TEMPO,
A Program Specializer for C

Renaud MARLET

Compose group

IRISA / INRIA Rennes (France)

What it is / What it does

- Automatic compile-time and run-time specialization
- Program and data specialization
- Modular specialization
- Incremental specialization

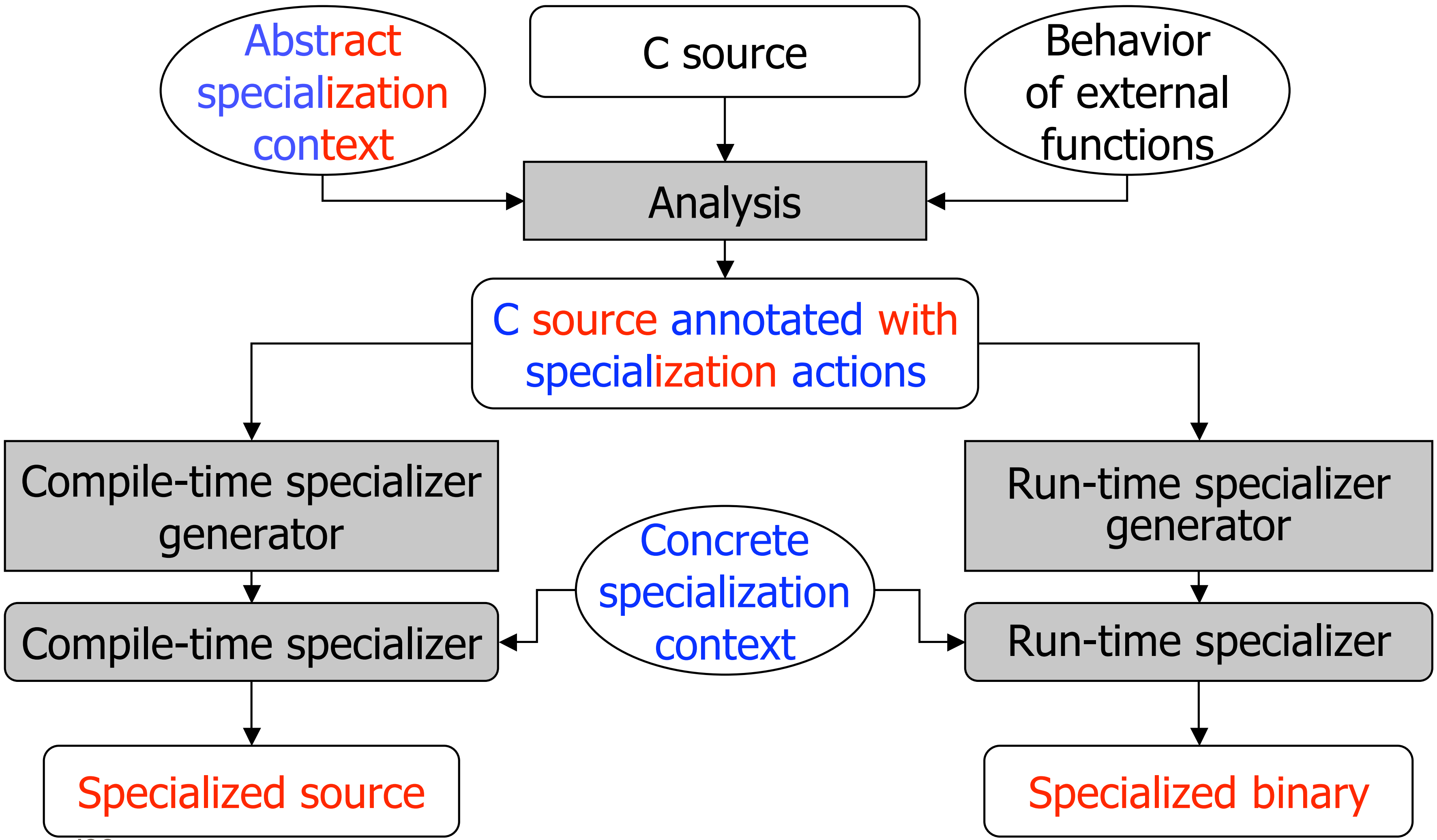
- Real-size applications (~ 6,000 specialized lines) ∵

