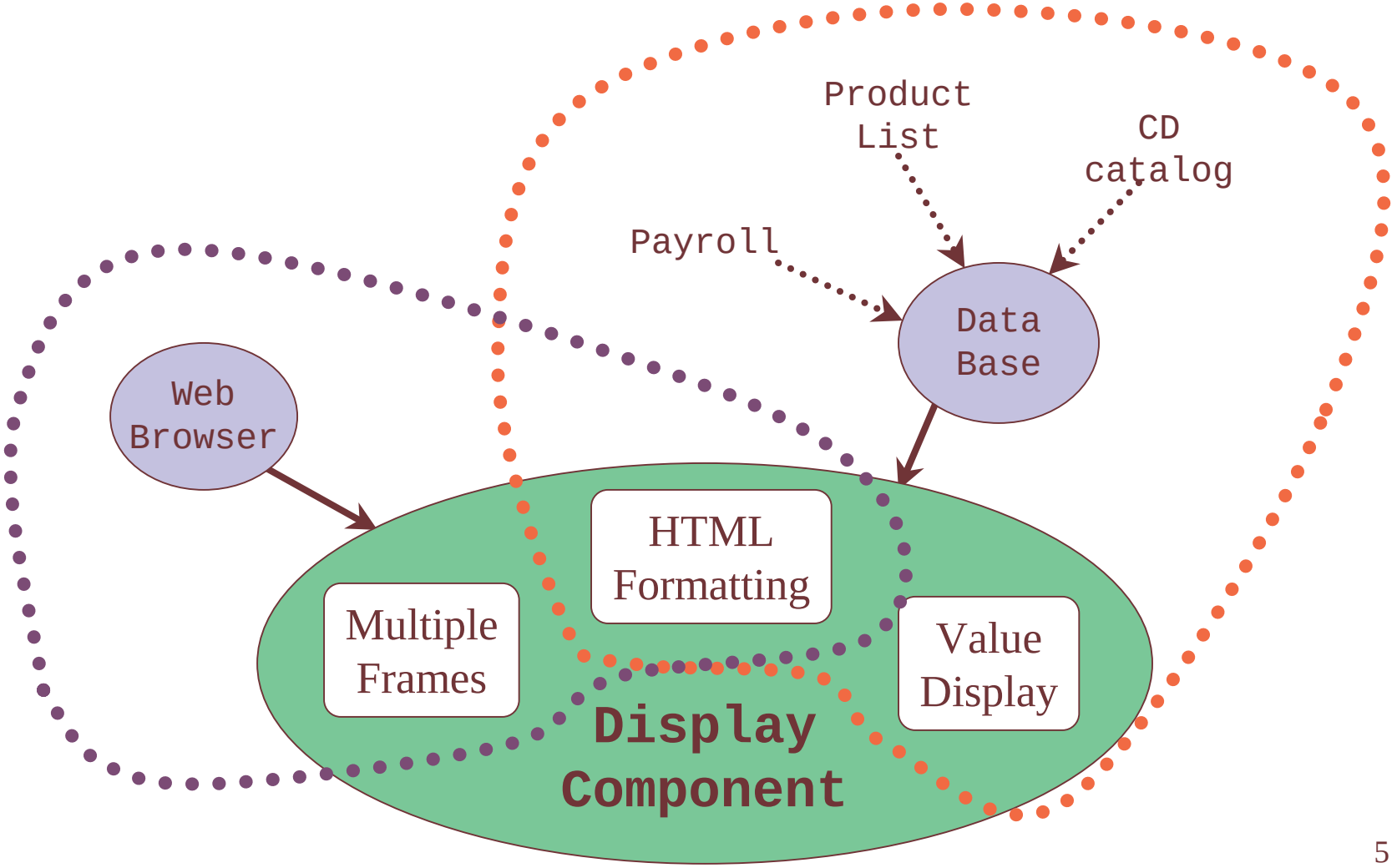
Example: Generic Component



Example: Generic Component



Specific Components?



