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*.

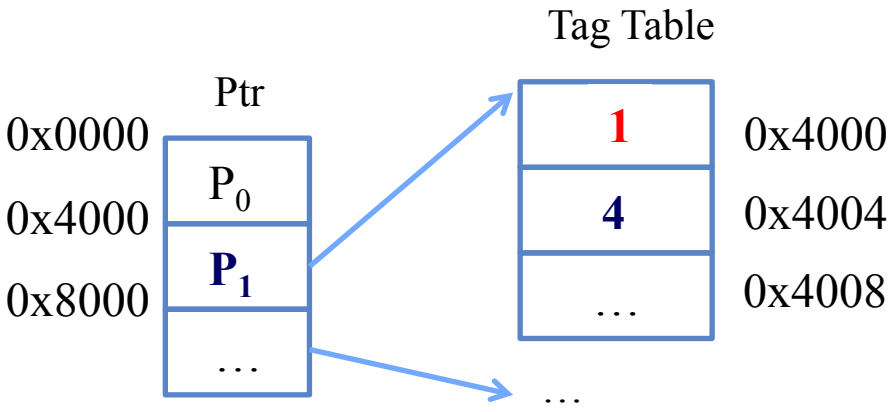
Motif #1 Shadow Memory: An Enabler for Dynamic Memory Analysis

- 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

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

Sample C Code



Two-level Shadow Memory

Dynamic Program Analyses Employing Shadow Memory

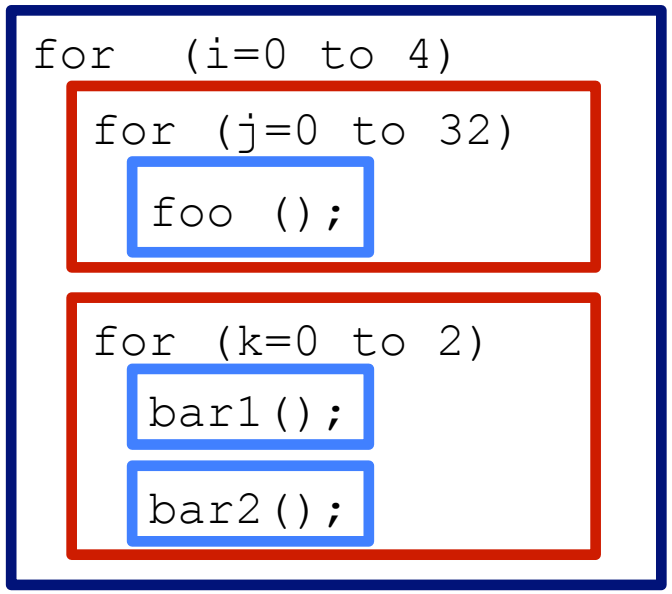
- **Critical Path Analysis [1988]**
  - Finds the longest dynamic dependence chain in a program.
  - Employs shadow memory to propagate earliest completion time of memory operations.
  - Useful for approximating the quantity of parallelism available in a program.
- TaintCheck [2005]
- Valgrind [2007]
- 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 *total-time*.
  - total-time (scope) = $time_{end}$(scope) – $time_{begin}$(scope)
- Pinpoints important scopes to optimize by using *self-time*.
  - self-time (scope) = total-time(scope) - total-time(children)

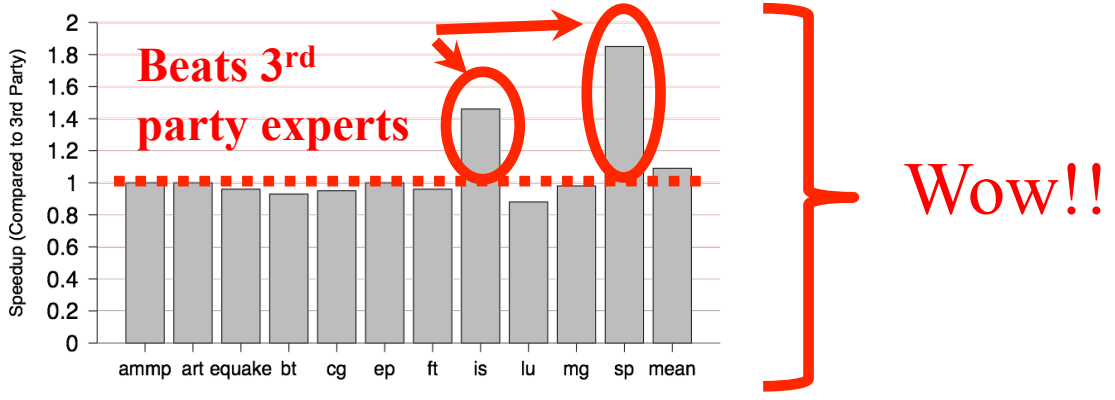


Poly-Scopic Analysis

Scope	total-time	self-time
Loop i	100%	0%
Loop j	50%	0%
foo()	50%	**50%**
Loop k	50%	0%
bar1()	25%	**25%**
bar2()	25%	**25%**