- Back-end partial evaluator for Java (Jspec)
- Publicly available (~ 40 licenses)

Some Applications of Tempo

- Operating systems *[PEPM'97, ICDCS'97]*
  - Sun RPC (3.7x), Chorus IPC (1.5x), BPF (4x)
- Numerical computations *[LNCS, ICCL'98, PEPM'99]*
  - FFT (4–12x), standard library routines
- Computer graphics *[ECOOP'99]*
  - Convolution filters (4x)
- Software architectures *[ASE'97]*
  - Selective broadcast, software layers, generic libraries, …
- Compilers/JITs for interpreters *[DSL'97, SRDS'98, ICDCS'99]*
  - PLAN-P (80x, 96% of C throughput), O'Caml (1.2–2.5x) …

Overview


Abstract specialization context
C source
Behavior of external functions
Analysis
C source annotated with specialization actions
Compile-time specializer generator
Run-time specializer generator
Concrete specialization context
Compile-time specializer
Run-time specializer
Specialized source
Specialized binary

Stages: **S D**

Specialization Templates

```
dotprod(size,u[],v[])
{
  res = 0;                                  T1
  for(i = 0; i < size; i++)
  {                      H1         H2
    res += u[i] * v[i];                     T2
  }
  return res;
}                                           T3
```

```
size=3   u[]={7,4,6}
```

```
dotprod_size_u(v[])
{
  res = 0;                  T1
  res += 7 * v[0];          T2
  res += 4 * v[1];          T2
  res += 6 * v[2];          T2
  return res;
}                           T3
```

`dotprod_size_u(v[])` ---> | *T1* | *T2[7 , 0]* | *T2[4 , 1]* | *T2[6 , 2]* | *T3* |

Dedicated Run-Time Specializer

Code generation instructions: ■ | Stages: **S** **D**

```
dotprod(size,u[],v[])
{
  res = 0;                          T1
  for(i = 0; i < size; i++)
  {                  H1       H2
    res += u[i] * v[i];             T2
  }
  return res;
}                                   T3
```

```
dotprod_spec(size,u[])
{
  buf = alloc();
  copy_temp(buf,T1);
  for(i = 0; i < size; i++)
  {
    copy_temp(buf,T2);
    fill_hole(buf,H1,u[i]);
    fill_hole(buf,H2,i);
  }
  copy_temp(buf,T3);
  return buf;
}
```

buf ---→

T1	T2[u[0] , 0]	T2[u[1] , 1]	T2[u[2] , 2]	T3	

Tentative Balance-Sheet for Tempo (1994 – 1999)

Pros

- Automation, safety
- Non-intrusiveness
- Accurate analyses ∵
- Predictability
- Low break-even point
- Easy engineering
  - AST, compiler re-use
- Realistic applications
- Framework for CT/RT

Cons

- Complex declarations
- Slicing & re-plugging
- Fixed precision
- A posteriori control
- Code less optimized
- Limitations
  - BT precision, optimisation
- Prototype

Precision of the Analyses

[PEPM'97, SAS'97, TCS'00]

Analyses	Alias	Binding time
Interprocedural	✓	✓
Flow-sensitive	✓	✓
Context-sensitive	on-going work	✓
Return-sensitive	N.A.	✓
Use-sensitive	N.A.	✓
Field-sensitive	per struct type (or instance)	per struct type (or instance)

Challenges?

- Detecting specialization opportunities:
  - Existing code already hand-optimized
  - Little hope

Challenges

- Architecturing software for specialization
  - Development methodology
  - More quantitative prediction

- Declaring specialization
  - More automation: no slicing and plugging (guards)
  - Less inference, more checking: downgrade Tempo

