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

Donghwan Jeon [*] Saturnino Garcia [†] Michael Bedford Taylor

Department of Computer Science & Engineering
University of California, San Diego
La Jolla, CA, USA

Abstract

Shadow memory is a critical component of many dynamic program analysis frameworks with applications ranging from memory debugging to computer security. Most recent work has focused on optimizing the execution time of analyses that associate a single tag with each memory address. However, an important new class of dynamic analyses (poly-scopic analyses) requires multiple tags for each memory address. These new analyses place additional burdens on memory shadowing infrastructures, especially with regards to memory overhead. Existing shadow memory infrastructures are either unequipped to handle these additional burdens or result in runtime and memory overheads that make them impractical for all but small inputs.

In this paper we propose vector shadow memories (VSMs) as an infrastructure to support poly-scopic analyses. Furthermore we introduce Skadu, a VSM implementation that employs several novel techniques to greatly reduce the runtime and memory overhead associated with the two major challenges of VSMs: tag validation and garbage collection. Our results show that on two separate poly-scopic analyses, memory footprint profiling and hierarchical critical path analysis, Skadu significantly reduces the associated memory overhead: by 14.2× and 11.4×, respectively. In both cases, Skadu makes poly-scopic analysis practical for ordinary desktop and laptop machines.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging; E.1 [*Data Structures*]

General Terms Measurement, Performance

Keywords Poly-Scopic Analysis, Vector Shadow Memory, Hierarchical Critical Path Analysis, Memory Profiling

1. Introduction

Memory shadowing is a general technique that has been used in a wide range of applications: from memory analysis [4, 19] to computer security [5, 17, 23]. Memory shadowing allows metadata, referred to as a *tag*, to be associated with each memory address in a program. These tags vary in function across different dynamic program analyses. For example, tools such as MemCheck [19] have a 1-bit tag for each memory location to indicate whether that address has been properly initialized. Memory shadowing is also widely used in computer security tools to track untrusted data [5].

Introducing Poly-Scopic Analysis While traditional applications of memory shadowing have proven useful, they are limited in their power to identify properties of individual program regions. Their fundamental limitation is that they are inherently *mono-scopic* in nature: they examine only one region (i.e. scope) of a program at a time, usually the whole program. While it is possible to gather information about multiple subregions through multiple runs of the same program, the process is tedious and inefficient, requiring a controlled environment with reproducible I/O. Furthermore, this multiple-run approach can result in prohibitively large log files, often terabytes in size.

The increasing prevalence of many-core and heterogeneous systems points towards an increasing need to localize information within specific regions of a program. In many cases these regions are nested, and it is necessary to separately analyze a nested region. For example, when parallelizing programs for a multi-core processor, we would like to identify regions that could be efficiently parallelized. Likewise, when scheduling a heterogeneous system containing coprocessors with private memory hierarchies, we would like to identify those regions that can be offloaded to a coprocessor without overflowing memory. In these cases, we need to replicate the analyses across many different scopes

[*] The author is now at Google, Inc.

[†] The author is now at the University of San Diego.

CGO '13 23-27 February 2013, Shenzhen China.
978-1-4673-5525-4/13/$31.00 ©2013 IEEE. . . $15.00

of the same program, e.g. across every function boundary or loop body during the program's dynamic execution. We refer to these type of analyses as *poly-scopic* analyses.

A simple example of poly-scopic analysis is the self-time calculation in serial profilers. Self-time is the time spent in each program region, exclusive of the time spent in nested regions. In an implementation that does not employ sampling, the analysis would record the entry and exit time of each region, and upon exit of a region, the region's *total-time* would be computed by subtracting the two, and the region's *self-time* would be computed by subtracting child region total-times from the region's total-time. This self-time for one dynamic invocation of the region would be accumulated across all invocations of the region.

Vector Shadow Memory Although self-time is poly-scopic because it requires that we differentiate entry and exit times across each scope, it is less complex than other poly-scopic analyses because it does not require the use of shadow memory. Poly-scopic analyses that require metadata to be associated with memory require multiple copies of such metadata to be associated with each memory location, one for each scope. We refer to such a shadow as *vector shadow memory* (VSM) because it associates a vector of metadata with each memory location. One could imagine non-poly-scopic analyses that also require vector shadow memory. Although the focus of this paper is poly-scopic VSM implementations, it's quite likely that the techniques in this paper could be adapted to other VSM use cases. Throughout the rest of this paper, we will focus on poly-scopic analyses that require VSM.

Examples of Poly-scopic Analyses An early example of a poly-scopic analysis that requires VSM is the *hierarchical critical path analysis* (HCPA) introduced by Kremlin [6, 7] and Kismet [8]. HCPA concurrently analyzes multiple nested program regions (e.g. functions and loops) to calculate self-parallelism, the amount of parallelism in a region independent of its subregions. It does this by first computing both the time required to execute the instructions in the region (i.e. work) and the longest path through the dependencies between those instructions (i.e. critical path). HCPA then computes the self-parallelism of each dynamic region by using an equation that incorporates a region's critical path, its work, and the critical path of its children.

Kremlin and Kismet have demonstrated the importance of poly-scopic analysis: HCPA has proven highly effective at reducing the number of regions a programmer needs to parallelize [6] and predicting the parallel performance of sequential code [8].

Memory footprint analysis is another example of poly-scopic analysis that requires VSM. This analysis tracks the memory addresses touched by every region in a program to find the total memory used by each region. The information gained from memory footprint analysis could be used by a heterogeneous system to determine pairings between program regions and processing elements based on their respective memory requirements and capabilities.