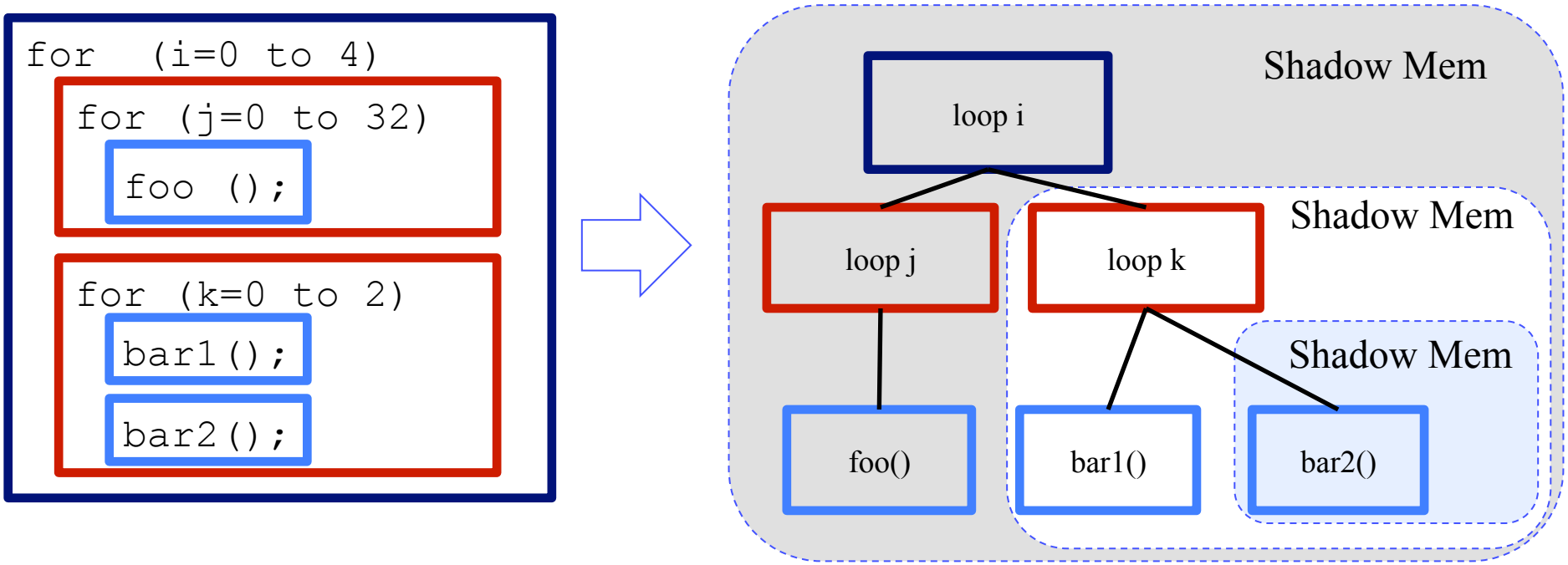
Hierarchical Critical Path Analysis (HCPA): Converting CPA to Poly-Scopic

- Recursively measures *total-parallelism* by running CPA.
  - "How much parallelism is in *each* scope?"
- Pinpoint important scopes to parallelize with *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].



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).



