Towards Automatic Tuning of Adaptive Computations in Autonomic Middleware

Ying Zhang, Gang Huang
Xuanzhe Liu, Zizhan Zheng, and Hong Mei

http://sei.pku.edu.cn/~zhangying06

Peking University
2010-11-30
Outline

- The Resource Competition Problem
- Solution
- Evaluation
- Future Work
Autonomic Middleware

- An autonomic middleware performs adaptive computations for self-management of the system
- Adaptive Computation (AC) by IBM, “An architectural blueprint for autonomic computing”
Resource Competition! - The Old Challenge

- **PKUAS**
- **The three concerned ACs in PKUAS**
  - RTM (Response Time Monitor)
  - LP (Log Processor)
  - TPA (Thread Pool Adjuster)
- **ECperf – the test driver**

- **Two computing entities in an autonomic middleware**
  - Business functions
  - Adaptive computations (ACs)
Resource Competition – The Solutions

 infectious

 Solutions

 ➢ Think about computing runs on resources (e.g., CPU and Memory)
 ➢ Bottom-Up
   • Controlling resource allocation
     • E.g., provide fixed amount of memory to a process
     • Usually general enough for all computing
 ➢ Top-Down
   • Controlling resource consumption
     • E.g., stop/freeze a specific computing
     • Should be computing-specific

 Our solution

 ➢ Top-Down: autonomic-specific
Approach Overview

- Using an orchestrating tuner to tune the computation level of an AC (i.e., autonomic computation)
# Tuning Model

## Tuning policies

<table>
<thead>
<tr>
<th>Function</th>
<th>Triggering</th>
<th>Tuning Policy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Monitor</td>
<td>Time driven; Tuner driven</td>
<td>Start/Stop</td>
</tr>
<tr>
<td>Analyze</td>
<td>Time/monitor driven; Tuner driven</td>
<td>Start/Stop</td>
</tr>
<tr>
<td>Plan</td>
<td>Analyzer driven; Tuner driven</td>
<td>Start/Stop</td>
</tr>
<tr>
<td>Execute</td>
<td>Planner/analyzer driven; Tuner driven</td>
<td>Start/Stop, or tune the execution cost</td>
</tr>
</tbody>
</table>

## The valid state of an AC

<table>
<thead>
<tr>
<th>State</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>S₀</td>
<td>All the four functions in the control are stopped</td>
</tr>
<tr>
<td>S₁</td>
<td>Only the <em>monitor</em> function is started</td>
</tr>
<tr>
<td>S₂</td>
<td>S₁ is true and the <em>analyze</em> function is started</td>
</tr>
<tr>
<td>S₃ (E₀)</td>
<td>S₂ is true and if a change is necessary, the <em>execute</em> function is started</td>
</tr>
</tbody>
</table>
Computation Level

- **Definition**
  - The relatively **maximum allowed completion-level** in an iteration of an AC’s corresponding control loop
  - It is identified using the **highest state allowed to occupy** in an iteration of the loop

- **The resource costs of an AC in different levels can be sorted in ascending order as follows:**
  \[ S_0 < S_1 < S_2 < S_3 (E_0) < E_1 < \ldots < E_n \]
  - **Upgrade**: level-changing in the left-to-right order
  - **Degrade**: right-to-left order

- **Tuning (i.e., upgrading or degrading)**
  - Setting a proper **highest allowed state** for the control loop
    - Based on the workload and the features and gains of the AC
Tuning Multiple Adaptive Computations

- **Adaptation gains of an AC**
  - The relative importance and cost-effectiveness of an AC with regard to business performance
  - Please note:
    - The gains brought by executing an AC are not limited to business performance
    - The basic goal of our approach is to guarantee business performance when resources are limited and competed

- **Obtaining the gain value**
  - Autonomic
    - By profiling
    - By considering the maturity Level
  - Manual
  - Gains are used to compare different ACs, but not for exactly evaluating an AC
Pinned Level and SafeRange

