Simplifying Cyber Foraging

Rajesh Krishna Balan
Computer Science Department
Carnegie Mellon University

Motivation: mobile interactive applications
- speech recognition, language translation, augmented reality, ...
  - Resource-heavy, but need bounded response time
    - Unfortunately, handhelds are weak!!

Motivation: Handhelds are weak!
- Resource intensive
- Huge Data Sets
- 2 GHz, 1 GB, 3-D graphics
- 2 GB of data
- 200 MHz, 32 MB, no 3-D, no FPU
- 32 MB Flash

Solution: Cyber Foraging
- "To live off the land"
  - Use resources in environment
- Remote execution
  - Augment device capabilities
- Fidelity Adaptation
  - Reduce resource usage
- OS Component of Aura

Problem #1
- How do we get large applications to work on small devices?
- Made more difficult by the mobile domain
  - Bandwidth fluctuates wildly
  - Battery concerns require variable operating modalities
    - Conserve battery at expense of quality and latency
    - Vice versa

Problem #2
- Seems likely that middleware can be built
  - With enough sweat, engineering, and plumbing
- But, what about the applications??
Problem: Retargeting Applications

- Large number of useful apps already exist
  - Complicated and large (> 100K lines of code)
  - Infeasible to write them from scratch
- Must be able to quickly retarget apps
  - Solution should be language independent
  - State of the art is 2 - 4 weeks per application

Thesis Statement

- It is possible to easily, quickly, and effectively modify an important class of existing computationally-intensive applications, such as language translators, speech recognizers, face detectors, and graphics applications, for cyber foraging.

Strawman Solutions

- Run all applications on remote server
  - No application modifications needed
  - Use ssh or vnc to access server
  - Network latency is a problem
  - Requires bandwidth
  - No degraded / corrective failure mode
- Run all applications locally
  - Not enough resources

Roadmap

- Motivation & Background
- Solution Requirements
- Systems + Software Engineering Solution
- Evaluation
- Conclusion

Requirement No. 1

- Support as many applications as possible
  - No restriction on language
    - Must support C, C++, Java, .NET, etc.
  - No restriction on programming style
    - Procedural, functional, event-based (OpenGL), etc.
  - No restriction on type of application
    - Language translators, face detectors, graphics apps

Requirement No. 2

- Target Entry-level Application Developers
  - Effectively fresh hires right out of college
    - Representative of who actually does this work
  - Assume no expert knowledge of anything
    - Basic programming skills assumed
Requirement No. 3

- Solution must be good
  - Retargeting time must be low
    - < 6 hrs per application
    - Current time is ~ 2 to 4 weeks per application
- Retargeted application is isolated
  - From runtime / OS / hardware changes
  - Vital for fast moving mobile market
- Performance must be excellent
  - Comparable to hand-retargeted applications

Summary of Requirements

- Any application
- Any developer
- Fast
- Good

Sounds Impossible

Key Insight

Application aspects relevant for remote execution and fidelity adaptation can be expressed in a short declarative form

- Expected Application Resource Usage
  - Use concept of parameters
- Runtime adaptive aspects of application
  - Application runtime settings => fidelity variables
  - Remote execution partitionings => tactics

Roadmap

- Motivation & Background
- Solution Requirements
- Systems + Software Engineering Solution
- Evaluation
- Conclusion

Application Domain

Applications must satisfy three criteria

1. Useful Mobile Application
   - Language translation, speech recognition etc.
2. Resource Constrained wrt. Mobile Devices
   - Eliminates email, calendar etc.
3. Interactive
   - User provides input, application does work. Repeat
   - Eliminates media streaming type applications

Design Considerations

- Achieve excellent mobile performance
  - Make dynamic decisions
  - Requires resource measurements, resource predictions, "optimal" solver
  - Large amount of extra code needed for each application
- Keep retargeting cost low
  - Minimize application modifications
  - Abstract as much as possible
  - Place common functionality in common layers
Solution Design

• Coarse-grained remote execution
  – Modular level (RPCs)
  – Finer grain requires language support