HCPA's Outrageous Memory Overhead

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

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

BEFORE

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

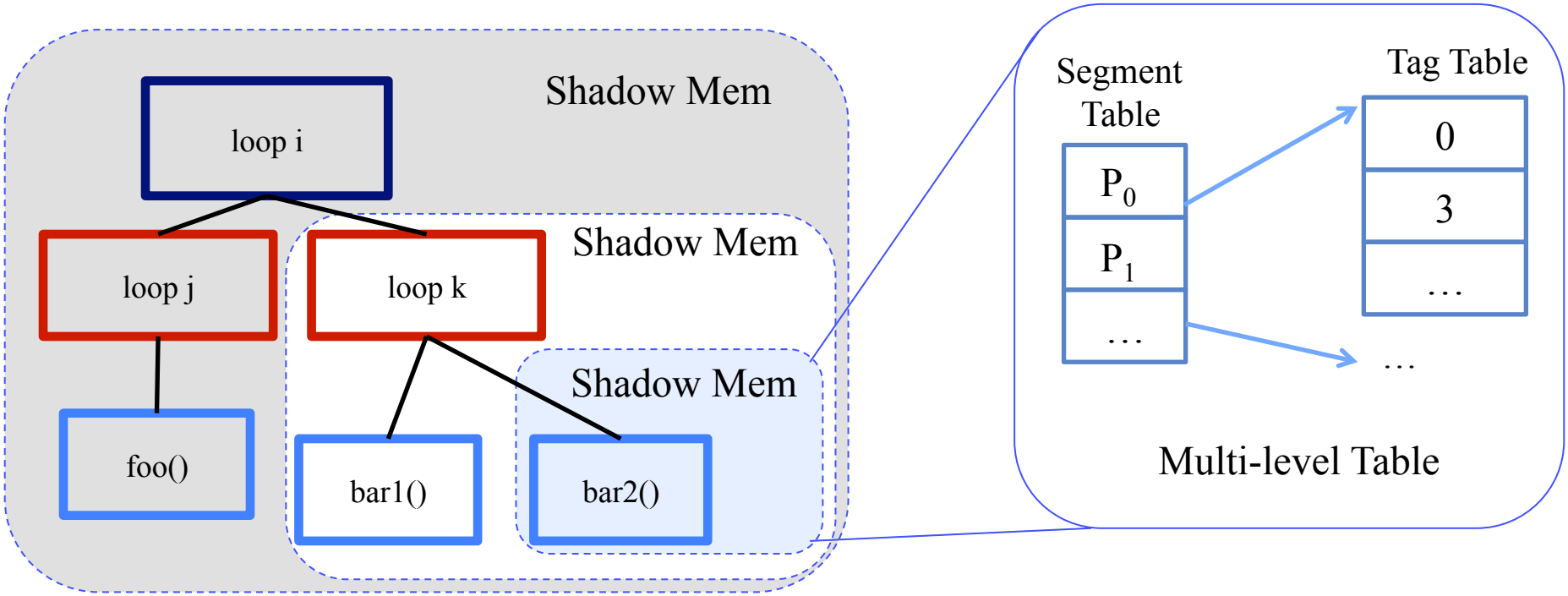
AFTER

Outline

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

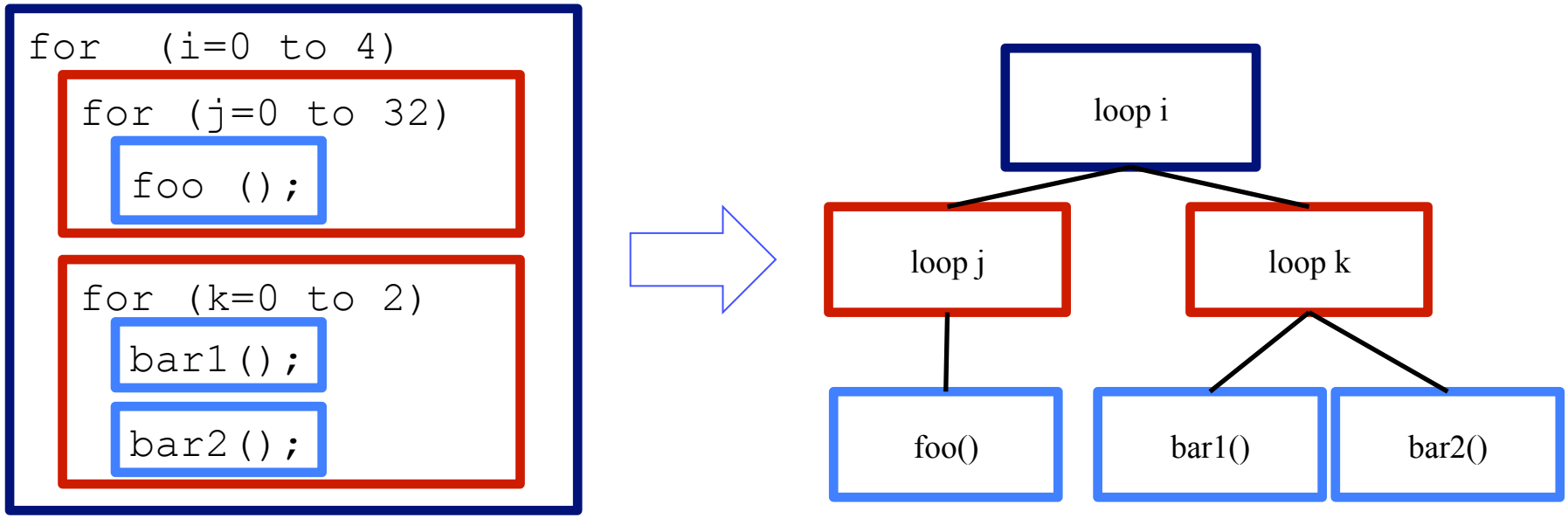
What's Wrong Using Conventional Shadow Memory Implementations?