Suite	Bench mark	Native Memory (GB)	**w/ Shadow Memory (GB)**	Mem. Exp. Factor
Spec	bzip2	0.189	**28.2**	149×
	mcf	0.152	**16.0**	105×
	gzip	0.200	**21.7**	109×
NPB	mg	0.449	**13.0**	29×
	cg	0.427	**14.4**	34×
	is	0.384	**13.9**	36×
	ft	1.683	**66.0**	39×
Geomean		0.355	**20.8**	59×

Table 1. Memory Overheads of Hierarchical Critical Path Analysis (HCPA) Before Applying the Techniques in this Paper. The memory overhead of shadow memory greatly diminishes the practicality of poly-scopic analyses such as HCPA [6]. For an assortment of memory-intensive Spec 2000 and NAS Parallel Bench (NPB) inputs, the shadow memory analysis required a geometric mean of 20.8 GB of memory (an expansion of 59× over the programs' native memory requirements). This paper introduces Skadu, which reduces this memory expansion to a geometric mean of 5.2×.

Memory Overhead of Poly-scopic Analysis Whereas recent work on traditional memory shadowing techniques has focused on reducing runtime overhead [14], a larger concern with poly-scopic analysis is memory overhead. This increased memory overhead is a consequence of the requirement that each active region have its own set of shadow memory metadata (i.e. tags). This requirement leads to multiple tags for each memory address and a corresponding multiplicative increase in the memory usage over that required natively.

High memory overhead has historically confined the use of poly-scopic analyses to supercomputers or other large-memory systems and has likely hindered the development of new poly-scopic analyses. The scale of this problem is demonstrated in Table 1, which shows the memory overhead of Kremlin and Kismet's original implementation of HCPA: the required memory for poly-scopic analysis was between 29× and 149× the memory required natively. Poly-scopic analysis clearly requires new techniques to reduce these overheads if it is to be used on typical developer machines and see widespread use. In this paper we will demonstrate techniques that reduce the HCPA memory expansion factor's geomean from 59× to 5.2×.

Introducing Skadu In this paper we present Skadu, an efficient vector shadow memory (VSM) implementation. Unlike conventional shadow memory that maps a memory address to a tag, VSM maps a memory address to a tag vector, providing efficient read and write operations for poly-scopic analyses without managing a separate shadow memory space for each dynamic scope of a program. Because multiple tags for a shadow memory address tend to be updated at the same time, storing them contiguously as a vector optimizes for locality, reduces translation overheads, and allows SIMD instructions to be used.

This paper contains the following major contributions.

- We define poly-scopic analysis and introduce the concept of vector shadow memory (VSM) to enable this class of analyses.
- We introduce a novel dual representation that allows for efficient management of VSM storage. The first representation, VCache, seeks to maximize the speed at which VSM updates are performed and serves as a cache for the second representation. The second representation, VStorage, optimizes for storage efficiency and allows space-saving techniques including garbage collection and compression to be used for less frequently accessed data. This dual representation allows us to have the best of both worlds–fast access and efficient storage–and greatly improves memory locality.
- We introduce a set of techniques that optimizes memory management as VSM logically grows and shrinks. For instance, rather than actively allocating and deallocating vectors when entering and leaving scopes, the system leaves the storage in place but relies upon *tag validation* to detect when existing data stored in the vector is no longer valid. We evaluate two techniques to reduce the overhead of VSM tag validation: Slim Tag Validation (SlimTV) and Bulk Tag Validation (BulkTV). SlimTV creates a total ordering of all dynamic regions in the program, and uses this ordering to reduce the memory overhead of tag validation to a single value for each memory address's shadow vector. BulkTV further reduces tag validation memory overhead by simultaneously validating entire pages of addresses. Together, SlimTV and BulkTV make the memory overhead of tag validation negligible.
- Skadu introduces an efficient garbage collection mechanism for poly-scopic analyses with large vectors.
- We demonstrate Skadu's effectiveness on two distinct poly-scopic analyses: memory footprint profiling and HCPA. The result is a reduction in memory overhead by 14.2× for memory footprint profiling and by 11.4× for HCPA when compared to baseline implementations.

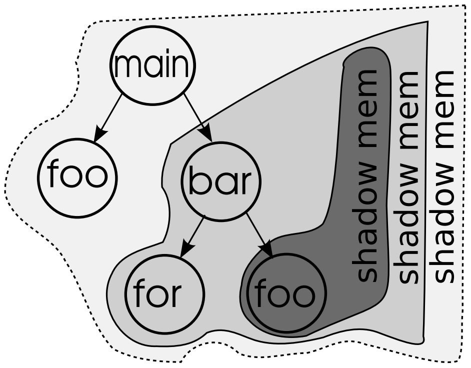
Throughout the rest of this paper, we will expand upon each of these contributions.

2. Vector Shadow Memory (VSM)

In this section we will examine the application of Vector Shadow Memory (VSM) to poly-scopic analyses. We start by introducing traditional shadow memory organization before describing the region-based poly-scopic dynamic analysis employed in the paper. Finally, we overview the baseline vector shadow memory implementation that our techniques improve upon.

```
int main() {        void foo() {
  foo();              // no subregions
  bar();            }
  ...
}                   void bar() {
                      for(i=0..10) { x++; }
                      foo();
                    }
```

(a) Pseudo-code example.



(b) Matching region tree and shadow memories.

Figure 1. Region Hierarchy Overview. The pseudo code in (a) results in the region tree shown in (b). Poly-scopic analyses require separate shadow memories for each scope, as shown by the shaded regions in (b).