• Cannot wrap around existing APIs
  – Application / language independent
  ➔ Applications must be hand-modified
    - Cannot use generic code analysis techniques

Design Considerations

• Achieve excellent mobile performance
  – Make dynamic decisions
  – Requires resource measurements, resource predictions, “optimal” solver
  ➔ Large amount of extra code needed for each application

• Keep retargeting cost low
  – Minimize application modifications
  – Abstract as much as possible
  – Place common functionality in common layers

Components of Solution

1. Language for describing applications
   • Vivendi

2. Adaptive Runtime System
   • Chroma
   • Handles common adaptation requirements

3. Smart Stub Generator
   • Generates most of the app to runtime interface code

4. Well-defined procedure for modifying app

Overview of Chroma

(Provided by application)

Tactics
description
Utility
Function
Core functionality
Tactics
selection
engine
Selected tactic
Operation
executor

Utility
Function
User-specific
knowledge

Resource
availability

Predicted
resource usage

Resource
Monitors

Resource
Demand
Predictor

Log file

Vivendi

APPLICATION pangloss-lite;

IN int num_words;
OUT float quality;

/* RPC Specifications */

Language Language
modeler
Dictionary
Example based

/* Tactics (Useful Ways to Combine the RPCs) */

RPC server_dict (IN string line, OUT string dict_out);
RPC server_ebmt (IN string line, OUT string ebmt_out);
RPC server_lm (IN string gloss_out, IN string dict_out,
                IN string ebmt_out, OUT string translation);

/* Porting Applications */

(1) Collaborate
Domain expert

(2) Sub generator
Application description file

(3) Modify
Modified source code

(4) Compiler
Application source code

APPENDIX
Client API Calls

- Basic
  - Register: Register app with runtime
  - Cleanup: Remove app from runtime

- Core Functionality
  - Find_fidelity: Asks runtime to decide appropriate runtime settings
  - Do_tactics: Perform the operation

Roadmap

- Motivation & Background
- Solution Requirements
- Systems + Software Engineering Solution
- Evaluation
  - Chroma
  - RapidRe
- Conclusion

Chroma: Evaluation Objective

- To show that Chroma has comparable performance to an oracle
  - Oracle selects best tactic for current environment
  - Oracle’s selection is determined offline while Chroma selects online

- Is the overhead of Chroma acceptable?

Applications Used

- Three real research applications
  - Useful for mobile users
  1. Pangloss-Lite is a natural language translator (7 tactics)
  2. Janus is a speech recognizer (2 tactics)
  3. Face detects faces in images (1 tactic)

Experimental Setup

- Thinkpad 560X Client (200 Mhz Pentium)
  - Representative of fastest handhelds
- HP Omnibook 6000 Servers (1 Ghz Pentium 3)
  - Unloaded except where stated otherwise
- 100 Mb/s Ethernet
- Testing methodology
  - Different inputs representing an operation
  - Each result average of 20 runs
  - State of system reset before each run

Tactics Don’t Hurt

- Application: Janus, Metric = \( \text{fidelity} / \text{latency} \)
Tactics Don’t Hurt

<table>
<thead>
<tr>
<th>Utterance</th>
<th>Oracle</th>
<th>Chroma</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Latency (s)</td>
<td>Fidelity</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0.71</td>
<td>0.50</td>
<td>0.73</td>
</tr>
<tr>
<td>2</td>
<td>1.00</td>
<td>0.50</td>
<td>1.00</td>
</tr>
<tr>
<td>3</td>
<td>0.76</td>
<td>0.50</td>
<td>0.80</td>
</tr>
<tr>
<td>4</td>
<td>0.78</td>
<td>0.50</td>
<td>0.79</td>
</tr>
<tr>
<td>5</td>
<td>0.90</td>
<td>0.50</td>
<td>1.03</td>
</tr>
<tr>
<td>6</td>
<td>0.95</td>
<td>0.50</td>
<td>0.98</td>
</tr>
<tr>
<td>7</td>
<td>0.70</td>
<td>0.50</td>
<td>0.71</td>
</tr>
<tr>
<td>8</td>
<td>1.30</td>
<td>0.50</td>
<td>1.22</td>
</tr>
<tr>
<td>9</td>
<td>0.77</td>
<td>0.50</td>
<td>0.77</td>
</tr>
<tr>
<td>10</td>
<td>0.99</td>
<td>0.50</td>
<td>0.99</td>
</tr>
</tbody>
</table>

Application: Janus, Metric = \frac{\text{fidelity}}{\text{latency}}

Location: Remote in all case

Overhead Concerns

• Does the solver take too long?
  – Can it handle a large number of tactics?
  – What happens when the number of servers increases?