- 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.



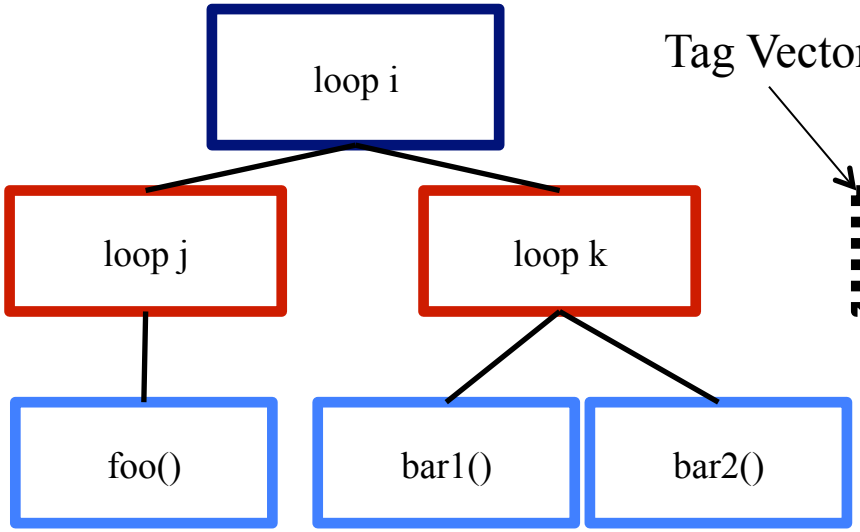
The Scope Model of Poly-Scopic Analysis

- 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.



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.

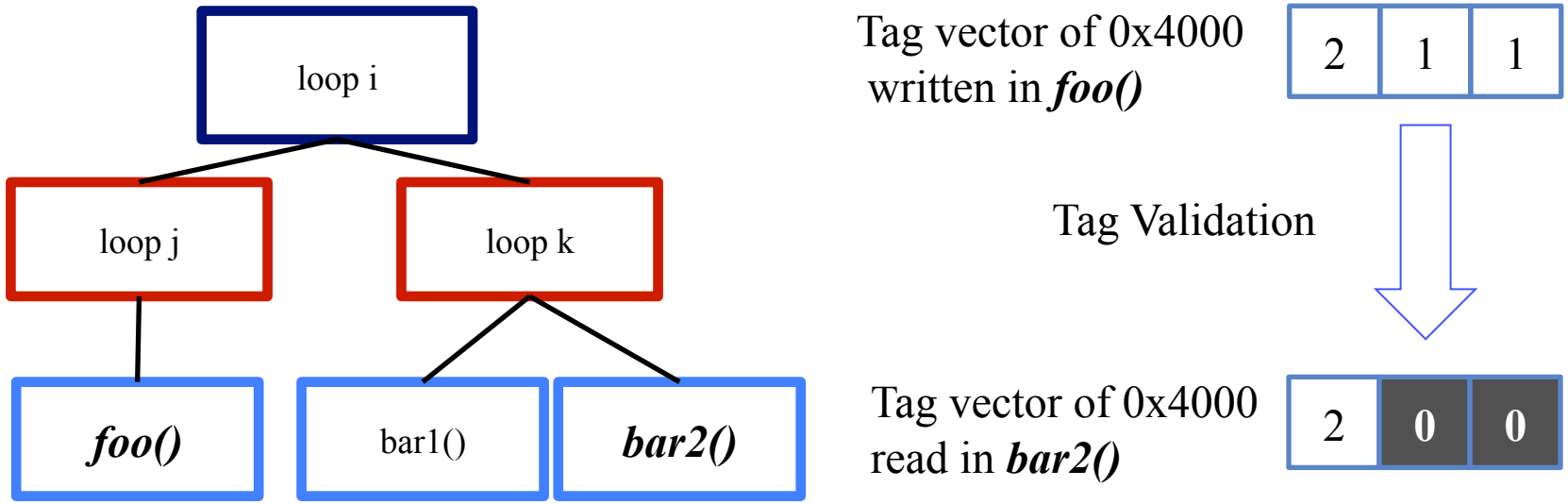


Tag Vector

addr	size	Level 0	Level 1	Level 2
0x4000	3	tag[0]	tag[1]	tag[2]
0x4004	3	tag[0]	tag[1]	tag[2]
0x4008	3	tag[0]	tag[1]	tag[2]

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.