Traditional Memory Shadowing Technique In traditional shadow memory infrastructures, each memory address has an associated shadow memory address. Each shadow address may contain some metadata about the associated memory address (a *tag*). Many shadow memory infrastructures use a basic two-level table for accessing tags [12, 15, 19], an organization similar to those used in virtual memory page tables. The size of the tag can range from tiny to large: it is common to see one bit tags in taint tracking infrastructures while applications such as hierarchical critical path analysis [6] require vectors of 64-bit tags.

Poly-Scopic Analysis Traditional memory shadowing requires tracking only a single scope, usually the whole program (i.e. the `main` function). The poly-scopic analyses we consider in this paper require separate dynamic sub-analyses to be simultaneously applied to multiple scopes in the program.

Poly-scopic analyses view an execution of a program as a hierarchical region tree. We define a region to be any single-entry piece of code but we will focus on two particular types of regions: functions and loops. During the execution of a program, the regions of a program form a natural hierarchy. Figure 1 demonstrates this hierarchy (shown in the form of

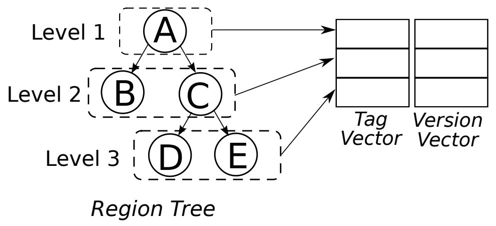


Figure 2. Vector Shadow Memory. VSM eliminates the need for creating a new shadow memory space for every dynamic region by sharing storage across all the dynamic regions in the same level. In addition to tag vectors, version vectors are stored to provide tag validation operation in baseline implementation.

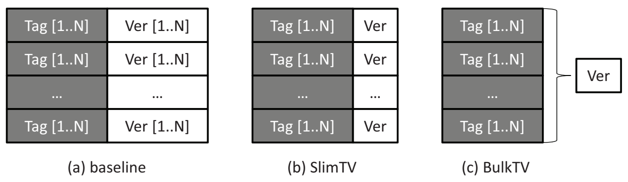


Figure 3. Space Overhead of SlimTV and BulkTV. Compared to the baseline where each tag requires version information, SlimTV uses a version information for a whole tag vector. BulkTV further reduces the space overhead by sharing a single version number across a range of memory addresses. Each row corresponds to the shadow memory information for a single address.

a region tree) for an example piece of code. Each node in the region tree is a dynamic region while an edge from A to B indicates that B is a child of A. When a program execution enters a region but has not exited, we say the region is *active*. Due to the nature of hierarchical region tree, multiple active regions with different scopes can exist at a specific moment in a program's execution.

One way to handle analyzing multiple independent regions could be to include a copy of the shadow memory for each dynamic region, as shown in Figure 1(b). Unfortunately, this naive approach incurs excessive runtime overhead. First, it has to create, maintain, and destroy a large, page table-like data structure for each dynamic region. Second, for every memory operation, it has to traverse multiple shadow memory data structures to retrieve tags for the given address, even though those tags are associated with a single address. These shortcomings led us to idea of vector shadow memory.

Vector Shadow Memory (VSM) Vector shadow memory is a type of shadow memory that has a vector of tags associated with each address. For poly-scopic analyses, a key insight that allows us to create a memory-efficient VSM implementation is that there is at most one active region in any given level in the region tree. Skadu's VSM takes advantage of this hierarchical property to minimize the runtime overhead associated with multiple shadow address spaces. As shown in Figure 2, every region in a given level of the tree is mapped to the same tag storage, forming a tag vector with one entry per level of the tree. Reusing the same storage across regions means that no setup and cleanup overhead is incurred, much like in a callstack.

Sharing storage, however, brings two major challenges. First, a VSM must ensure that stale tags from previous invocations of the static region are ignored using an operation called *tag validation*. This is because a tag is valid only until the program exits the region in which the tag was last updated. Upon a tag vector access, VSM must check if each tag element is still valid and reset to the default value if not. However, naively implemented tag validation requires a significant amount of memory because every tag must also have an associated id that tracks the region that last updated the tag. Section 3 discusses how Skadu achieves tag validation with negligible memory overhead.

Second, the tag vector length for a memory address varies with the program's execution. This variance can lead to a large amount of memory allocated to storing tags for dynamic regions that are no longer alive. A VSM must employ some form of garbage collection to reclaim unneeded storage that results from these effects. Section 4 discusses Skadu's efficient garbage collector for large-tag analyses.

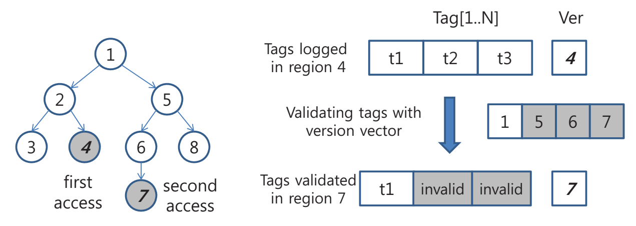
3. Lightweight Tag Validation

Tag validation is an essential operation in a VSM. As described in the previous section, VSM uses version information to determine ownership of tags. Unfortunately, naive tracking of version information is scalable neither in memory overhead nor in performance; it multiplies the native space and time complexity by $\Theta(n)$, where n is the depth of the deepest region that accesses a specific memory address. This section introduces two techniques that enable lightweight tag validation: Slim Tag Validation (SlimTV) and Bulk Tag Validation (BulkTV).

