Skadu: Efficient Vector Shadow Memories for Poly-Scopic Program Analysis

Donghwan Jeon*, Saturnino Garcia+, and Michael Bedford Taylor
UC San Diego

* Now at Google, Inc.
+ Now at University of San Diego

Skadu means ‘Shadow’ in Afrikaans.
Dynamic Program Analysis

- Runtime analysis of a program’s behavior.
- Powerful: gives *perfect* knowledge of a program’s behavior with a *specific* input.
  - Resolves all the memory addresses.
  - Resolves all the control paths.

Challenges (*Offline DPA*)
- Memory overhead (> 5X is a serious problem)
  - 1GB -> 5GB
- Runtime overhead (> 250X is a serious problem)
  - 5 minutes -> 1 day
Motifs for Dynamic Program Analysis

- **Shadow Memory**: associates analysis metadata with program’s *dynamic memory addresses*.

- **Poly-Scopic Analysis**: associates analysis metadata with program’s *dynamic scopes*. 
Motifs for Dynamic Program Analysis

- **Shadow Memory**: associates analysis metadata with program’s *dynamic memory addresses*.

- **Poly-Scopic Analysis**: associates analysis metadata with program’s *dynamic scopes*.

- **Vector Shadow Memory (this paper)**: associates analysis metadata with BOTH *dynamic memory addresses AND dynamic scopes*. 
Motif #1 Shadow Memory: An Enabler for Dynamic Memory Analysis

- Data structure that associates metadata (or tag) with each memory address.

- Typically implemented with a multi-level table.

Example: Counting Memory Accesses of Each Address

```c
int *addr = 0x4000;
int value = *addr;
*(addr+1) = value;
```

Sample C Code

![Two-level Shadow Memory Diagram]
Dynamic Program Analyses Employing Shadow Memory

- **Critical Path Analysis [1988]**
  - Finds the longest dynamic dependence chain in a program.
  - Employs shadow memory to propagate earliest completion time of memory operations.
  - Useful for approximating the quantity of parallelism available in a program.

- TaintCheck [2005]
- Valgrind [2007]
- Dr. Memory [2011]
- …
Motif #2 Poly-Scopic Analysis

- Analyzes multiple dynamic scopes (e.g. loops and functions on callstack) by recursively running a dynamic analysis.

- Main benefit: provides scope-localized information.

- Commonly found in performance optimization tools that focus programmer attention on specific, localized program regions.
Poly-Scopic Analysis Example: Time Profiling (e.g. Intel VTune)

- Recursively measures each scope’s \textit{total-time}.
  \vspace{1em}
  \begin{itemize}
  \item total-time (scope) = time_{end}(scope) – time_{begin}(scope)
  \end{itemize}

- Pinpoints important scopes to optimize by using \textit{self-time}.
  \vspace{1em}
  \begin{itemize}
  \item self-time (scope) = total-time(scope) - total-time(children)
  \end{itemize}

\begin{tabular}{ |l|c|c| }
  \hline
  \textbf{Scope} & \textbf{total-time} & \textbf{self-time} \\
  \hline
  Loop i & 100\% & 0\% \\
  Loop j & 50\% & 0\% \\
  foo() & 50\% & \textbf{50\%} \\
  Loop k & 50\% & 0\% \\
  bar1() & 25\% & \textbf{25\%} \\
  bar2() & 25\% & \textbf{25\%} \\
  \hline
\end{tabular}
Hierarchical Critical Path Analysis (HCPA): Converting CPA to Poly-Scopic

- Recursively measures \textit{total-parallelism} by running CPA.
  - “How much parallelism is in each scope?”
- Pinpoint important scopes to parallelize with \textit{self-parallelism}.
  - “What is the merit of parallelizing this loop? (e.g. outer, middle, inner)”
- HCPA is useful.
  - Provides a list of scopes that deserve parallelization [PLDI 2011].
  - Estimates the parallel speedup from a serial program [OOPSLA 2011].

![Graph showing speedup compared to 3rd party experts with Wow!! comment]
**Memory Overhead: Key Challenge of Using Shadow Memory in a Poly-Scopic Analysis**

- A conventional shadow memory already incurs high memory overhead (e.g. CPA).
- Poly-scopic analysis requires an independent shadow memory space for each dynamic scope, causing multiplicative memory expansion (e.g. HCPA).

```
for (i=0 to 4)
   for (j=0 to 32)
      foo();
   for (k=0 to 2)
      bar1();
      bar2();
```
## HCPA’s Outrageous Memory Overhead

