Predicting Whole-Program Locality through Reuse Distance Analysis

Chen Ding  Yutao Zhong
Computer Science Department
University of Rochester
Rochester, New York
{cding,ytzhong}@cs.rochester.edu
outline

• Reuse pattern analysis
  • Approximate reuse distance analysis
  • Pattern recognition
  • Distance base sampling
• Evaluation
  • Reuse distance measurement
  • Pattern prediction
• Use of pattern information
• Related work
• Conclusions
Approximate reuse distance analysis

Measuring reuse distance between two data accesses means counting the number of distinct data between them.

Naïve algorithm would need $O(N^2)$ time and $O(N)$ space for a trace of length $N$.

we can organize the last-access time of all data in a search tree.
Take $O(\log M)$ time and $O(M)$ space.

(a) An example access sequence.
The reuse distance between two b’s is 5.

(b) Store and count only the last access of each data.

(c) Organize last access times as a tree. Each node represents a distinct element. Attribute time is its last access time, weight is the number of nodes in the subtree. The tree search for the first b finds the reuse distance, which is the number of nodes whose last access time is greater than 4.
Approximate reuse distance analysis

trade accuracy for efficiency especially space efficiency

O(\log \log M) time and O(\log M) space

The tree data small enough to fit in physical memory and processor cache

Two approximation algorithms
dm = the measured distance
da = the actual distance
1. bounded relative error e 1 \geq e \geq 0 \ da - dm / da \leq e
2. bounded absolute error B B \geq 0 \ da - dm \leq B
We guarantee dm \leq da
Analysis with a bounded relative error
Code for Approximate analysis with a bounded relative error

**data declarations**

```
TreeNode = structure (time, weight, capacity, size, left, right, prev)
```

- **root**: the root of the tree representing the trace
- **epsilon**: the upper bound to the error rate

**algorithm** `ReuseDistance(last, current)`

// inputs are the last and current access time
1. `TreeSearchDelete(last, distance)`
2. `new = TreeNode(current, 1, 1, 1, 1, 1, 1)`
3. if (tree size ≥ 4 * log₁⁺ε root.weight + 4) TreeCompression(new) Assert(compression more than halves the tree)
4. return distance
end algorithm

**subroutine** `TraceSearchDelete(time, distance)`

// time is last access time of the current data.
// distance will be returned.
```
node = root; distance = 0
while true
    node.weight = node.weight - 1
    if (time < node.time and
        node.prev.exists and time ≤ node.prev.time)
        if (node.right.exists)
            distance = distance + node.right.weight
        if (node.left not exists) break
        distance = distance + node.size
        node = node.left
    else if (time > node.time)
        if (node.right not exists) break
        node = node.right
end if
```
```
break
end if
end while
node.size = node.size - 1
return distance
```
end subroutine `TraceSearchDelete`

**subroutine** `TreeCompression(n)`

// n is the last node in the trace
distance = 0
n.capacity = 1
while (n.prev exists)
    if (n.prev.size + n.size ≤ n.capacity)
        // merge n.prev into n
        n.size = n.size + n.prev.size
        n.prev = n.prev.prev
        deallocate n.prev
    else
        distance = distance + n.size
        n = n.prev
        n.capacity = max(distance * ε, 1)
    end if
end while
Build a balanced tree from the list and return the root
end subroutine `TreeCompression`
Analysis with a bounded absolute error
Comparison

• categorize previous methods by their organization of the data access trace.

<table>
<thead>
<tr>
<th>Analysis methods</th>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>trace as a stack (or list) [30]</td>
<td>$O(NM)$</td>
<td>$O(M)$</td>
</tr>
<tr>
<td>trace as a vector [6, 2]</td>
<td>$O(N \log N)$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>trace as a tree [34, 41, 2]</td>
<td>$O(N \log M)$</td>
<td>$O(M)$</td>
</tr>
<tr>
<td>list-based aggregation [25]</td>
<td>$O(NS)$</td>
<td>$O(M)$</td>
</tr>
<tr>
<td>block tree [44]</td>
<td>$O(N \log \frac{M}{N})$</td>
<td>$O(\frac{M}{N})$</td>
</tr>
<tr>
<td>dynamic tree compression</td>
<td>$O(N \log \log M)$</td>
<td>$O(\log M)$</td>
</tr>
</tbody>
</table>