• Complete system overhead
  – How long does the system need to make decisions on a slow client?
    • Includes solver overhead + overhead of resource measurement & prediction

Overhead of Solver

- Synthetic results
- Slow client
- Max overhead < 0.9 ms
- Okay for interactive apps

Overhead of Entire System

- Pangloss – Lite
- Slow client
- Overhead is high – Between 0.5 – 0.8s

Breaking Down the Overhead

- Measurers are slow
  – Caching helps
  – Tradeoff accuracy for latency

Chroma: Summary

- Achieves close to optimal performance
  – Errors due to bad resource measurements or predictions

- Overhead is reasonable
  – Brute force computation is okay
  – Resource values should be cached for optimal performance
### Other Results In Thesis

- **Overprovisioned Environments**
  - Chroma can automatically use extra servers to improve application performance
- **Integration with dynamic user preferences** (Prism)
- **Security**
  - Simple mechanisms to detect rogue servers
- **Fairness** (In Cheng’s thesis)

### Roadmap

- **Motivation & Background**
- **Solution Requirements**
- **Systems + Software Engineering Solution**
- **Evaluation**
  - Chroma
  - RapidRe
- **Conclusion**

### RapidRe: Evaluation Objectives

To show that RapidRe is:

- **Quick**
  - Retargeting time is low. < 6 hrs per app
- **Easy**
  - Users don’t find the retargeting task difficult
- **Effective**
  - Retargeted apps equivalent to hand modified versions

### Evaluation Methodology

- **Conduct rigorous software usability study**
  - Large number of real apps
  - Large number of representative developers
    - 13 Senior undergrads
    - Measure time taken
    - Measure number of errors in solutions
- **Conduct systems evaluation**
  - Compare user study apps with expert apps
    - Expert apps are hand retargeted for the same runtime

### Test Applications

<table>
<thead>
<tr>
<th>App</th>
<th>Lines of Code</th>
<th>File Count</th>
<th>Language</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>Face</td>
<td>20K</td>
<td>105</td>
<td>Ada / C</td>
<td>Face Recognizer</td>
</tr>
<tr>
<td>Flite</td>
<td>570K</td>
<td>182</td>
<td>C</td>
<td>Speech Synthesis</td>
</tr>
<tr>
<td>GLVU</td>
<td>25K</td>
<td>155</td>
<td>C++ / OpenGL</td>
<td>3D Model Viewer</td>
</tr>
<tr>
<td>GOCR</td>
<td>30K</td>
<td>71</td>
<td>C++</td>
<td>Char Recognition</td>
</tr>
<tr>
<td>Janus</td>
<td>126K</td>
<td>227</td>
<td>C / Tcl / Motif</td>
<td>Speech Recognizer</td>
</tr>
<tr>
<td>Music</td>
<td>9K</td>
<td>55</td>
<td>Java / C++</td>
<td>Music Recognizer</td>
</tr>
<tr>
<td>Panlite</td>
<td>150K</td>
<td>349</td>
<td>C++</td>
<td>Language Translator</td>
</tr>
<tr>
<td>Radiator</td>
<td>65K</td>
<td>213</td>
<td>C++ / OpenGL</td>
<td>3D Lighting Models</td>
</tr>
</tbody>
</table>

### Procedure

- **Train each participant for 1 hr**
  - Using GOCR
- **Measure the time needed for them to perform each stage of the process**
  - Stage 1: create tactics file
  - Stage 2: create client component
  - Stage 3: create server component
RapidRe: Summary 1