Solution:

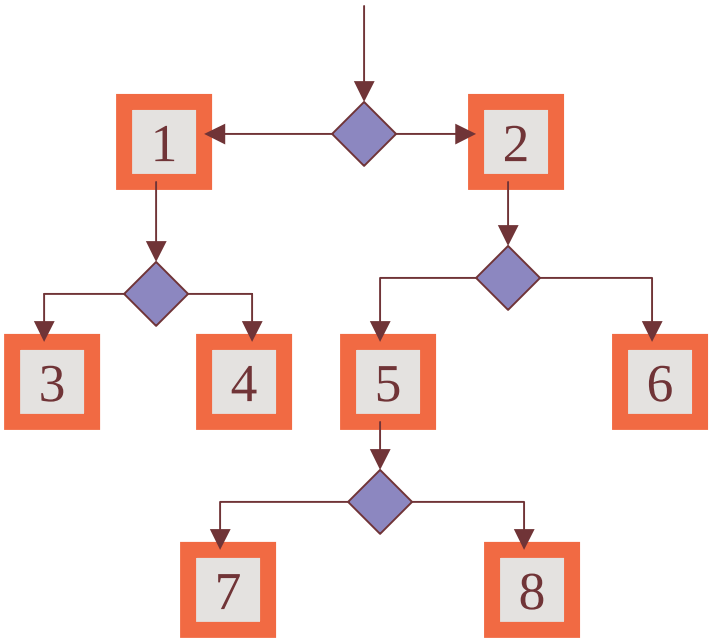
Automatic Program Specialization

Program Specialization?

- Automatically generates specific implementations from a generic one
- Logical, functional, and imperative languages
- Successfully applied to realistic programs:
  - OS components (Sun RPC, Chorus IPC, K)
  - Scientific computations (FFT, interpolation, K)
  - Graphics (ray tracing, image filtering, K)

What Does Specialization Do?

- Specializes program to a context
- Globally propagate static values
- Leave behind only dynamic computations



Generic

Specific

So What About Java?

- Java = Imperative + Object-Oriented
- Imperative: been there, done that
- Object-Oriented mechanisms:

  Re-use imperative features

  – Objects modelled as structures,

  – Virtual calls modelled as indirect function calls

  – …

Result of Specializing a Java Program

- Globally: program specialized to the context
  - Methods adapted to the object context
  - Static parameters globally propagated
  - Monolithic program
- Locally: propagate and reduce
  - Single framework for imperative and object-oriented features

Example: Encapsulation of Values

```
class Int {
  private int i;
  Int(int j){i=j;}
  int value() {
    return i;
  }
}

int sum(Int s,Int d) {
  return s.value()
         +d.value();
}
```

s.i=87

- Values are systematically encapsulated
- Specialization globally propagates static values

```
int sum1( Int d ) {
  return 87+d.value();
}
```

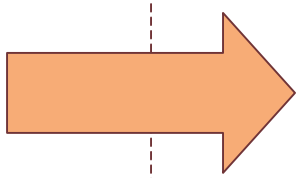
Generic — static dynamic — Specific

Example: Virtual Call Elimination

```
interface Fn {
  int apply(int i);
}
class Square implements Fn {
  int apply(int i) {
    return i*i;
  }
}
void use(Fn f) {
  M
  while(0<k--)
    x=f.apply(x);
  M
}
```

f is a Square

- Virtual calls complicate control flow
- Specialization simplifies control flow



```
void use1() {
  M
  while(0<k--)
    x=x*x;
  M
}
```

(after inlining)

Generic

static dynamic

Specific

Implementation

- C serves an intermediate language
  - Harissa translates Java to C
  - Tempo specializes C
  - Optional back-translation to Java (at work)
- Re-use of existing technology
- Cheap exploration of OO specialization
- No loss of generality
- Permits implementation-level specialization