Figure 3 compares the space overhead of these techniques against a baseline implementation. Whereas the baseline stores every active region's version for tag validation, SlimTV stores only a single version. BulkTV further reduces the tag validation overhead by sharing a single version ID across multiple addresses. Together, they make the space requirements of tag validation almost negligible and significantly lower the runtime overhead.

3.1 Baseline Implementation

We begin with a baseline implementation that is based on the work by Garcia et al [6], and is shown in Figure 3a. This baseline implementation features a simple procedure to check tag validity that is based on a design property of the shadow memory: sharing of poly-scopic memory is limited to regions within the same level of the region tree. The



(a) SlimTV's Region Versioning (b) Tag Validation Process

Figure 4. A SlimTV Example. (a) SlimTV exploits the ordering encoded in the version ID of dynamic regions in the program. (b) Illustrates the tag validation process. Suppose a memory location is accessed first in region 4 and later in region 7. After the access in region 4, tag[1:3] will be logged with the region's version number, 4. When the same address is accessed in region 7, the version vector [1, 5, 6, 7] is compared against the stored version number, 4. From the comparison, SlimTV detects that only region level 1 started before the previous tags were written. SlimTV therefore sets tag[2:3] to the value that they would be set to if that memory location had never been accessed (i.e. *scrubs* them) and updates the version field in the shadow memory.

baseline implementation utilizes this sharing property by assigning a unique ID to each region in a level. The unique IDs of all active regions are stored in a version vector associated with a tag vector and updated whenever that memory location is updated. This stored version vector is then used for tag validation on each read: if there is a mismatch between the ID of the current active region in a level and the ID for that level in the version vector, then the tag for that level is invalid.

The baseline implementation suffers from the drawback that it requires storing a version vector for every shadow memory address. This storage requirement leads to a multiplicative space overhead of $\Theta(n)$ just for tag validation, where n is the depth of the region. This approach also incurs a large number of memory loads and stores from reading/writing version vectors, resulting in higher runtime overhead.

3.2 Slim Tag Validation (SlimTV)

Skadu introduces a new tag validation technique known as Slim Tag Validation (SlimTV). SlimTV improves upon the baseline implementation by eliminating the need to store a version vector with each tag vector; only a single value needs to be stored. This technique not only reduces the space overhead from $\Theta(n)$ to $\Theta(1)$ but also eliminates the excessive loads/stores associated with accessing the stored vector, greatly reducing the runtime overhead.

SlimTV relies on the key insight that unique IDs can be used to create a total ordering of all dynamic regions in the dynamic region tree. SlimTV assigns IDs to regions in the order in which they begin. During the access of a tag vector only the ID of the most recently entered, active region is stored, ensuring that is the stored ID is the largest current ID. The stored ID is compared with the vector containing one ID for each currently active region. Active regions with IDs greater than the stored ID started *after* that region and are therefore invalid. SlimTV reduces the problem of tag validation to finding the minimum region level with an invalid tag: active regions at deeper levels must have started later and therefore are also invalid.

Figure 4 provides an example of SlimTV's tag validation. In this example a memory address is written to in the region with version 4. This single version number is then stored along with the tags for each active region. This same address is read later in the region with version 7, at which time the active version vector is $\langle 1,5,6,7 \rangle$. Of the active regions only the first (1) has an ID less than the stored version (4); starting from region 5, all other regions are invalid, and must be cleared – i.e. set to their initial value as if those addresses had never been touched before. We refer to this process of finding and clearing invalid values as *scrubbing*.

Theorem 1. *Suppose V_{curr} is the current version vector while v and T are the version and tag vector stored in shadow memory for an arbitrary memory address. $T[i]$ is valid if and only if $V_{curr}[i] \leq v$.*

Proof. Assume for contradiction that $V_{curr}[i] > v$ but $T[i]$ is valid. Let V_{old} be the version vector when v and T were stored. Because $T[i]$ is valid, $V_{old}[i] == V_{curr}[i]$, and therefore $V_{old}[i] > v$. However, this is a contradiction because v (the stored version ID) is by design the largest ID at the time of access, meaning that it should be the largest value in V_{old}. □

3.3 Bulk Tag Validation

Skadu's SlimTV technique reduces the memory overhead of tag validation to a constant factor but this may still result in significant memory overhead. For example, when shadowing every byte of memory, the overhead incurred from an 8-byte version identifier is 8X the original memory space. Skadu therefore introduces an additional technique, Bulk Tag Validation (BulkTV), shown in Figure 3c, that can reduce the memory overhead to a negligible amount while additionally reducing the runtime overhead.

BulkTV's key idea is to amortize the tag validation's memory overhead across many addresses. BulkTV accomplishes this amortization by using only a single version number for a range of memory addresses (a *page*). Each page of shadow memory data will include a number of tag vectors, one for each address, but only a single version number. When any of the tag vectors of any addresses needs to be updated, the SlimTV scrubbing algorithm will be applied to all of the addresses corresponding to that page, scrubbing stale values, and ensuring that the new version number correctly

```
1  void scrubBulkTV(Addr addr, Vector<Version> verVector) {
2      // do common work for all the tag vectors in the page
3      Page page = getPage(addr);
4      Version version = page.version();
5
6      if (version == verVector.lastElement()) return;
7
8      int scrubStart = 1 + maxValidLevel(version, verVector);
9      page.setVersion(verVector.lastElement());
10
11     // foreach mem addr, scrub invalid tags in tag vector
12     for (Vector<Tag> tagVector : page.tagVectors())
13         for (int level = scrubStart to tagVector.size())
14             tagVector[level].clear();
15 }
```

Figure 5. BulkTV's Pseudocode. BulkTV builds upon SlimTV, significantly reducing tag validation's memory overhead by sharing a single version across all the version vectors in a page. BulkTV finds the maximum valid level of the tag vectors in the page (line 8). Once BulkTV finds the maximum valid level, it scrubs invalid tag elements – for each address, it traverses the tag vector and sets invalid elements to the default value.

characterizes the entire page of tag vectors. Figure 5 shows the pseudocode of BulkTV.

