LARA Tutorial

7. *Programming Strategies for Runtime Adaptivity*

Tiago Carvalho, Pedro Pinto, João Bispo, Ricardo Nobre, Luís Reis, and João M.P. Cardoso

University of Porto, FEUP, Porto, Portugal

April 20th, 2016

U.PORTO
FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

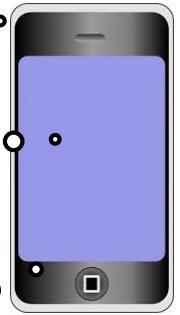
ANTAREX 10^{18}

SPeCS
Special-Purpose Computing Systems, languages and tools

Runtime Adaptation

Collision Detection?
- Fast detection
- Image quality/size

Device failure?
- Security measures
- Swap device



Low fuel/battery?
- Turn off devices
- Reduce image size

Image processing algorithm?
- Background noise
- Processing window
- Quality Requirement

Speedup process?
- Code Specialization
- Runtime compiler optimizations

Runtime Adaptation (2)

- Offline solution: compile-time specialization
  - Multiple versions generated offline – offline profiling information
  - Runtime selection
  - Impracticable in some situations

- Runtime compiler optimizations and code generation
  - Possible improvements – use of runtime information
  - Avoid code explosion
  - May impose unacceptable overhead

Runtime Adaptation in LARA

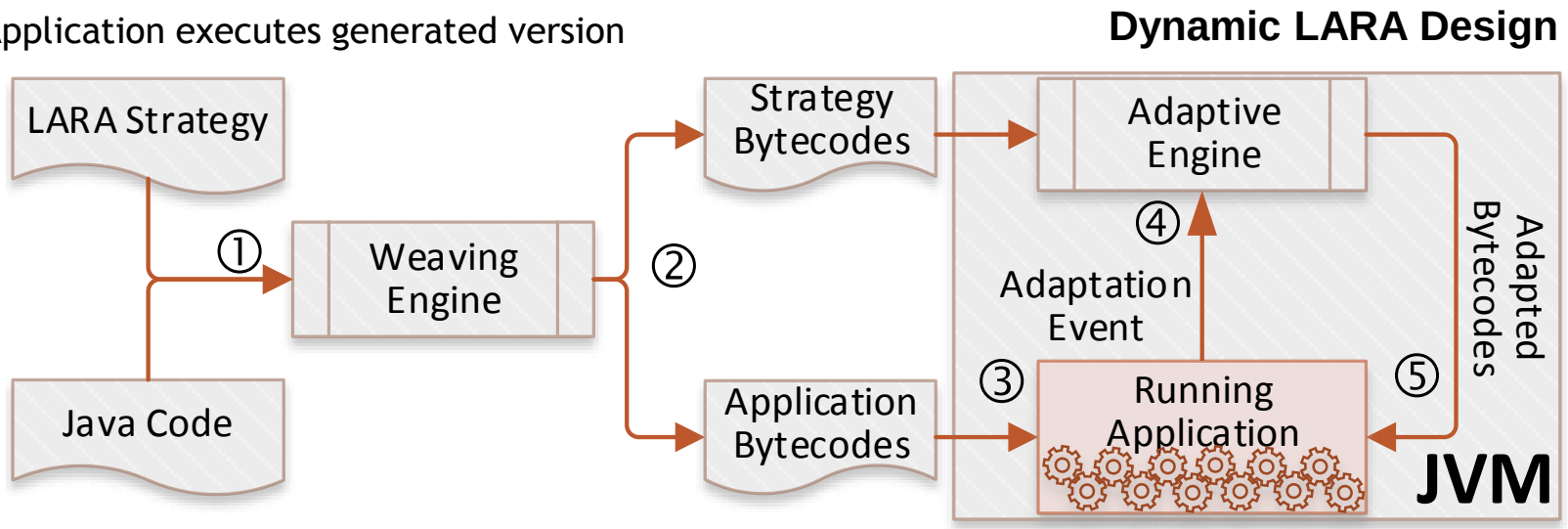
- LARA approach with runtime extensions
  - Access runtime information
  - Runtime strategy execution

- Adaptation based on:
  - Parameter tuning
  - Algorithm selection: based on predefined versions
  - Algorithm specialization:
    - selection between implementations
    - template-based code generation

- Triggered by:
  - Information attainable: when available/updated
  - Events: low battery, camera adjustment
  - Periodic adaptation: every 10 seconds, every 10 calls to a specific function

Runtime Adaptation Design

1. Target application + LARA strategies
2. Adapted application + Template-based generators
3. Application executed in the JVM
4. **Event**: generate specialized code
   - Strategy + Templates + Input
5. Application executes generated version

