Programming Strategies for Contextual Runtime Specialization

Tiago Carvalho
t.carvalho@fe.up.pt

Pedro Pinto
p.pinto@fe.up.pt

João M. P. Cardoso
jmpc@acm.org
Outline

- Problem
- LARA approach for Runtime Adaptability
- Case Study
- Experimental Results
- Conclusions
Problem

- Applications may have to adapt
  - Contextual information and/or resource availability
  - Parameter tuning, algorithm selection, resource management, ...

- One approach is code specialization
  - Multiple versions generated offline
  - Runtime selection

- Runtime compile optimizations and code generation
  - Can achieve important improvements
  - May impose unacceptable overhead

- Idea: a template-based Java bytecode generation
  - Dynamically generate specialized versions
  - Reduce runtime overhead
  - Specialization strategy separated from the target code
Why Runtime Specialization?

- Achieve higher performance than original version
- Satisfy requirements, save energy, etc.
- Take advantage of contextual information
  - Program: parameters, data, ...
  - System: energy, environment, ...
  - Multi-target: sensors, devices, ...
- Select best algorithm implementation
- Apply optimizations with runtime information
The LARA Language

- Domain-Specific Language (DSL) with aspect-oriented programming (AOP) concepts
- Secondary concerns detached from application logic code
- Partially agnostic to: architecture, programming language and toolchain
- Strategies for instrumentation and synthesis/compiler optimizations
- Explore optimization sequences (DSE mechanisms)

Runtime Strategies in LARA

- LARA approach with runtime extensions
  - Access runtime information
  - Accomplish adaptation requirements

- Program adaptation based on:
  - Parameter tuning
  - Algorithm selection: based on predefined versions
  - Algorithm specialization: selection between implementations or template-based approach

- Runtime adaptation triggered by:
  - Information retrieval: 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
Case Study: Median Smooth

1. Use of sliding NxN window (3x3 in the figure)
2. Get neighbors
3. Sort elements
4. Take middle value (median) and assign to output pixel
5. Move window and repeat
Case Study: Median Calculation

- Find the maximum and remove
  - Repeat for half of the input values (Hipr2 benchmark)

- Or use a sorting algorithm
  - Quick Sort, sorting network, counting sort,

- Specialization Opportunity
  - Select best sorting algorithm
    - Window size and values range (gray image: [0:255])
  - Apply compiler optimizations
    - Window size
Case Study: LARA Example

- Select the best median implementation

```plaintext
aspectdef BestMedianImpl

medianFunc: select function{"median"} end
window: select class{"ImageUtils"}.field{"K"} end

apply dynamic to medianFunc::window
var bestMedian;
switch($field.value){
  case 9: bestMedian = "sorting_net.tpl"; break;
  //...
  case 49: bestMedian = "counting_sort.tpl"; break;
}
run replace($function, bestMedian, $field.value);
end
end
```

Select target method...
... and the decision point
According to the runtime information (window size)...
... select the best sorting template...
... and specialize with the runtime information
... now we adapt every 10 calls to median
Specialization Phase: Use of Template-Based Code Generators

- For a 3x3 window for the median smooth example
- Selected: odd-even sorting network of size 9
Template-Based Code Generator for Sorting Network

- Starting point Java code for the odd-even sorting network

```java
void sortNet(int[] values, int length){
    for(int i = 0, int init = 0; i < length; i++){
        for(int j=init; j < length-1; j+=2){
            if(values[j] > values[j+1]) {
                int temp = values[j];
                values[j] = values[j+1];
                values[j+1] = temp;
            }
        }
        init = 1-init; //for odd-even swaps
    }
    if(length % 2 == 0) { //length is even
        int middleLeft = values[length/2 - 1];
        int middleRight = values[length/2];
        return (middleLeft + middleRight)/2;
    }
    //8 lines omitted
    return values[length/2];
}
```

Template-Based Code Generator for Sorting Network

- Fixed size sorting network (9 elements)
- Specialized via loop unrolling

```c
void sortNet(int[] values, int length){
    for(int i = 0, init = 0; i < length; i++){
        for(int j=init; j < length-1; j+=2){
            if(values[j] > values[j+1]) {
                int temp = values[j];
                values[j] = values[j+1];
                values[j+1] = temp;
            }
        }
        init = 1-init; //for odd-even swaps
    }
    if(length % 2 == 0) { //length is even
        int middleLeft = values[length/2 - 1];
        int middleRight = values[length/2];
        return (middleLeft + middleRight)/2;
    }
    ... //8 lines omitted
    return values[length/2];
}
```

```c
void sortNet(int[] values, int length){
    if (values[0] > values[1]) {
        int temp = values[0];
        values[0] = values[1];
        values[1] = temp;
    }
    ... // +/-173 lines omitted
    if (values[4] > values[5]) {
        int temp = values[4];
        values[4] = values[5];
        values[5] = temp;
    }
    if (values[6] > values[7]) {
        int temp = values[6];
        values[6] = values[7];
        values[7] = temp;
    }
    return values[4];
}
```
Template-Based Code Generator for Sorting Network