The memory savings of BulkTV is clearly tied to the size of the page: the bigger the page, the bigger the benefit. For example, a modest 4KB page leads to a drastic reduction of validation storage by 4096$\times$ when shadowing every byte.

BulkTV can also have an impact on the runtime overhead of tag validation. BulkTV negatively impacts runtime by adding overhead associated with validating a whole page—we must scrub tag vectors corresponding to all of the addresses, even though those memory addresses may never end up being accessed. This factor ultimately depends on the locality of memory accesses: higher locality will lead to less wasted scrubbing.

Spatial locality often causes consecutive calls to the function `maxValidLevel` to return the same result, resulting in few if any invalidations after the first call. This means that one of the deepest levels is the highest valid level and therefore reverse linear search will quickly find the maximum level. The exact runtime behavior is dependent on the locality exhibited throughout the program but our results in Section 6 show that only a moderate amount of locality is needed to result in a net reduction in runtime overhead.

4. Efficient Storage Management

For runtime efficiency we would like to avoid freeing elements of tag vectors immediately when a region exits. However, leaving these invalid tags in shadow memory increases the memory overhead associated with VSMs. This overhead may not be important for analyses that require a small tag (e.g. a vector of 1-bit tags for each memory location tracked by memory footprint profiling) but some analyses require large tags (e.g. a vector of 64-bit tags for each memory location tracked by HCPA). In these large-tag analyses, simply leaving invalid tags in memory can severely increase memory overhead. Skadu supports garbage collection as a method for balancing runtime and memory overhead for these large-tag applications.

Skadu's garbage collector applies the tag validation to existing vectors, releasing invalid tag elements as they are uncovered. However, efficient garbage collection is challenging for two primary reasons. First, there is a trade-off between ease of garbage collection and the speed of access: garbage collection can reclaim memory most aggressively when data from invalidated levels is grouped together but access speed is optimized when data from the same address is grouped together. Second, garbage collection should avoid addresses that will be frequently used in the near future so as to avoid frequent vector resizing. In this section, we discuss how Skadu overcomes these two challenges.

Dual Representation The key idea behind Skadu's storage management is to separate fast, short-term shadow memory from space-efficient, long-term shadow memory. Figure 6a shows the interaction between these two representations, VCache and VStorage.

Skadu initially places tag vectors in the VCache, evicting them to the VStorage only as needed. The VCache is geared toward fast-access time; sized appropriately, it minimizes the number of accesses to the slower-access VStorage. The VStorage is designed for long-term storage and therefore attempts to minimize memory overhead. It does this through a level-based storage infrastructure that facilitates efficient garbage collection.

Vector Cache (VCache) The VCache stores frequently used tag vectors, making common case access time fast. These tag vectors are stored in an array format to further reduce access time. Unlike conventional hardware caches, VCache does not pull in shadow storage associated with consecutive addresses because it does not improve performance to do so. Because of this, the VCache uses SlimTV for low-overhead tag validation but not BulkTV.

Figure 6b shows the structure of the VCache. Each cache line contains the version and the tag vector associated with a memory address. All cache lines have the same allocated maximum vector size for better performance at the cost of possibly wasted memory but the actual size of the vector is stored with each line. VCache's memory requirement is very small compared to that of VStorage because of the reduced address space it covers. The VCache is direct mapped to reduce access time while still providing good hit ratios. The VCache is automatically resized if the maximum vector size is reached.

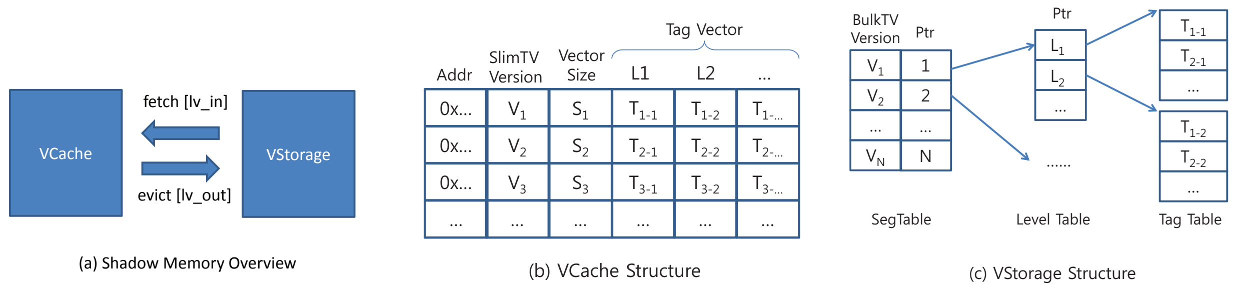


Figure 6. Overview of Skadu Shadow Memory Organization. (a) To exploit the memory footprint and liveness characteristics of hierarchical regions, Skadu uses a VCache, reducing memory requirements and improving performance. (b) The VCache is optimized for the performance, handling most shadow memory requests and allowing a memory-efficient organization of the VStorage. (c) The VStorage is optimized for low memory overhead with the addition of a level table. These level tables group tags by scope (i.e. their location in the dynamic region tree), allowing efficient storage deallocation when a scope is exited. Paired with BulkTV, this three-level organization enables low-overhead storage of infrequently accessed tags. Bits from the memory address are used to hash into both the SegTable and Tag Table.