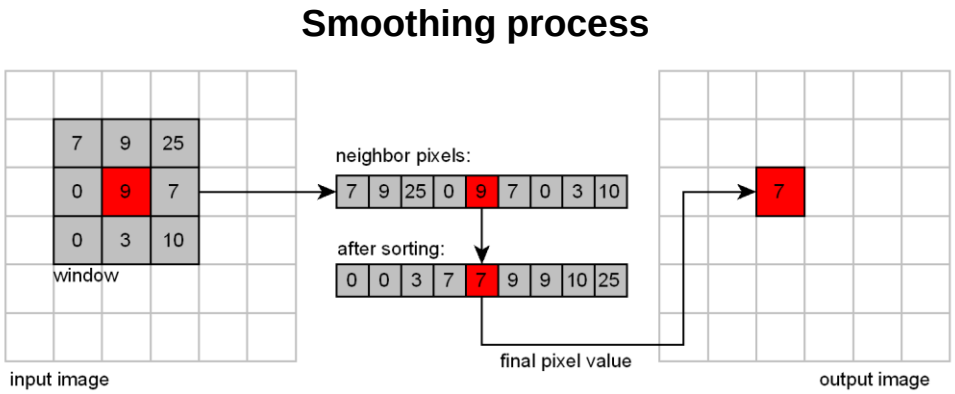


Experimental Setup

- Experiments with program specialization
- Environment
  - Ubuntu 13.10
  - Intel® Core™ i5 @ 3,20GHz * 4
  - 8GB RAM
  - Oracle JRE 1.8.0 11 Virtual Machine
- Benchmark
  - Median smooth filter (Hipr2[1])
  - Sobel (UTDSP[2])

[1] Hipr2 Library: http://homepages.inf.ed.ac.uk/rbf/HIPR2/
[2] UTDSP Benchmark Suite: http://www.eecg.toronto.edu/~corinna/DSP/infrastructure/UTDSP.html

Experimental Results: Median Smooth



✓Specialization Opportunity

1. Select best sorting algorithm
   - Window size
   - Values range (gray image: [0:255])
2. Apply compiler optimizations
   - Window size

Experimental Results: LARA Example

```
1  aspectdef BestMedianImpl
2    medianCall: select fcall{"median"} end
3    kernel: select method{"smooth"}.param{"kernelSize"} end
4    apply dynamic to medianCall::kernel
5      var bestMedian;
6      switch($param.value){
7        case 3:    bestMedian = "sorting_net"; break;
8        //...
9        case 7:    bestMedian = "counting_sort"; break;
10     }
11
12     run generator($fcall, bestMedian, $param.value);
13   end
14 end
15
```

According to *kernelSize*...

...select the best sorting template...

... and generate code based on *kernelSize*

Every call to smooth!

Experimental Results: LARA Example (2)

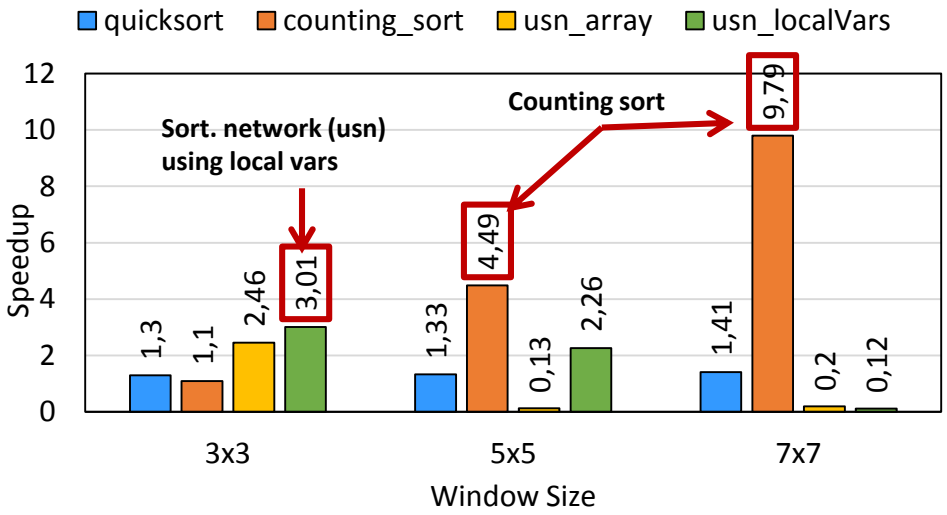
- Periodic adaptability

```
1  aspectdef BestMedianImpl
2    medianCall: select fcall{"median"} end
3    kernel: select method{"smooth"}.param{"kernelSize"} end
4    apply every 10 sec dynamic to medianCall::kernel
5      … // same as before
6    end
7  end
8
```

now we execute every 10 sec

Experimental Results: Median Calculation

- Speedups over original library implementation (Hipr2*)
- Adaptation allows changing to the best version