<table>
<thead>
<tr>
<th>Suite</th>
<th>Benchmark</th>
<th>Native Memory (GB)</th>
<th>w/ HCPA (GB)</th>
<th>Mem. Exp. Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spec</td>
<td>bzip2</td>
<td>0.19</td>
<td>28.2</td>
<td>149X</td>
</tr>
<tr>
<td></td>
<td>mcf</td>
<td>0.15</td>
<td>16.0</td>
<td>105X</td>
</tr>
<tr>
<td></td>
<td>gzip</td>
<td>0.20</td>
<td>21.7</td>
<td>109X</td>
</tr>
<tr>
<td>NPB</td>
<td>mg</td>
<td>0.45</td>
<td>13.0</td>
<td>29X</td>
</tr>
<tr>
<td></td>
<td>cg</td>
<td>0.43</td>
<td>14.4</td>
<td>34X</td>
</tr>
<tr>
<td></td>
<td>is</td>
<td>0.38</td>
<td>13.9</td>
<td>36X</td>
</tr>
<tr>
<td></td>
<td>ft</td>
<td>1.68</td>
<td>66.0</td>
<td>39X</td>
</tr>
<tr>
<td>Geomean</td>
<td></td>
<td>0.36</td>
<td><strong>20.8 GB</strong></td>
<td><strong>59X</strong></td>
</tr>
</tbody>
</table>

Before applying techniques in this paper… [PLDI 2011]
This Paper’s Contribution

Make shadow-memory based poly-scopic analysis practical by reducing memory overhead!

32-core NUMA w/ **512GB** RAM @ supercomputer center

Macbook Air w/ **4GB** RAM @ student’s dorm room

BEFORE  
AFTER
Outline

- Introduction
- **Vector Shadow Memory**
- Lightweight Tag Validation
- Efficient Storage Management
- Experimental Results
- Conclusion
What’s Wrong Using Conventional Shadow Memory Implementations?

- Setup / clean-up overhead at scope boundaries incurs significant memory and runtime overhead.
- For every memory access, all the scopes have to lookup a tag by traversing a multi-level table.
The Scope Model of Poly-Scopnic Analysis

■ What is a scope?
  – Scope is an entity with a single-entry.
  – Two scopes are either nested or do not overlap.

■ Properties
  – Scopes form a hierarchical tree at runtime.
  – Scopes at the same level never overlap.

```
for (i = 0 to 4)
  for (j = 0 to 32)
    foo();
  for (k = 0 to 2)
    bar1();
    bar2();
```
Vector Shadow Memory

- Associates a tag vector to an address.
  - Scopes in the same level share the same tag storage.
  - Scope’s level is the index of a tag vector.

- Benefits
  - No storage setup / clean-up overhead.
  - A single tag vector lookup allows access to all the tags.
**Challenge: Tag Validation**

- A tag is valid only within a scope, but scopes in the same level share the same tag storage.
- Need to check if each tag element is valid.

Counting Memory Accesses in Each Scope

How can we support a lightweight tag validation?
Challenge: Storage Management

- Tag vector size is determined by the level of the scope that accesses the address.
- Need to adjust the storage allocation as the tag vector size changes.

<table>
<thead>
<tr>
<th>Event</th>
<th>Tag Vector of 0x4000</th>
<th>Vector Size</th>
<th>Storage Op</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access from level 2</td>
<td>0 1 5</td>
<td>3</td>
<td>allocate</td>
</tr>
<tr>
<td>Access from level 9</td>
<td>1 2 6 ... 1</td>
<td>10</td>
<td>expand</td>
</tr>
<tr>
<td>Access from level 1</td>
<td>2 3</td>
<td>2</td>
<td>shrink</td>
</tr>
</tbody>
</table>

How can we efficiently manage storage without significant runtime overhead?
Outline

- Introduction
- Vector Shadow Memory
- **Lightweight Tag Validation**
- Efficient Storage Management
- Experimental Results
- Conclusion
Overview of Tag Validation Techniques

For tag validation, Skadu uses version that identifies **active scopes** (scopes on callstack) when a tag vector is written.

- Baseline: store a version for each tag element.
- SlimTV: store a version for each tag vector.
- BulkTV: share a version for a group of tag vectors.
Baseline Tag Validation

- for each level

\[
\text{If (Ver[level] }\neq \text{ Ver\_active[level]) } \{ \\
\quad \text{// invalidate the level} \\
\quad \text{Tag[level] }\leftarrow \text{ Init\_Val} \\
\quad \text{Ver[level] }\leftarrow \text{ Ver\_active[level]} \\
\}
\]

(a) Baseline

<table>
<thead>
<tr>
<th>Tag [N]</th>
<th>Ver [N]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tag [N]</td>
<td>Ver [N]</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Tag [N]</td>
<td>Ver [N]</td>
</tr>
</tbody>
</table>
Slim Tag Validation:
Store Only a Single Version

- Clever trick: create a total ordering between all versions in all dynamic scopes (timestamp).