- Reduced modification times from weeks to under 4 hrs
- Very few errors in solutions (not shown)
- But what about performance of modified apps?
  - Test modified apps against expert created apps
    - Under various mobile computing scenarios (not listed)

Performance of Modified Apps

- Tactics files had few errors (not shown)
- Client App performance shown
  - All server apps had equivalent performance

RapidRe: Summary 2

- Performance is also good
  - 25 / 25 server apps correct
  - 16 / 25 client apps correct
    - Not perfect but this was expected
- Performance problems due to two factors
  - Failure to specify application resource usage
  - Failure to use returned fidelity values
    - Measures can be taken to reduce these errors

Other Results In Thesis

- Detailed Breakdown of Errors
- Additional User-Perceived Results
  - Amount of Isolation
  - Effect of Experience
- Detailed Process Analysis and Suggestions for Improvements
Roadmap

• Motivation & Background
• Solution Requirements
• Systems + Software Engineering Solution
• Evaluation
• Conclusion

Key Challenges and Solution

• Any language
  – Runtime written in C

• Any developer
  – Vivendi + well-defined API insertion rules + proper isolation + good documentation

• Fast
  – Proper abstraction + stub generator + good documentation

• Easy
  – Proper abstraction + isolation + good documentation

• Effective
  – Chroma + Prism Integration

Thesis Requirements Summary

• Any application ➔ Yes!
• Any developer ➔ Yes!
• Fast ➔ Yes!
• Good ➔ Mostly (can be Yes!)

➔ There must be a catch

Reasons for Success

• Exploiting previously unknown similarities common to all the applications
  – Commonly used software development methods encourage proper modularization

• Naturally causes the "thin waistband" needed for this method
  – Facilitates high level description
  – Makes it easy for novices to add APIs
  • Interactive app model facilitates this as well

Only Part of the Solution

• 4 big missing pieces still needed
  – Integrating user preferences
    • Work by Sousa and Poladian (Aura project)

  – Application hint modules
    • Map fidelities / resources to application quality

  – Create appropriate UIs for a mobile device
    • Work by Eisenstein et al. and Myers et al.

  – Actual test deployment

Some Related Work

• Dynamic runtime systems
  – Odyssey [Noble], Puppeteer [deLara], Rover [Joseph], Coign [Hurd]

• Little Languages
  – Make [Feldman], QML [Frolund], QDL [Loyal], Aspect Languages [Kiczales]

• Retargeting Frameworks
  – IBM’s WebSphere, Microsoft’s .NET Framework
Summary

- Big challenge in modern computing
  - Design high performance systems that are provably usable
- Systems are usually fast enough
- However, they are frequently not usable
- This is a concerted attempt to bridge the gap
  - Using techniques from very diverse areas

The End

- Questions?

Benefits of Solution

- Developer describes app using Vivendi
  - Knowledge of small part of app needed
- Stub creates code to interface with Chroma
  - Knowledge of Chroma unnecessary
  - Easy well defined process to insert APIs into app
- App APIs are runtime / OS agnostic
  - Only stub generator needs to be modified
    - Apps just need to be recompiled with new stub code
    - Similar to glibc changes

Didn’t Talk About

- Chroma Runtime System in detail
- Other research
  - Distributed infrastructure for supporting massively multiplayer games
  - Virtual machine techniques to improve scientific applications
  - Changes to TCP's protocol
    - Improve wireless network performance
    - Reduce power consumption

App-Inserted API Calls (Server)

Basic
- Service_init :- Register app with runtime

Core Functionality
- Run_server :- Start the stub-generated event loop

Other Modifications
- Create RPC stubs required by server
  - The RPCs specified in the vivendi syntax file

Adjusting to Environment

- Availability of compute servers varies wildly
  - Mobility ⇒ Cannot expect just one situation

  Resource Poor
  Resource Rich
  (Smart Rooms etc.)