Example: Low-Level Specialization

t.length=7

```
int last(int[] t) {
 return t[t.length-1];
}
```

t->length=7

```
int last(Array_int *t) {
 int i=t->length-1;
 if(i<0||i>=t->length)
  throwExn(Λ);
 return t->data[i];
}
```

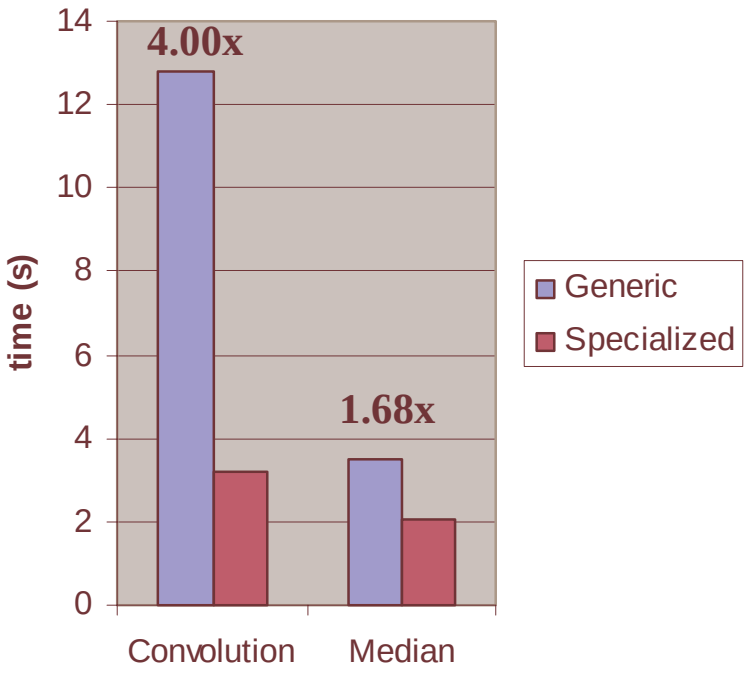
Generic

- Java operations include implicit check
- Specialization eliminates static tests
- Check done before program execution

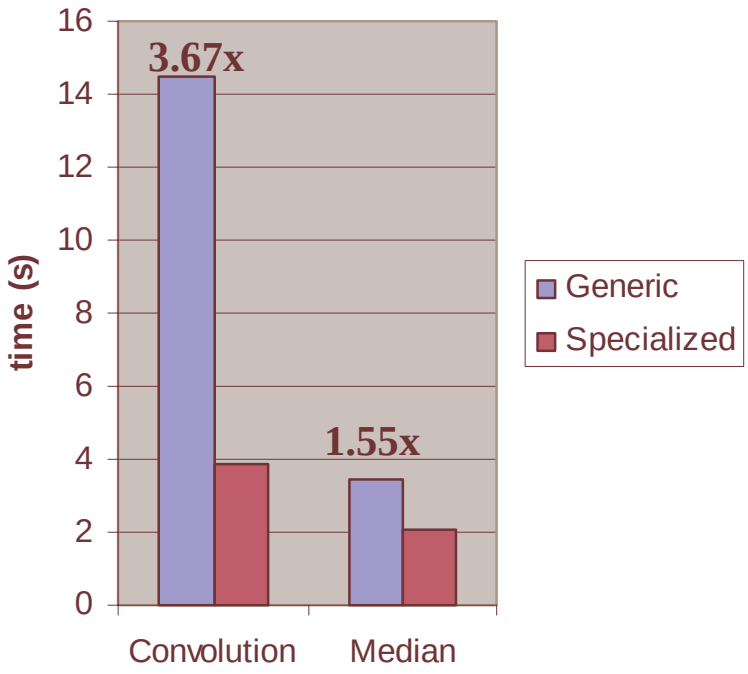
```
int last(Array_int *t) {
 return t->data[6];
}
```