- **Make the technology usable by humans**

Extra slides

Stages: **S** **D**

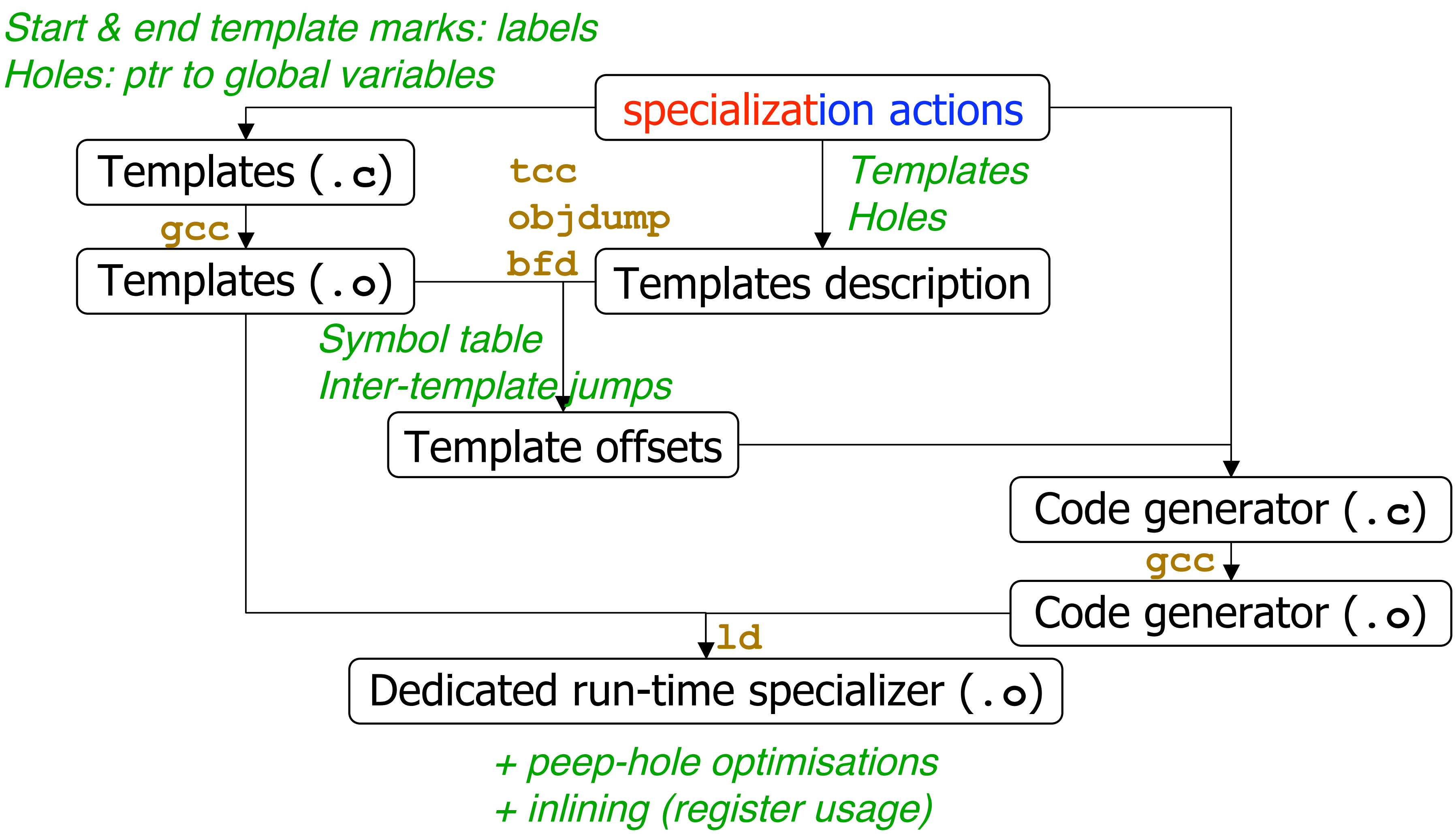
Making Templates

```
dotprod(size,u[],v[])
{
  res = 0;                                   T1
  for(i = 0; i < size; i++)
  {                   H1          H2
    res += u[i] * v[i];                      T2
  }
  return res;
}                                            T3
```

```
/* T1_start: */
dotprod(v[])
{
  res = 0;
T1_end:
  while( dummy ){
  T2_start:
    res += &h1 * v[&h2];
  T2_end:
  }
T3_start:
  return res;
}
/* T3_end: */
```

- *Re-use existing compiler*
- *Symbol table*
- *Original control flow*
- *Prevent inter-template code motion*