- Tactics allow us to automatically use extra resources in environment
  - Without modifying the application
Why is this Useful?
- Shields application from uncertainty in environment
  - Load on servers, wireless bandwidth
  - Execute same tactic on multiple servers
  - Take fastest result
- Opportunistic execution
  - To meet latency constraints
  - Execute multiple tactics on multiple servers
  - Return highest fidelity result that satisfies latency constraints

Additional Benefits
- Code is easier to test and debug
  - Very few developer-added lines
  - Most of the new code is stub generated
- Experts can now do the retargeting
  - Should require less time than novices
  - Resulting applications should be excellent

Tactics: Using Extra Servers

Meeting Latency Constraints

<table>
<thead>
<tr>
<th>Metric</th>
<th>Fidelity</th>
<th>Latency</th>
<th>Average (s)</th>
<th>Standard Deviation (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Running to Completion</td>
<td>1.0</td>
<td>1.96</td>
<td>0.15</td>
<td>0.51</td>
</tr>
<tr>
<td>Taking Best Result after 1s</td>
<td>0.75</td>
<td>1.00</td>
<td>0.01</td>
<td>0.75</td>
</tr>
</tbody>
</table>

Application = Pangloss-Lite
Thinkpad 560X (200 Mhz Pentium)

Metric = \( \frac{fidelity}{latency} \)

Two–Phase Evaluation
- Chroma is good
  - Achieves near-optimal performance in mobile environments
  - Overhead is low
- RapidRe is good
  - Allows legacy applications to be easily, quickly, and effectively retargeted to use Chroma

Applications
Application Modification Times

<table>
<thead>
<tr>
<th>App</th>
<th>Stage A (s)</th>
<th>Stage B (s)</th>
<th>Stage C (s)</th>
<th>Total (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Face</td>
<td>10.3 (1.7)</td>
<td>36.6 (4.5)</td>
<td>33.6 (17.8)</td>
<td>80.5 (22.7)</td>
</tr>
<tr>
<td>Flite</td>
<td>12.6 (7.8)</td>
<td>37.7 (6.7)</td>
<td>20.6 (16.4)</td>
<td>70.9 (20.4)</td>
</tr>
<tr>
<td>Javes</td>
<td>29.3 (14.9)</td>
<td>31.0 (6.5)</td>
<td>42.1 (10.2)</td>
<td>102.4 (26.2)</td>
</tr>
<tr>
<td>GLVU</td>
<td>66.3 (20.8)</td>
<td>65.1 (22.5)</td>
<td>40.3 (7.7)</td>
<td>171.7 (33.8)</td>
</tr>
<tr>
<td>Music</td>
<td>49.6 (15.7)</td>
<td>68.2 (17.1)</td>
<td>83.0 (23.0)</td>
<td>200.8 (45.4)</td>
</tr>
<tr>
<td>Paulite</td>
<td>36.2 (7.7)</td>
<td>48.7 (20.2)</td>
<td>32.8 (14.7)</td>
<td>117.8 (36.6)</td>
</tr>
<tr>
<td>Radiator</td>
<td>17.3 (6.0)</td>
<td>45.3 (8.7)</td>
<td>39.4 (7.0)</td>
<td>101.9 (11.7)</td>
</tr>
</tbody>
</table>

Solution: Remote Execution

- Augment capabilities of handhelds by using nearby servers

- But how can good performance be achieved in mobile environments?
- And easily allow legacy applications to use remote execution?

Performance of Modified Apps

- Tactics files had few errors (not shown)
- Client App performance shown

<table>
<thead>
<tr>
<th>Partitions</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>Face</td>
<td>100%</td>
<td>100%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GLVU</td>
<td>86%</td>
<td>90%</td>
<td>93%</td>
<td>93%</td>
<td>87%</td>
<td>89%</td>
<td>88%</td>
<td>91%</td>
<td>91%</td>
<td>89%</td>
<td>89%</td>
<td>90%</td>
<td>89%</td>
</tr>
<tr>
<td>Music</td>
<td>80%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>Paulite</td>
<td>80%</td>
<td>80%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>Radiator</td>
<td>80%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
</tbody>
</table>

- All server apps had equivalent performance

Key Insight

For a large number of applications

- Number of useful remote partitions is small
  - Largest so far is 7 partitions
    - Modular level coarse-grained partitions
- Application developer specifies these partitions (static partitioning)
  - At runtime, pick the optimal partition and locations (dynamic partitioning)