Vector Storage (VStorage) The VStorage acts as a memory efficient backing store for tag vectors evicted from the VCache. The VCache handles most shadow memory accesses, allowing the VStorage to focus on reducing memory rather than runtime overhead.

Figure 6c shows the structure of the VStorage. The VStorage utilizes a three-level structure that is similar to traditional shadow memory infrastructures [1] but with the novel addition of a level table. The VStorage groups tags by their level (or the index in the vector) rather than the address they shadow. The VStorage also employs BulkTV: all tag tables associated with a segment table entry share a single version ID, which is located in the segment table next to the level table pointer.

Garbage Collection This distribution of tags enables efficient garbage collection, exploiting the fact that all the tags in a tag table become invalidated when regions–and therefore levels–are exited. The garbage collection is performed during BulkTV. When a fetch or eviction occurs at VStorage, BulkTV compares the current version vector against the stored BulkTV version, finding out the deepest level whose tag table is still valid. All the tag tables associated with deeper levels must be invalidated.

The VStorage organization makes invalidation of a tag table simple. Skadu maintains a list of free tag tables: tag invalidation only requires sending off the tag table to be asynchronously scrubbed and returned to the free list. This makes garbage collection extremely lightweight. After a tag table is invalidated, its associated level table entry points to either a clean tag table or a null table, depending on the current tag vector size.

5. Case Studies

To demonstrate Skadu's effectiveness, we implemented two poly-scopic analyses with Skadu: a memory footprint profiler and hierarchical critical path analysis (HCPA). The first represents a relatively lightweight application of Skadu with vectors of small 1-bit tags whereas the second represents a heavyweight one with vectors of larger 64-bit tags. In addition to the techniques described in previous sections, we additionally applied dynamic compression to further reduce memory overhead.

[1] Although not shown, this structure is easily modified to handle 64-bit addressing via an additional table before the segment table, similar to what was proposed in [25].

5.1 Memory Footprint Profiler

The memory footprint profiler tracks the number of memory locations accessed in each *dynamic* region and reports the average memory footprint for each *static* region. It illuminates a program's region-specific memory usage, guiding memory optimizations.

Tag Format Each tag is a single bit that tracks whether or not the address has been touched by a region. This leads to a tag vector of n bits, where n is the depth of the region accessing the address. The profiler watches for the first touch of an address (i.e. tag changing from 0 to 1), incrementing a counter associated with the region when this event happens. This counter is checked when a dynamic region exited; its value then propagates to the statistics associated with the corresponding static region.

The region hierarchy leads to an inclusion property for memory footprint analysis: if a memory address is touched in a region, it must also have been touched in all its ancestor regions. Thus tag vectors are always a sequence of 1's followed by 0's. The footprint profiler exploits this property by performing a simple run-length encoding based compression on the tag vector.

The footprint profiler is able to directly work with the encoded version of the vector, eliminating the need to constantly encode/decode the vector and therefore avoiding additional runtime overhead. The efficiency of this encoded representation places even more importance on efficient tag validation; the footprint profiler uses both SlimTV and BulkTV to make validation as space and time efficient as possible.

Efficiently Measuring Memory Footprint Each memory access triggers a check to see if the footprint of the active regions needs to be increased. This check involves three steps: tag validation, footprint update, and tag update. The tag validation step reads and updates the stored tag vector with the techniques described in Section 3. The footprint update step finds and updates the range of region levels whose memory footprint should be incremented. The tag update step updates shadow memory with the new tag and version for the given address.

Implementation The memory footprint analyzer uses LLVM 2.8 [9] to insert functions calls into the source code that demarcate region boundaries and trigger events on memory accesses. These functions are implemented in a runtime library that is linked in at compile time. The footprint analyzer uses functions and loops as regions because they are natural, programmer-centric boundaries.

We modified the traditional two-level shadow memory organization described in Section 2 to support tag validation and a 64-bit address space. Each segment table and tag table covers 4GB and 64KB of address space, respectively. Each tag is an 8-bit integer, supporting a region tree of depth 256. This was more than enough for all benchmarks we examined in our results. The footprint analyzer supports the use of baseline tag validation, SlimTV, or BulkTV; this allowed us to examine the overheads associated with each of these techniques. Since the encoded tag size is already only extremely small, we do not use garbage collection.

5.2 Hierarchical Critical Path Analysis

Overview Hierarchical critical path analysis (HCPA) is a dynamic program analysis that computes the self-parallelism of each program region [6, 8]. Self-parallelism is the parallelism of a region exclusive of the parallelism of its child regions. HCPA calculates self-parallelism by performing critical path analysis (CPA) on every region of the program, utilizing the program hierarchy to determine the relationships of regions. CPA incurs a large amount of overhead as it requires every operation to be instrumented; this is required to find the *critical path* of the program, its longest set of dependent instructions.

HCPA concurrently calculates CPA on multiple regions, requiring a tag vector of n 64-bit timestamps for n active regions. The size of each tag makes memory overhead a severe issue in HCPA, much more so than the memory footprint profiler. HCPA further exacerbates the memory overhead problem by treating loop bodies as regions; this is in addition to the function and loop regions seen in the memory footprint profiler. The addition of loop bodies increases the depth of the region tree, increasing tag vector sizes and the memory overhead as a result.

HCPA operates on all instructions not just the loads and stores that were instrumented in the memory footprint profiler. This increased instrumentation increases performance overhead. HCPA does not access shadow memory on all instructions though: all non-memory operations utilize a shadow register file. This shadow register file is much smaller than shadow memory and can therefore be optimized for access time rather than space overhead in much the same way as the VCache.