Event	Tag Vector of 0x4000	Vector Size	Storage Op
Access from level 2	0 1 5	3	allocate
Access from level 9	1 2 6 … … 1	10	expand
Access from level 1	2 3	2	shrink

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.

(a) Baseline (b) SlimTV (c) BulkTV

Baseline Tag Validation

- for each level

```
If (Ver [level] != Ver_active[level]) {
        // invalidate the level
        Tag[level] ← Init_Val
        Ver[level] ← Ver_active[level] );
}
```

Tag [N]	Ver [N]
Tag [N]	Ver [N]
…	…
Tag [N]	Ver [N]

(a) Baseline

Slim Tag Validation: Store Only a Single Version

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

(b) SlimTV

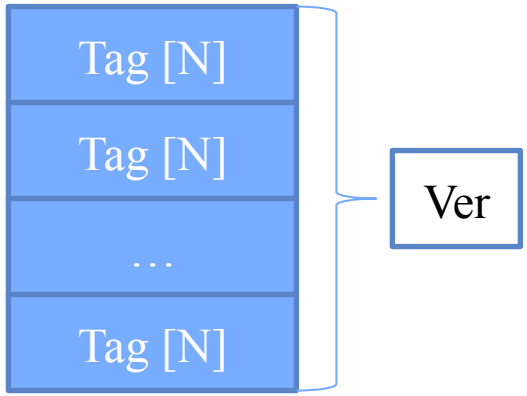
```
// find max valid level
j = find ( Ver, Ver_active[]);

// 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.



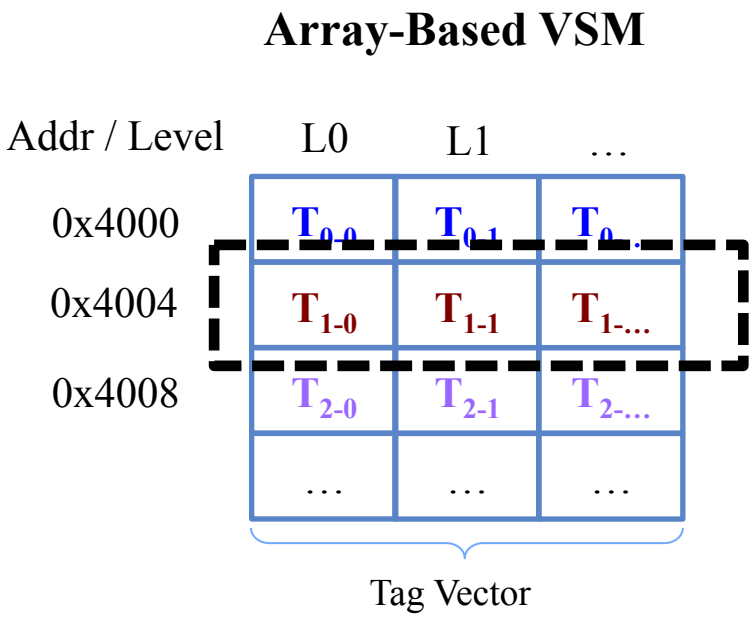
(c) BulkTV

Outline

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

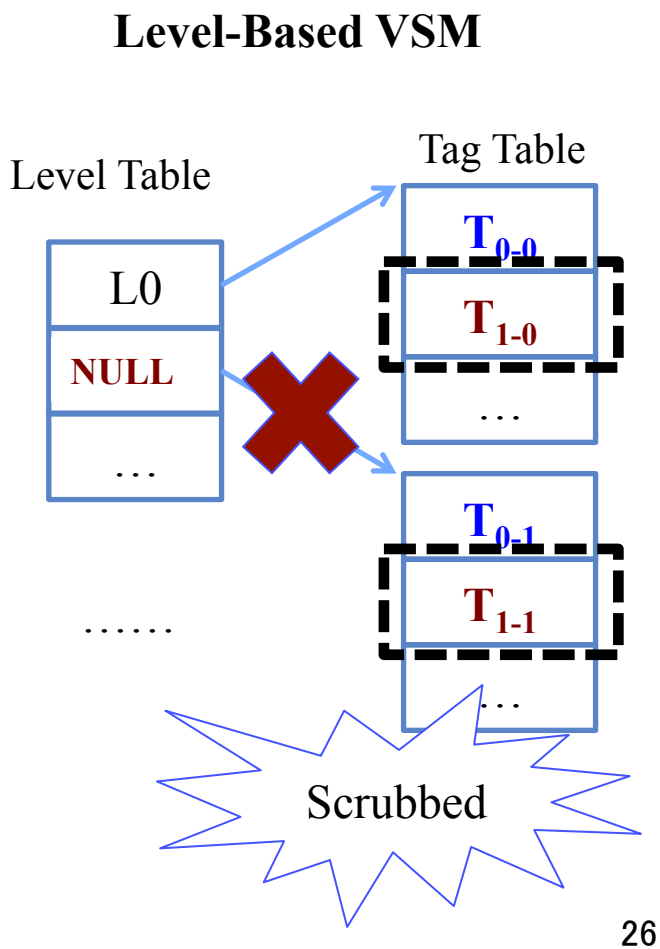
Baseline: Array-Based VSM Organization

- 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.



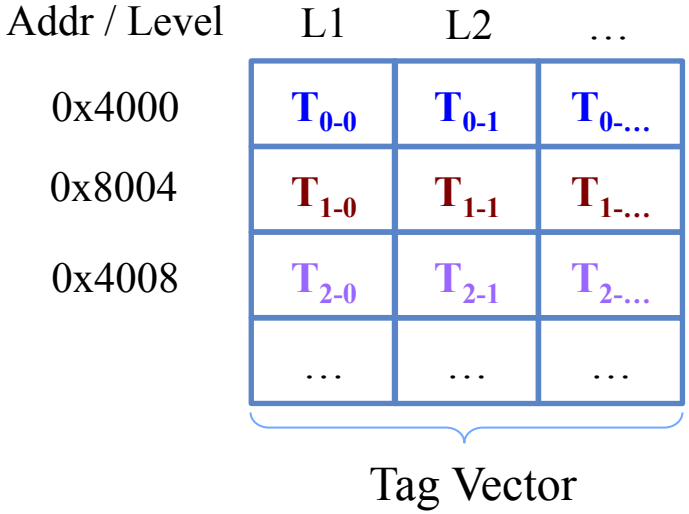
Alternative: Level-Based VSM Organization

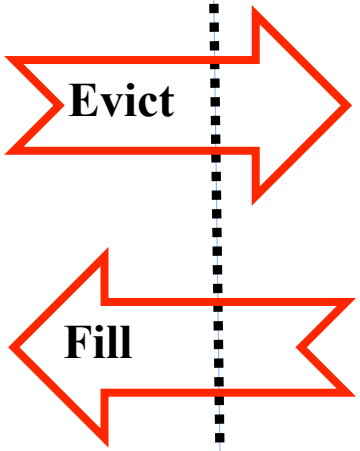