Specific

Assessment: Image Processing Application (Harissa)



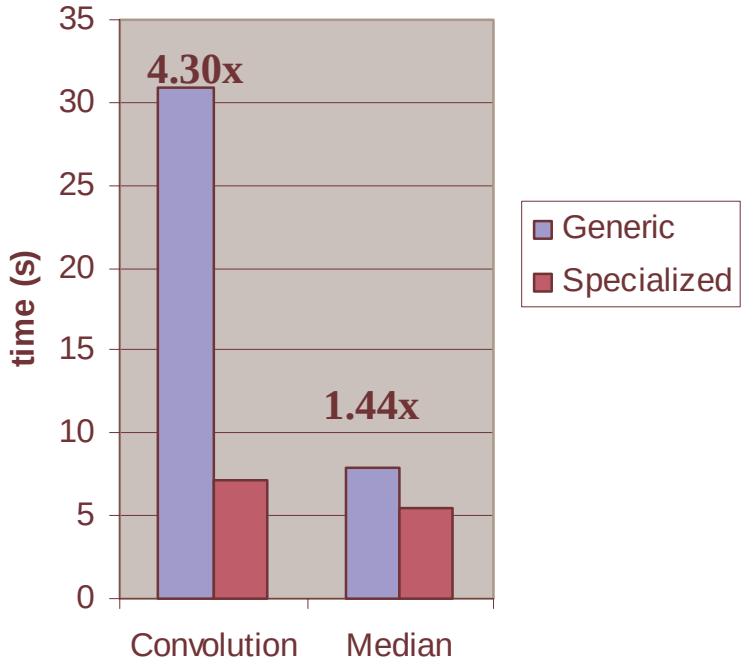
SPARC



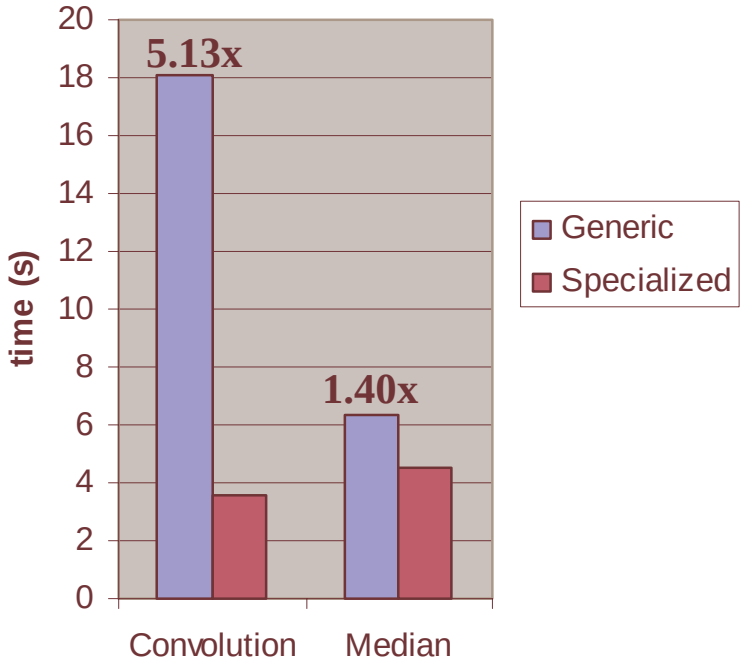
Pentium

Back To Java? Prospective Benchmarks

JDK 1.2b4r



SPARC



Pentium

Future Directions

- Back-translation to Java
- Run-time specialization using Tempo
- Structured programs: software components
- Systematizing the approach: design patterns

Summary

- Object-oriented languages encourage generic programming
- **Generic** —Program Specialization→ Specific
  - modular
  - flexible
  - reusable
  - Efficient
- Java-to-C done, Java-to-Java coming

Compose Project URL: `http://www.irisa.fr/compose`