HCPA follows a three step procedure for handling loads. First, it accesses shadow memory to load in the tag vector (the timestamps) for the specified memory address. Next, it calculates the updated tag vector for the target register based on three factors: the loaded tag vector, the tag vector of control dependences, and the estimated cost of a load. Finally, it updates the shadow register file entry for the target register. The process for a store is similar except that the tag vector is initially loaded from the shadow register file and finally stored in shadow memory.

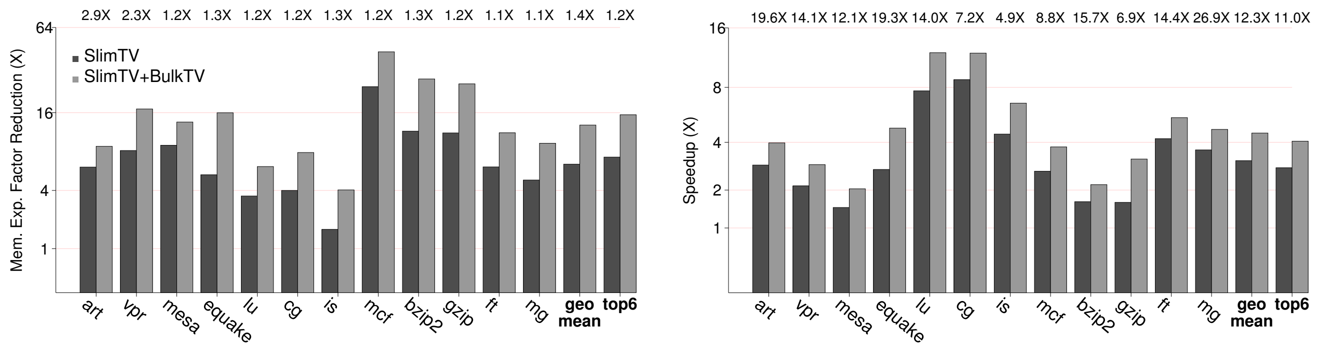
Implementation HCPA uses LLVM to instrument code so that instructions of interest trigger an appropriate handler in the HCPA library. This process is similar to the process for memory footprint analysis, but HCPA instruments nearly all instructions, not just memory loads and stores. The additional instrumentation is required to calculate the critical path through the program.

HCPA utilizes all of Skadu's techniques in order to reduce both the memory and runtime overhead. Shadow memory operations first access the VCache to determine if the target address is available. A VCache miss forces a load from and eviction to the VStorage in the case of a load instruction; a miss on a store instruction simply requires an eviction to the VStorage. HCPA uses a tag table that covers 4KB of address space, which is smaller than the tag tables used by the memory footprint analyzer that cover 64KB. This smaller size reduces the runtime overhead associated with BulkTV, helping offset the increased runtime from having a variable size tag vector in HCPA.

We implemented dynamic compression in HCPA, leveraging Skadu's VCache-VStorage organization. Since the VCache is a smaller memory serving frequently accessed addresses and VStorage is a large memory serving infrequently accessed addresses, we compress most tag tables in VStorage at the granularity of a level table. Only a list of recently referenced level tables are left uncompressed. This list works like a victim cache, protecting against large performance penalties during bursts of high miss rates in the VCache. These bursts would otherwise incur decompression costs on top of the already high cost of accessing the VStorage.

6. Experimental Results

Methodology We examine the effectiveness of Skadu's proposed techniques using the two analyses described in Section 5: a memory footprint profiler and hierarchical critical path analysis (HCPA). Our experiments focus on both the memory and performance overheads associated with vector



(a) Memory Expansion Factor Reduction (b) Speedup

Figure 7. Memory Overhead Reduction and Speedup in Footprint Profiler. Skadu reduces the memory expansion factor from the baseline's 17.8× to 1.4× while maintaining comparable execution time. Numbers on the top represent the (a) memory expansion factor and (b) slowdown versus native execution when both SlimTV and BTV are employed.

Benchmark		Native	Native	Region Depth	
Suite	Name	Mem (MB)	Exec. (Sec)	Mem Profiler	HCPA
SpecInt	bzip2	189	57.1	17	25
	gzip	200	41.0	17	21
	mcf	152	90.5	48	53
	vpr	3	72.5	13	17
SpecFP	art	2	6.5	10	11
	equake	37	114	7	21
	mesa	20	120	20	26
NPB	cg	55	6.4	6	10
	ft	419	11.4	11	18
	is	68	2.0	4	7
	lu	43	82.9	6	12
	mg	434	5.6	8	13

Table 2. Benchmark Characteristics. We examined 12 benchmarks from three benchmark suites. These benchmarks display a wide variety of characteristics including memory usage (2MB to 434MB) and execution time (2 seconds to 2 minutes).

shadow memory (VSM). We tracked the maximum memory overhead because it determines the minimum amount of memory required to successfully run the analysis. All measurements were performed on a 32-core system (8X AMD Opteron 8380 Quad-core processors) with 256 GB of memory running on the Linux 2.6.18 Kernel. For compression, we employed the miniLZO 2.06 library [1].