- Idea: reorganize tag storage *by scope level* so that like-invalidated tags are contiguous
- Efficient tag vector resizing
  - Resizing is part of tag validation.
  - Simply update a pointer in level table.
  - Dirty tag tables added to a free list and asynchronously scrubbed.
- Inefficient tag vector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



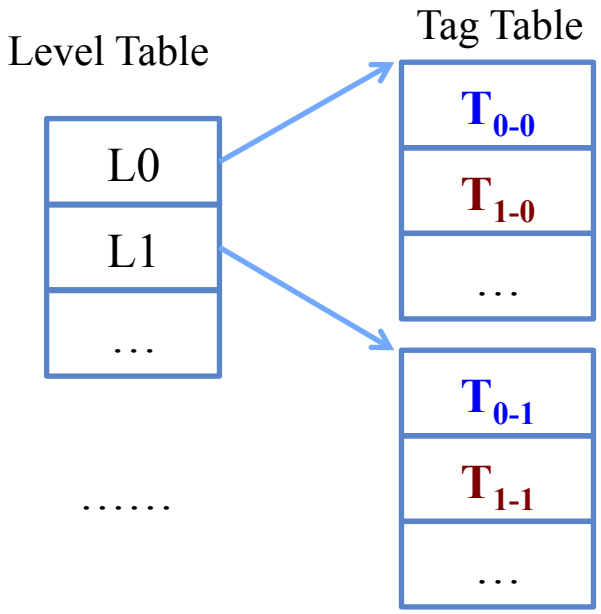
Use Best of Both: Dual Representation

Implement Tag Vector *Cache* Using Arrays





Implement Tag Vector *Store* Using Levels



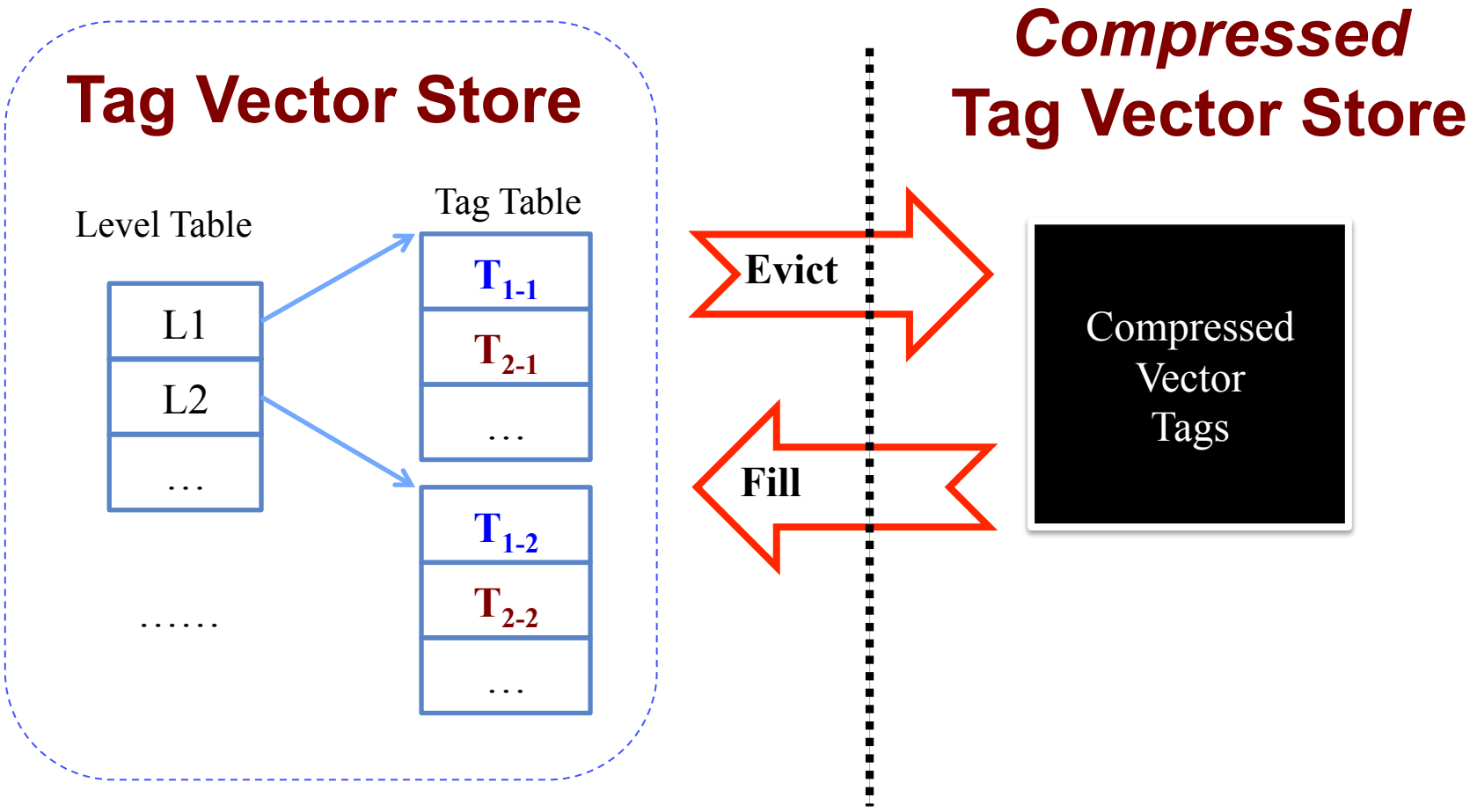