- Tag Validation:

  // find max valid level
  j = find ( Ver, Ver_active[]);

  // scrub invalid tags from level j+1
  memset ( & Tag[j+1], N-j-1, init_val);

  // update total-ordered version number
  Ver = Ver_active[current_level]
Bulk Tag Validation: Share a Version Across Multiple Tag Vectors

- Clever trick: Exploit memory locality and validate multiple tag vectors together.
- Benefit: Reduced memory footprint and more efficient per-tag vector invalidation.
- Downside: Some tag vectors might be never accessed after tag validation, wasting the validation effort.
Outline

- Introduction
- Vector Shadow Memory
- Lightweight Tag Validation
- Efficient Storage Management
- Experimental Results
- Conclusion
**Baseline:**
**Array-Based VSM Organization**

- A tag vector is stored contiguously in an array.
- Efficient tag vector operations
  - loop through each array element.
- Expensive resizing
  - Resizing would require expensive array reallocation.
  - Unclear when to shrink a tag vector to reclaim memory.

<table>
<thead>
<tr>
<th>Addr / Level</th>
<th>L0</th>
<th>L1</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x4000</td>
<td>(T_{0-0})</td>
<td>(T_{0-1})</td>
<td>(T_{0-...})</td>
</tr>
<tr>
<td>0x4004</td>
<td>(T_{1-0})</td>
<td>(T_{1-1})</td>
<td>(T_{1-...})</td>
</tr>
<tr>
<td>0x4008</td>
<td>(T_{2-0})</td>
<td>(T_{2-1})</td>
<td>(T_{2-...})</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Alternative: Level-Based VSM Organization

- Idea: reorganize tag storage by scope level so that like-invalidated tags are contiguous
- Efficient tag vector resizing
  - Resizing is part of tag validation.
  - Simply update a pointer in level table.
  - Dirty tag tables added to a free list and asynchronously scrubbed.
- Inefficient tag vector operations
  - Tag vectors are no longer stored contiguously.
Use Best of Both: Dual Representation

### Implement Tag Vector Cache Using Arrays

<table>
<thead>
<tr>
<th>Addr / Level</th>
<th>L1</th>
<th>L2</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x4000</td>
<td>(T_{0-0})</td>
<td>(T_{0-1})</td>
<td>(T_{0-})</td>
</tr>
<tr>
<td>0x8004</td>
<td>(T_{1-0})</td>
<td>(T_{1-1})</td>
<td>(T_{1-})</td>
</tr>
<tr>
<td>0x4008</td>
<td>(T_{2-0})</td>
<td>(T_{2-1})</td>
<td>(T_{2-})</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

**Tag Vector**

### Implement Tag Vector Store Using Levels

<table>
<thead>
<tr>
<th>Level Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>L0</td>
</tr>
<tr>
<td>L1</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tag Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>(T_{0-0})</td>
</tr>
<tr>
<td>(T_{1-0})</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

**Evict**

**Fill**

### Fast Execution

**For Recently Accessed Tags**

### Efficient Storage

**For not recently Accessioned Tags**
**Triple Representation: Add Compressed Store for Very Infrequently Used Tag Vectors**

**Tag Vector Store**

- **Level Table**
  - L1
  - L2
  - ...
- **Tag Table**
  - $T_{1-1}$
  - $T_{2-1}$
  - ...
  - $T_{1-2}$
  - $T_{2-2}$
  - ...

**Compressed Tag Vector Store**

- **Evict**
- **Fill**

Compressed Vector Tags
Outline

- Introduction
- Vector Shadow Memory
- Lightweight Tag Validation
- Efficient Storage Management
- **Experimental Results**
- Conclusion
Experimental Setup

- Measure the peak memory usage.
  - Compare to our baseline implementation [PLDI 2011].
  - Target Spec and NAS Parallel Benchmarks.

- HCPA
  - Tag: 64-bit timestamp, Version: 64-bit integer
  - Tag Vector Cache: covers 4MB of address space.
  - Tag Vector Store: 4 KB units.

- Memory Footprint Profiler
  - Please see the paper for details.
HCPA

Memory Expansion Factor Reduction

Final Memory Expansion Factors

<table>
<thead>
<tr>
<th>Native execution’s memory usage.</th>
</tr>
</thead>
<tbody>
<tr>
<td>37.9X 25.5X 8.7X 4.3X 6.6X 3.5X 4.0X 3.9X 3.1X 1.9X 2.3X 2.1X 5.2X 2.8X</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mem. Exp. Factor Reduction (X)</th>
</tr>
</thead>
</table>

- SlimTV
- Dual Representation (includes BulkTV)
- Triple (+ Compression)
Conclusion

- Conventional shadow memory does not work with poly-scopic analysis due to excessive memory overhead.

- Skadu is an efficient vector shadow memory implementation designed for poly-scopic analysis.
  - Shares storage across all the scopes in the same level.
  - SlimTV and BulkTV: reduce overhead tag validation.
  - Novel Triple Representation for performance / storage. (Tag Vector Cache, Tag Vector Store, Compressed Store)

- Impressive Results
  - HCPA: **11.4X memory reduction** from baseline implementation with only 1.2X runtime overhead.
HCPA Speedup

Final Memory Expansion Factors

202X 131X 188X 227X 401X 213X 148X 231X 211X 170X 221X 475X 219X 224X

Speedup (X)

SlimTV  VCache + VStorage  + Dynamic Compression