$N$ is the length of execution, $M$ is the size of program data

Table 1: Asymptotic complexity of measuring full distance

• the uses of reuse distance analysis in cache optimization.
Pattern recognition

• Pattern recognition detects whether the recurrence pattern is predictable across different data inputs
• Three step of pattern recognition
  • 1. collecting reference histograms
  • 2. Recognizing patterns
  • 3. limitation
1. collecting reference histograms

• A reference histogram is a transpose of the reuse distance histogram. It sorts all memory accesses based on their reuse distance and shows the average distance of each k percent of memory references.

• Two purpose for using the reference histogram:
  • 1. isolate the effect of non-recurrent parts of the program
  • 2. the size of bin controls the granularity of prediction.
Recognizing patterns

• Given two reference histograms from two different data inputs (training input) we construct a formula for each bin

\[
\begin{align*}
  d_{1i} &= c_i + e_i \cdot f_i(s_1) \\
  d_{2i} &= c_i + e_i \cdot f_i(s_2)
\end{align*}
\]

• \(d_i\) be the distance of the \(i\)th bin
• \(s_1\) be the data size of the first training input
• \(C_i\) and \(e_i\) is coefficients

Fi can be constant pattern, linear pattern or sub-linear pattern

\[
\begin{align*}
p_{\text{const}}(x) &= 0 \\
p_{\text{linear}}(x) &= x \\
p_{1/2}(x) &= x^{1/2} \text{ or } p_{1/3}(x) = x^{1/3} \text{ or } p_{2/3}(x) = x^{2/3}
\end{align*}
\]
Limitations

• The profiling inputs should be large enough to factor out the effect of non-recurring accesses.

• The smallest input used in the experiment has four million memory accesses.

• For linear and sub-linear patterns, analysis needs inputs of different data sizes.

• The difference should be large enough to separate pattern functions from each other.
Distance-base sampling

• The purpose of data sampling is to estimate data size in a program execution.

• When the program starts to execute, the sampling version starts to run in parallel until it finds an estimate of data size.

• The sampling is distance-base means that it uses the reuse distance analyzer and monitors each measured distance.

• Peaks which are time samples whose height (reuse distance) is greater than that of its preceding and succeeding time samples
Evaluation – reuse distance measurement

ZDK-2k and Sampling are approximate analysis with the error bound $B = 2048$, but use different type of tree

99% and 99.9% are the analysis with the bounded relative error $e = 1\%$ or 0.1%

The most scalable performance is obtained by the analyzer with 99% accuracy ($e = 1\%$)

The lower graph of Figure 3 compares the accuracy of approximation on a partial histogram of FFT.

99.9% and 99% approximation ($e = 0.1\%$ and $e = 1\%$ respectively), closely match the accurate distance
Evaluative – pattern prediction

The two histograms overlap by 95%

Figure 4: Pattern prediction for Spec2K/Lucas
Uses of pattern information

• Complier design
  • Reuse distance provides much richer information about a program than a cache miss rate does.

• Reconfigurable memory system
  • A recent trend in memory system design is adaptive caching based on the usage pattern of a running program.
  • Since the pattern analysis directly determines the best cache size for capacity misses, it should reduce the search space (and overhead) of run-time adaptation

• File caching
  • faster analysis, which reduces management cost for large buffers (such as server cache), handles larger traces, and provides faster run-time feedbacks.
  • predication
Related work

• Compiler analysis
• Data profiling
• Correlation among data inputs
• Run-time data analysis
Conclusions

• The paper has presented a general method for predicting program locality.

• By using approximate analysis, it is the first time reduces the space cost from linear to logarithmic.

• It extend profiling to provide predication for data inputs other than profiling one.

• It enables correlation among different executions with distance-based histogram and sampling, which overcomes the limitation of traditional code or data based techniques.