- **Pinned Level**
  - An AC never wants its level to be changed by the tuner
  - Once the AC reaches its *pinned level*, any tuning operations on it will have no effect

- **SafeRange**
  - The importance of an AC to business performance often relies on the states of the managed objects, and thus changes at runtime (e.g., the garbage collector)
  - We assume that each managed object of an AC has a *SafeRange*
    - E.g. the ratio between the free threads and the total threads in the threadpool should be higher than 5%
  - If *SafeRange is violated*, the corresponding AC should upgrade or degrade by itself without the control of the tuner
Two Concerns in Tuning Algorithm Design

**Priority**
- The essence of AC-tuning is a scheduling problem
- Use the gain value of each AC as its priority of tuning
- By considering the features of AC, the priority concern can be demonstrated at two representative granularity levels
  - The single state of AC (i.e., fine-grained)
  - The AC as a whole (i.e., coarse-grained)

**Fairness**
- How to demonstrate priority upon fairness is challenging
- By considering the features of AC, the fairness concern can be demonstrated at two representative granularity levels
  - The single state of AC (i.e., fine-grained)
  - The AC as a whole (i.e., coarse-grained)
An AC with a high gain value is upgraded before and degraded after those ACs with relatively low gain values.

The new level of an AC must be lower than or at most equal to the lowest level of those ACs with higher gain values; vise versa.

The tuner must not upgrade/degrade the same AC again until it upgrades/degrades all the other ACs.
Tuning Algorithm: CCTT

- An AC with a high gain value is upgraded before and degraded after those ACs with relatively low gain values.
- Before an AC being upgraded, each of the other ACs with higher gain values must have been upgraded to its highest level (e.g., S3), and all the other ACs with lower or equal gain values must not be tuned until the upgrading of the current AC to its highest level is finished; vise versa.
Tuning Algorithm: NLAT

- If an AC is upgraded before the other ACs, it should be degraded before them; vise versa.
- If an AC is upgraded, the new level of this AC must be lower than or at most equal to the lowest level of those ACs upgraded before it; vise versa.
- The tuner must not upgrade/degrade the same AC again until it upgrades/degrades all the other ACs.
If an AC is upgraded before the other ACs, it should be degraded before them; vise versa.

Before an AC being upgraded, each of the other already upgraded ACs must have been upgraded to its highest level (e.g., $S_3$), and all the other remain ACs must not be tuned until the upgrading of the current AC to its highest level is finished; vise versa.

---

**Tuning Algorithm: NCTT**

---

The NCTT Algorithm

NCTT: Non-covered Carry-Through Tune

---

If an AC is upgraded before the other ACs, it should be degraded before them; vise versa.

Before an AC being upgraded, each of the other already upgraded ACs must have been upgraded to its highest level (e.g., $S_3$), and all the other remain ACs must not be tuned until the upgrading of the current AC to its highest level is finished; vise versa.
The Atomic Tuning Procedure

- $S_0$: $sl < low \&\& sr(o)$
- $S_1$: $sl > high \&\& sr(o)$
- $S_2$: $sl < low \&\& sr(o)$
- $S_3$: $sl > high \&\& sr(o)$
Implementation: The Tuning Framework