- Specialized via loop unrolling + scalar replacement

```c
void sortNet(int[] values, int length){
    if (values[0] > values[1]) {
        int temp = values[0];
        values[0] = values[1];
        values[1] = temp;
    }
    //6 lines omitted
    if (values[4] > values[5]) {
        int temp = values[4];
        values[4] = values[5];
        values[5] = temp;
    }
    if (values[6] > values[7]) {
        int temp = values[6];
        values[6] = values[7];
        values[7] = temp;
    }
    return values[4];
}
```

```c
void sortNet(int[] values, int length){
    int value_0 = values[0];
    int value_1 = values[1];
    //6 lines omitted
    int value_8 = values[8];
    if (value_0 > value_1) {
        int temp = value_0;
        value_0 = value_1;
        value_1 = temp;
    }
    //6 lines omitted
    if (value_4 > value_5) {
        int temp = value_4;
        value_4 = value_5;
        value_5 = temp;
    }
    if (value_6 > value_7) {
        int temp = value_6;
        value_6 = value_7;
        value_7 = temp;
    }
    return value_4;
}
```
Experimental Setup

- 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)
  - Sobel (UTDSP)

- Adaptation
  - Java Agents for code redefinition
  - “tools.jar” from JDK
  - Jasmin for class-file generation from JVM code

Experimental Results: Median Calculation

- Speedups over original library implementation (Hipr2: Image processing learning resources)
- Using quicksort, sorting networks (usn) and counting sort

### Runtime Adapt. Mechanism:
- quicksort
  - Alg. Selection
- counting_sort
  - Alg. Selection
- usn_array
  - Alg. Selection
  - Full loop unroll
- usn_localVars
  - Alg. Selection
  - Full loop unroll
  - Scalar replacement

Experimental Results: Median Calculation

- For **input range > 256**, counting sort may not be feasible

![Bar chart showing speedup and window size for different algorithms.](chart.png)

- Runtime Adapt. Mechanism:
  - quicksort
    - Alg. Selection
  - usn_array
    - Alg. Selection
    - Full loop unroll
  - usn_localVars
    - Alg. Selection
    - Full loop unroll
    - Scalar replacement

Experimental Results: Sobel

- Same convolution method with different input coefficients
  1. Gaussian Convolution (smoothing)
  2. Vertical Convolution
  3. Horizontal Convolution

Optimizations:
- local_filter
  - New version for each op.
  - Move coef. array inside each new version
- constant_prop.
  - Replace coef. array accesses with corresponding values
- data reuse
  - Reuse neighbor values in next iterations
Experimental Results: Overhead

- > overhead experienced
  - Small number of images
  - Small window size
- Inefficient in some cases
  - 1 image for the Sobel operation
- > number of images
  - Increased speedups
    - 7x7 kernel: from 5.61x to 10.15x
  - Overhead shadowed by the speedup benefits
### Related Work: Selected Similar Approaches

<table>
<thead>
<tr>
<th>Approach</th>
<th>Overview</th>
<th>Focus</th>
<th>Strategy</th>
<th>Programming Model</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ADAPT</strong></td>
<td>Application-specific runtime systems</td>
<td>Runtime versioning</td>
<td>target application + available optimizations + heuristics</td>
<td>DSL</td>
</tr>
<tr>
<td>[Voss et al., PPoPP, 2001]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>deGoal</strong></td>
<td>Kernels in fast binary code generators</td>
<td>Specialization</td>
<td>“Compilettes” optimized to current program/system</td>
<td>C extended with high-level assembly</td>
</tr>
<tr>
<td>[Charles et al., CC, 2014]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Elastic Computing</strong></td>
<td>Multiple specialized implementations of a function</td>
<td>Algorithm selection</td>
<td>Library of. functions + impl. planning tool + runtime env. system</td>
<td>Library of specialized functions</td>
</tr>
<tr>
<td>[Wernsing et al., LCTES, 2010]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>PetaBricks</strong></td>
<td>Implicit parallel language based on algorithmic choice</td>
<td>Parallel prog. Alg. Selection</td>
<td>Mult. alg. spec. = Gen. of hybrid algorithms</td>
<td>Parallel programming language</td>
</tr>
<tr>
<td>[Ansel et al., PLDI, 2009]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>LARA Runtime</strong></td>
<td>AOP approach for runtime adaptation</td>
<td>profiling, versioning, alg. selection, specialization</td>
<td>target application + strategies + templates + runtime info.</td>
<td>DSL (LARA language)</td>
</tr>
</tbody>
</table>
Conclusions

- Runtime specialization based on contextual information
  - Template-based code generation
- Aspect-Oriented Programming approach
  - Domain-Specific Language to program adaptation strategies
- Preliminary experiments reinforce the benefits of code specialization and a template-based code generation approach using contextual information

- Ongoing and Future Work
  - Study different low-overhead techniques
  - Search examples suitable for template-based approach
  - Automate code generation from the LARA specifications and the weaving process
  - Test in different target platforms
THANK YOU!

Questions?

Tiago Carvalho
t.carvalho@fe.up.pt

LARA Wiki: http://fe.up.pt/lara

Acknowledgments