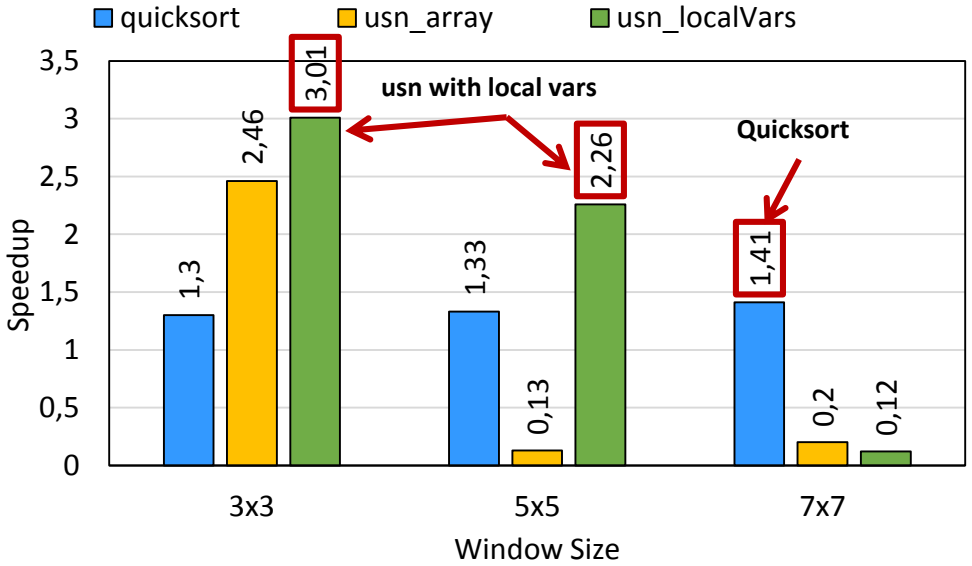
Templates:

- usn_array
  - All loops unrolled
- usn_localVars
  - All loops unrolled
  - Replace array accesses with local variables

*Hipr2, Image processing learning resources: http://homepages.inf.ed.ac.uk/rbf/HIPR2/

Experimental Results: Median Calculation

- For **input range > 256**, counting sort may not be feasible
- Strategies definition can conduct to the best speedup



Templates:

- usn_array
  - Full loop unroll
- usn_localVars
  - Full loop unroll
  - Scalar replacement

Experimental Results: Sobel

- Edge detection

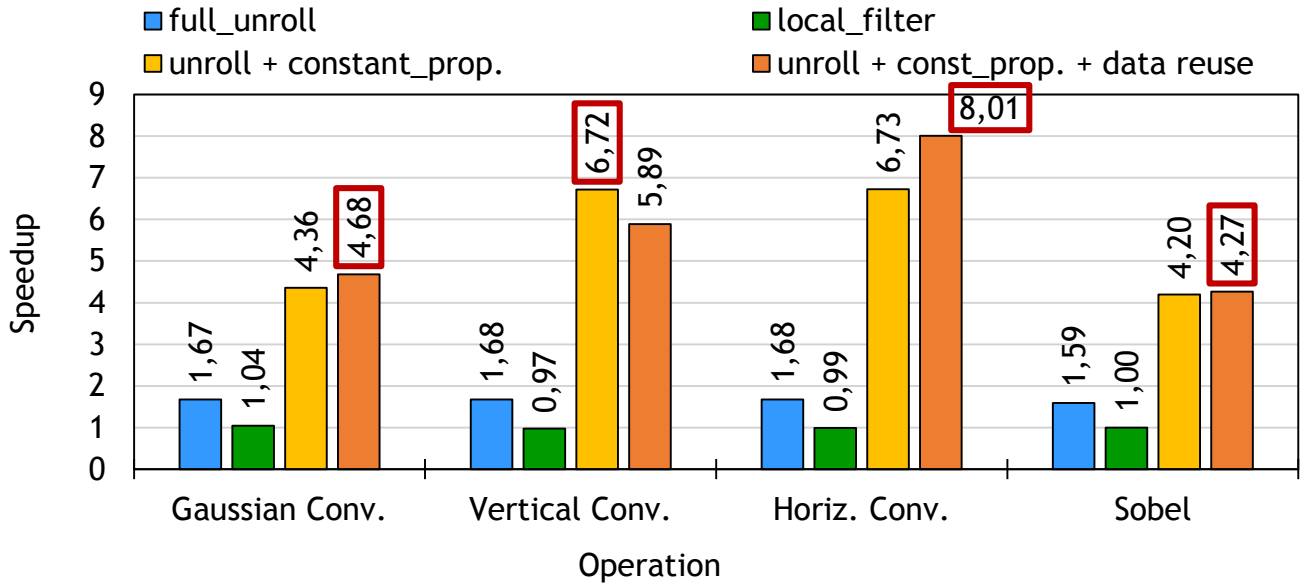
```
convolve(   gaussianCoeff); //smooth
convolve( horizontalCoeff);
convolve(   verticalCoeff);
```

✓Strategy
  - Different specialized versions for each convolution operation

- Templates:
  - full_unroll – fully unroll loops processing the coefficients array
  - local_filter - New version for each op
    - Insert coefficients array inside each new version (no loop unroll!)
  - constant_prop - Replace coefficients array accesses with corresponding constant values
  - data reuse - Reuse neighbor values in next iterations

Experimental Results: Sobel

- Speedups over the original Sobel (and convolution) version
- Example of unviable compile-time code generation
  - Coefficients array can take different values/size



Takeaway Points

- Specialization based on contextual information
  - Template-based code generation
- Adaptation: parameter, algorithm selection and specialization
- Experiments show benefits from code generation + runtime information