- The ACTuner is the essential component of the tuning framework
- The ACs
  - Interfaces requirements
  - Behavior requirements
Evaluation
Experiment Setup

```xml
<Service class="cn.pku.edu.as.ac.OrchestratingACTuner" name="Tuner:name=ACTuner">  
  <attribute name="Type" value="ACTuner" />
  <attribute name="Metric" value="CPU" />
  <attribute name="Threshold" value="75-85" />
  <attribute name="MonitorInterval" value="5" />
</Service>

<Service class="cn.pku.edu.as.ac.ResponseTimeMonitor" name="ACService:name=RIM">  
  <attribute name="Type" value="AC" />
  <attribute name="Gain" value="10" />
  <attribute name="InitialLevel" value="1" />
</Service>

<Service class="cn.pku.edu.as.ac.LogProcessor" name="ACService:name=LP">  
  <attribute name="Type" value="AC" />
  <attribute name="Gain" value="20" />
  <attribute name="InitialLevel" value="3" />
</Service>

<Service class="cn.pku.edu.as.ac.ThreadPoolAdjuster" name="ACService:name=TPA">  
  <attribute name="Type" value="AC" />
  <attribute name="Gain" value="30" />
  <attribute name="InitialLevel" value="3" />
  
  - <SafeRange id="ThreadSafeRange">
    
    - <Monitors>
      <Monitor class="cn.edu.pku.as.monitor.ThreadBusy" id="monitor_tb" name="monitor_tb" />
      <Monitor class="cn.edu.pku.as.monitor.ThreadCount" id="monitor_tc" name="monitor_tc" />
    </Monitors>
    
    - <SafeMeasures>
      
      - <Measure monitorref="monitor_tb" compare="less_equal" weight="5">
        <ComparableMonitor monitorref="monitor_tc" scale="0.95" />
      </Measure>
      </SafeMeasures>
  </SafeRange>
</Service>
```
Experimental Results

Fig. The txRate sequence

Fig. The CPU utilization (txRate=6)

With ACTuner

Without ACTuner
Experimental Results (2)

Response Time of 90% New Order (in seconds)

<table>
<thead>
<tr>
<th>txRate</th>
<th>No AC Runs</th>
<th>ACs Run (without ACTuner)</th>
<th>ACs Run (with ACTuner)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>0.5</td>
<td>0.8</td>
<td>0.9</td>
</tr>
<tr>
<td>4</td>
<td>0.9</td>
<td>1.4</td>
<td>1.1</td>
</tr>
<tr>
<td>5</td>
<td>1.2</td>
<td>2.1</td>
<td>1.5</td>
</tr>
<tr>
<td>6</td>
<td>1.7</td>
<td>2.6</td>
<td>1.9</td>
</tr>
<tr>
<td>7</td>
<td>3.4</td>
<td>3.7</td>
<td>2.5</td>
</tr>
<tr>
<td>8</td>
<td>5.8</td>
<td>4.6</td>
<td>3.2</td>
</tr>
<tr>
<td>9</td>
<td>9.3</td>
<td>5.4</td>
<td>3.9</td>
</tr>
<tr>
<td>10</td>
<td>13.9</td>
<td>6.3</td>
<td>4.8</td>
</tr>
</tbody>
</table>
Experimental Results (3)

(a) No SafeRange constraints are violated (txRate=6)

(b) TPA violates its SafeRange constraints (txRate=7)
Experimental Results (4)

The runtime throughputs graph
The Experiment Shows

- **The significance of the resource competition problem**
  - In a resource limited environment, ACs often compete with business functions for resources, which can interfere with the processing of business requests when the system load is high.

- **The effectiveness of the proposed approach**
  - The tuning framework is effective to change the computation levels of different ACs on the fly.
  - Thus it is efficient to control the resource costs of these computations for improving the business performance of the autonomic system as a whole.
Discussion

- The autonomic computing model and tuning policies are relatively simple
  - More fine-grained tuning other than MAPE?

- Tune business functions rather than ACs?
  - Although such a practice is much more application specific and lacks a unified model for tuning (which may prevent their widespread use), it is indeed a way to guarantee business performance.

- The trade-off between different adaptive computations
  - There are more different strategies
  - Which one is better? An Open Issue
Conclusion

- With the proliferation of the autonomic computing technology, systems such as middleware are implemented with many adaptive computations.
- It brings new challenges to resource allocation for achieving the overall business performance goals because of resource competition between business functions and adaptive computations.
- We have presented an automatic tuning approach to supporting a flexible tradeoff between business functions and adaptive computations by tuning the computation levels of the latter dynamically when resources are limited and competed.
- The experiment proves the effectiveness of the implemented tuning framework.
Thank You!

Question?