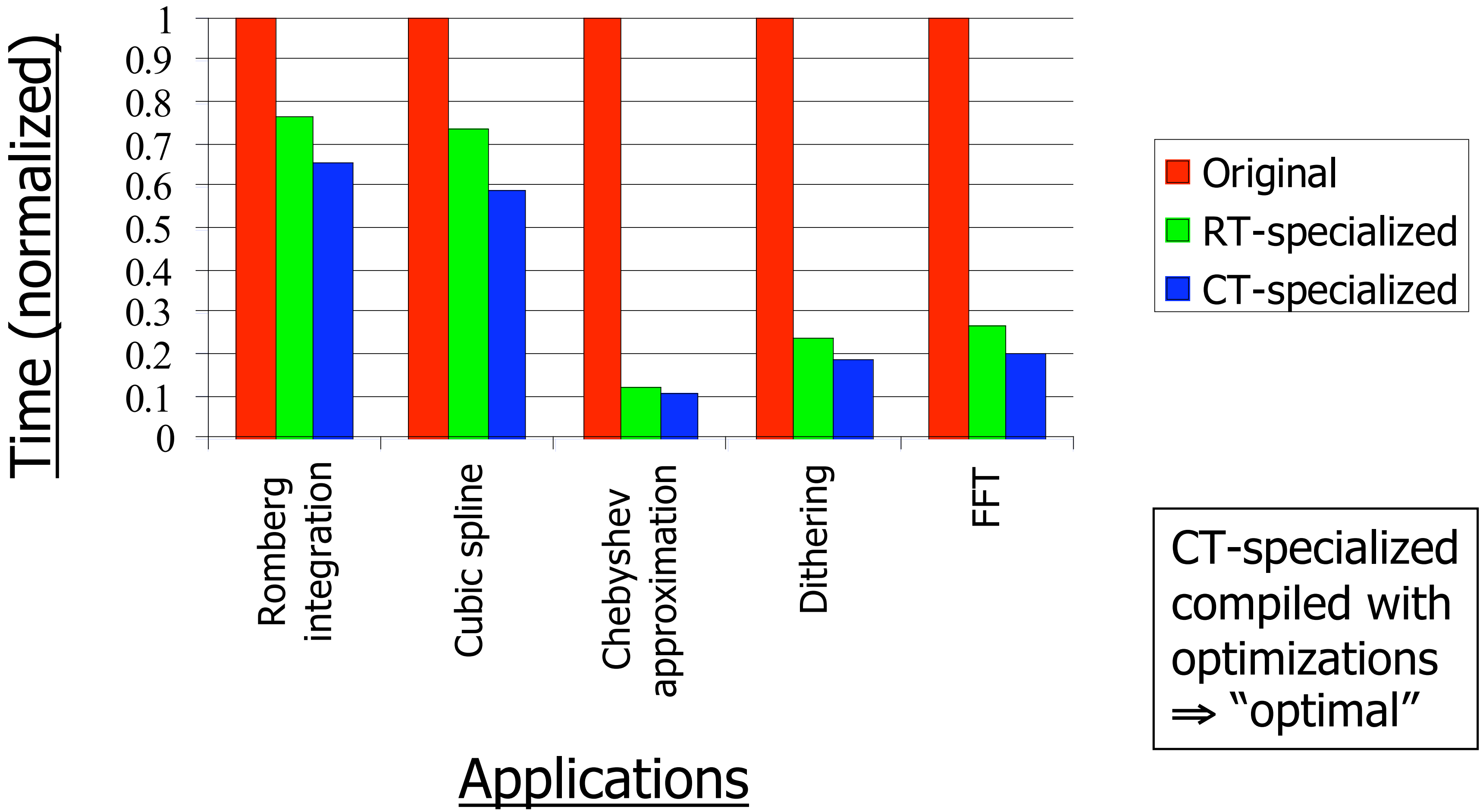
Generating
The Run-Time Specializer



Run-Time Specialization: Implementation

- Compilers: gcc, lcc
- Machines: Sparc, Pentium
- Main run-time cost: copying instructions
- Little inter-template optimizations
- Run-time inlining

Run-Time Specialization: Experimental Results



CT-specialized compiled with optimizations ⇒ "optimal"