Fast Execution

For *Recently* Accessed Tags

Efficient Storage

For not recently Accessed Tags

Triple Representation: Add Compressed Store for Very Infrequently Used Tag Vectors



Outline

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

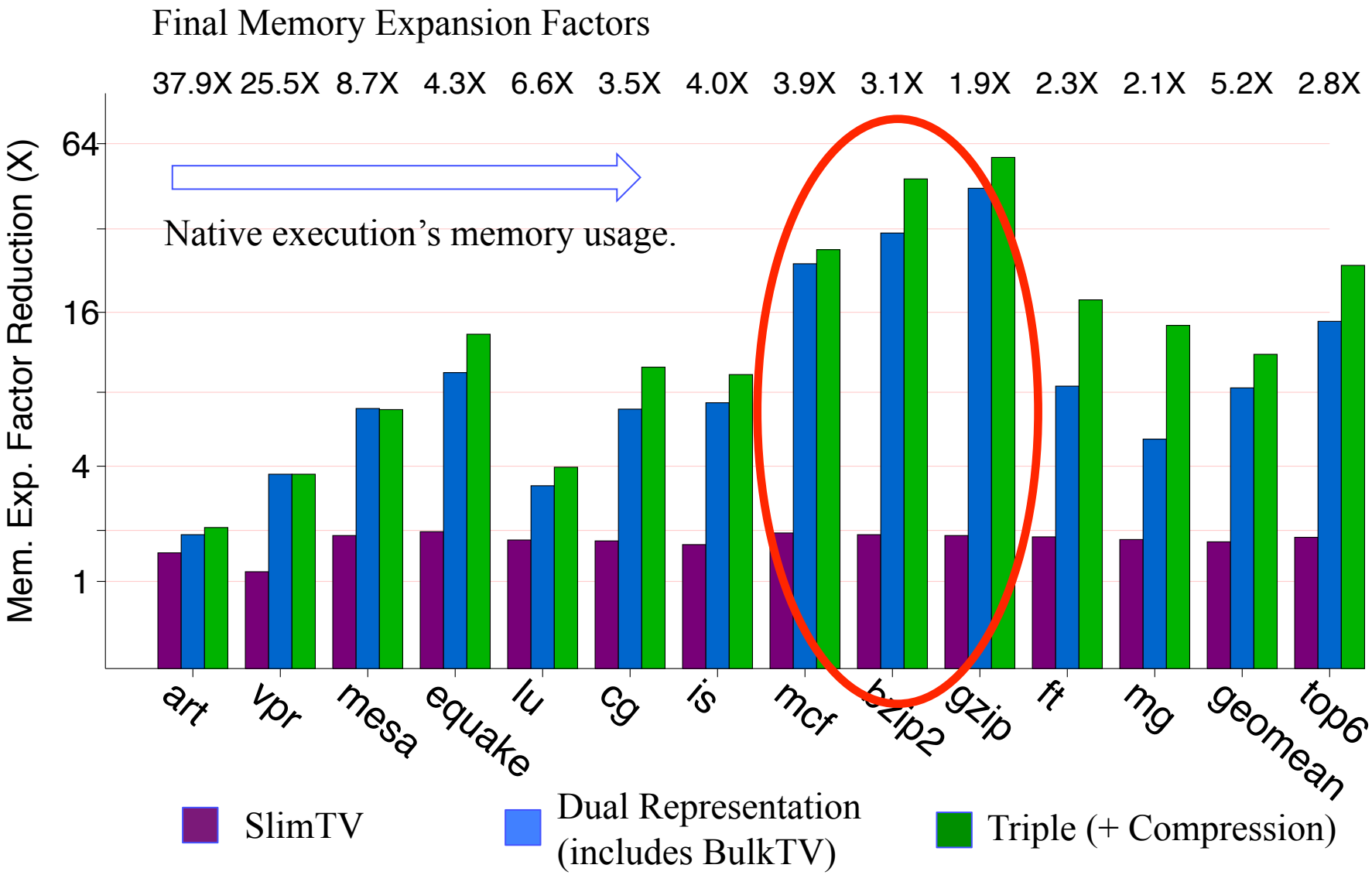
Experimental Setup

- 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.

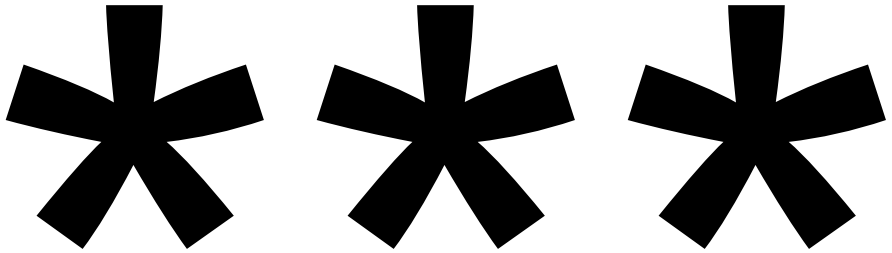
- Memory Footprint Profiler
  - Please see the paper for details.

HCPA
Memory Expansion Factor Reduction



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