Solution: Tactics

- Concise description of application’s remote execution capabilities
  - Only the useful remote partitions are described
  - Can be captured in a compact declarative form
  - Each tactic performs the required operation
    - Operation ➔ application specific notion of work
- Tradeoff between dynamic and static partitioning
  - RPC model
    - Assume servers have been discovered and are able to handle any RPC call (no code migration)
  - Coarse-grained remote execution

- All server apps had equivalent performance
Too good to be true?
• Results are incredibly good
  – < 3 ½ hours for novices to retarget complex applications with good performance
• What’s the Catch?

Validation of Ease of Use
• Conduct Software Usability Study
  – Using senior undergrads
• Modify a large number of real applications
  – 7 applications + 1 training application
  – Created by other groups
  – Written in many languages
  • Java, C, C++, Ada, TCL/TK

Background
• Proliferation of Mobile Devices
  – Cellphones, PDAs, watches
  – Laptops are not really mobile devices
• However, smaller size  compromise CPU and battery
  – Unable to run large applications
• But, these applications are highly useful
  – Language translation
  – Speech recognition
  – Pattern recognition (signs, text etc.)
Moore's Law

Usefulness of Training

Usefulness of Documentation

Effect of Experience

Using Loaded Servers
Stages for User Study

<table>
<thead>
<tr>
<th>Stage A</th>
<th>Stage B</th>
<th>Stage C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tactics File</td>
<td>Client Component</td>
<td>Server Component</td>
</tr>
<tr>
<td>Read Docs</td>
<td>Read Docs</td>
<td>Read Docs</td>
</tr>
<tr>
<td>Application</td>
<td>Include file</td>
<td>Include file</td>
</tr>
<tr>
<td>In</td>
<td>Register</td>
<td>service_init API call</td>
</tr>
<tr>
<td>Out</td>
<td>Cleanup</td>
<td>Create RPCs</td>
</tr>
<tr>
<td>RPC</td>
<td>Find Fidelities</td>
<td>run_server API call</td>
</tr>
<tr>
<td>Tactic</td>
<td>Do Tactics</td>
<td>Compile and fix</td>
</tr>
</tbody>
</table>

Errors in Tactics File

<table>
<thead>
<tr>
<th>Apps</th>
<th>Params</th>
<th>RPCs</th>
<th>Tactics</th>
<th>Harmsless</th>
<th># Apps</th>
<th>Okay</th>
</tr>
</thead>
<tbody>
<tr>
<td>Face</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>File</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>GLVU</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>Janus</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Music</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Panlite</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>Radiator</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Total</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>25</td>
<td>18</td>
</tr>
</tbody>
</table>

Errors in Stages B and C

<table>
<thead>
<tr>
<th>Apps</th>
<th>Trial Errors</th>
<th>Non-Trivial Errors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Apps</td>
<td>Reg. Late</td>
</tr>
<tr>
<td>Face</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>File</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>GLVU</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>Janus</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Music</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Panlite</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>Radiator</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Total</td>
<td>11</td>
<td>9</td>
</tr>
</tbody>
</table>

Experimental Scenarios

<table>
<thead>
<tr>
<th>ID</th>
<th>Load</th>
<th>BW</th>
<th>User Prefs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td>Low</td>
<td>High</td>
<td>Highest quality result</td>
</tr>
<tr>
<td>T</td>
<td>Low</td>
<td>High</td>
<td>Lowest latency result</td>
</tr>
<tr>
<td>LH</td>
<td>Low</td>
<td>High</td>
<td>Highest quality result within X seconds</td>
</tr>
<tr>
<td>HH</td>
<td>High</td>
<td>High</td>
<td>Highest quality result within X seconds</td>
</tr>
<tr>
<td>LL</td>
<td>Low</td>
<td>Low</td>
<td>Highest quality result within X seconds</td>
</tr>
<tr>
<td>HL</td>
<td>High</td>
<td>Low</td>
<td>Highest quality result within X seconds</td>
</tr>
</tbody>
</table>