The HSF Conditions Database: Intelligent Caching

Candidate: Ernest Sorokun
Mentor: Lino Gerlach
HSF Conditions DB – Overview

Conditions Data handling is reoccurring problem w/ unique challenges

- Heterogenous data structure (a priori unknown)
- High access rates (from distributed computing)

HSF gathered experience from various experiments

- Published set of recommendations *
- A reference implementation was developed **

- Separate meta-data & payloads (file catalogue)
- Already in use: sPHENIX @ BNL (~25k jobs)
- No server-side caching yet

* HSF Conditions Databases activity: https://hepsoftwarefoundation.org/activities/conditionsdb.html
** HSF Conditions Database Reference Implementation: link

```bash
curl http://<host>/api/payloadiovss/?gName=test_gt&iovNum=42
-> {type_1: url_1, type_2: url_2, ...}
```
Cache workflow

Cache cell

- request time
- frequency
- Add. info
- response
- url

Eviction policy

- evict
- new

Cache max size

New request

- No
  - In Cache
    - Get DB response
- Yes
  - Get Cache response

Update cache
- Request time
- Frequency
- Etc.

Cache full?

- No
  - End
- Yes
  - Evict one cell
### Classical cache strategies

<table>
<thead>
<tr>
<th>FIFO: First In First Out</th>
<th>LRU: Least Recently Used</th>
<th>LFU: Least Frequently Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Order for eviction is the same as an entry order</td>
<td>• Evict a cell with the oldest request timestamp</td>
<td>• Evict a cell with the lowest request frequency</td>
</tr>
<tr>
<td>• Insertion ordered dictionary</td>
<td>• Request ordered dictionary with timestamps</td>
<td>• Dictionary with request counters</td>
</tr>
</tbody>
</table>

**FIFO**
- Order for eviction is the same as an entry order.
- Insertion ordered dictionary.

**LRU**
- Evict a cell with the oldest request timestamp.
- Request ordered dictionary with timestamps.

**LFU**
- Evict a cell with the lowest request frequency.
- Dictionary with request counters.
Data example

- Web server access logs from sPHENIX
  - First iteration of calibrations
  - Time independent
  - No need of Intelligent caching
  - Classic caching strategies gives 99.6% serves from cache

- Cryogenic failure prevented next data taking period

- Retrieve log files from other experiments
  - ALICE
  - CMS
  - ATLAS

- We investigate them for our research but none of them are 1:1 comparable to HSF Conditions DB

Special thanks to:
Costin Grigoras (ALICE)
Andrea Formica (ATLAS)
Dave Dykstra (CMS)
Chris Pinkenburg (sPHENIX)
Simulated Data

- We study only the change in the minor IOV sub-parameter

- Individual ‘requesters’ run in parallel (at least have overlap)
  - A requester never repeats a request (local caching)
  - A requester takes a fixed period (plus random fluct.) before moving to next IOV

- Three ‘campaigns’ running in parallel:
  - Data-like, 100 requesters, 1s and 2s period
  - Simulation-like, 100 requesters, 2s period

- Each requester makes 100 calls
  - The requesters start with random delay
    - Simulate non-instant batch submission
Use of classical strategies

- Cheat cache - optimal cache strategy (theory)
  - Knows all requests (even in future)
- LRU – best strategy, but still room for improvement
- Goal: get closer to Cheat Cache's performance using ML
**Deep Learn approach**

**Deep Learn model**
- Input: last 100 requested minor IOV
- Hidden layers:
  - Dense 16
  - Dense 16
  - Dense 16
  - LSTM 16
- Output: probability for each of 100 minor IOVs to be in the next request
- Loss_func – categorical crossentropy

**Evict policy**
- Removes cell in the cache with the lowest probability to be requested on next step

**Problems**
- Processing time - slow
- Needs additional memory for processing
Reinforcement Learning | Q-Learning

Concepts
- State Space
- Action Space
- Reward
- Q-Table

New Q value calculation:

\[ \text{New } Q(s, a) = Q(s, a) + \alpha [R(s, a) + \gamma \max Q'(s', a') - Q(s, a)] \]
Q-Learning for caching

**Time Ordered Dict Cache**

\[ \{cell_0, cell_1, cell_2, \ldots, cell_{n-2}, cell_{n-1}, cell_n\} \]

**Actions:**
Eviction one of the first and last two values of the time-ordered cache

**States:**
Parametrized states by the combination of features
- IOV, counter, timestamp -> Boolean / discrete
  - Difference of timestamps and counters shows actuality of the value
  - Difference of IOV shows backward values

**DB calls vs Cache size**

- **Cache types**
  - CHEAT
  - LRU
  - LFU
  - RL
Conclusion & Outlook

• Investigated request logs of condition DB of sPHENIX
  • Cryogenic failure before time-dependent calibrations
  • Classic caching strategies would reduce DB-calls by 99.6%

• Resorted to more complex simulated access logs

• Developed two intelligent caching methods:
  • Supervised Deep Learning
  • Reinforcement Learning

• Both result in fewer db-calls than classic strategies
  • still have to optimize run-time and robustness
Thank you for attention
# RL Features

<table>
<thead>
<tr>
<th>Feature</th>
<th>Condition</th>
<th>True</th>
<th>False</th>
<th>Else</th>
</tr>
</thead>
<tbody>
<tr>
<td>Feature 1</td>
<td>$IOV_0 - IOV_1 &lt; IOV_{n-1} - IOV_n$</td>
<td>1</td>
<td>-1</td>
<td>0</td>
</tr>
<tr>
<td>Feature 2</td>
<td>$IOV_0 - IOV_{n-1} &lt; IOV_1 - IOV_n$</td>
<td>1</td>
<td>-1</td>
<td>0</td>
</tr>
<tr>
<td>Feature 3</td>
<td>$tm_{LRU} - tm_{avg} \leq tm_{MRU} - tm_{avg}$</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Feature 4</td>
<td>$use_{LRU} - use_{avg} \leq use_{MRU} - use_{avg}$</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Feature 5</td>
<td>$IOV_{LRU} - IOV_{avg} \leq IOV_{MRU} - IOV_{avg}$</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
All strategies

DB calls vs Cache size

Cache types
- CHEAT
- LRU
- LFU
- RL
- Deep Learn

Percentage of DB calls vs Cache size
DB calls vs requests

Cache size: 20

Cache types
- CHEAT
- FIFO
- LRU
- LFU
- Deep L

0 5000 10000 15000 20000 25000 30000

0 5000 10000 15000 20000