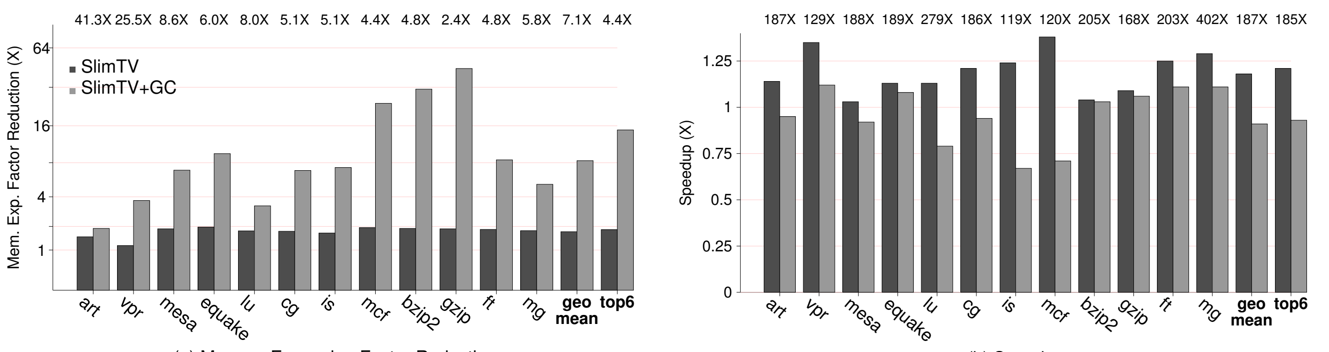
We examined 12 benchmarks across three benchmark suites: SpecInt 2000, SpecFP 2000, and NAS Parallel Bench (NPB) [3]. Due to the limitations in our toolchain, we targeted non-recursive programs written in C programming language. Underscoring the benefits of the techniques in this paper, `bt` and `sp` only run when Skadu is enabled; without Skadu they exit with out-of-memory errors on the 256 GB system. `ep` was excluded since the program has few memory accesses.

Table 2 characterizes each benchmark's native execution, listing runtime, memory footprint, and region depth. SpecFP and NPB benchmarks tend to have regular memory access patterns and contain many dense, array-based operations. Conversely, SpecInt benchmarks have more irregular memory access patterns in addition to deeper region hierarchies. We used SpecInt and SpecFP's 'ref' input set and NPB's 'A' input set for all results.

6.1 Memory Footprint Profiler

As mentioned in Section 5, the memory footprint profiler uses SlimTV and BulkTV for its VSM implementation. The footprint profiler's overheads stem almost solely from tag validation as the run-length encoding of a tag vector produces a representation that requires only 8-bits to store and can be directly manipulated to minimize overhead. The prominence of tag validation in the overheads of this analysis makes it a good target to evaluate the impact of tag validation. The results are compared against those in the baseline implementation. This baseline implementation associates a version with every tag; the vector size in this baseline implementation is fixed to the deepest region level in the program.

Figure 7 shows the memory expansion factors and runtime overheads from the memory footprint profiler. This graph is sorted in order of increasing native memory footprint. The numbers on top of the bars represent the final memory expansion factor and slowdown compared to the native execution. Skadu shows impressive reductions in the memory expansion factor of the memory footprint profiler when combining SlimTV and BulkTV. Skadu reduces the memory expansion factor by a geomean of 13.0×. Benchmarks with larger memory footprints show overall better reductions, 15.5× for top six benchmarks in memory footprint.



(a) Memory Expansion Factor Reduction (b) Speedup

Figure 8. Benefits of SlimTV and BulkTV with Garbage Collection in HCPA. Combining SlimTV, BulkTV and Garbage Collection reduces HCPA's memory expansion factor by 8.3× compared to the baseline implementation at a cost of only 9% in performance overhead. Numbers on the top represent the (a) memory expansion factor and (b) slowdown of Skadu compared to native execution. The techniques are even more effective for benchmarks with large memory footprint (top6), incurring a memory expansion factor of only 4.4×. Even further benefits are attained when compression is enabled.

SlimTV effectively reduces the memory expansion factor and improves performance. SlimTV's main benefits stem from its replacement of the version vector with a scalar version. These benefits will therefore be more pronounced in programs with deep region hierarchies. For example, `mcf` sees the largest reduction in memory expansion because of its region depth (48) that is more than twice the closest benchmark (20). SlimTV also speeds up the analysis by a factor of 3.1× because it eliminates the the large number of loads and stores associated with accessing version vectors.

BulkTV provides additional benefits beyond that of SlimTV. BulkTV reduces the memory overhead of tag validation from 7× (a 56-bit version for every 8-bit tag) to nearly zero (one 64 bit version per 64KB tag table). BulkTV is more effective at reducing memory expansion on programs with large memory footprints: the benefit increases as more tag tables are in use. Figure 7 shows this phenomenon: while the smallest (leftmost) benchmarks see little additionally benefit from BulkTV, the remaining benchmarks see significant improvements in the memory expansion factor. BulkTV also helps improve performance as explained in Section 3. With SlimTV and BulkTV, the geomean memory expansion factor is only 1.4× while slowdown is a manageable 12.3×.

6.2 Hierarchical Critical Path Analysis (HCPA)

Hierarchical critical path analysis is much more costly than the memory footprint profiler in terms of both memory and performance. HCPA's baseline version results in a memory expansion factor of 59.0×, severely limiting its use outside of supercomputers and other high memory environments. HCPA utilizes Skadu's full array of techniques to reign in its overheads. The results are impressive. Skadu's SlimTV, BulkTV and Garbage Collection reduces the memory expansion factor to 7.1×, a reduction of 8.3× compared to the baseline implementation. When Skadu's compression is enabled as described in Section 6.3, memory expansion is further reduced to 5.2×, a reduction of 11.4×.

Figure 8 shows the memory and performance improvements from SlimTV, BulkTV and Garbage Collection. Benchmarks are presented in the same order as they were in Figure 7: in order of increasing memory footprint. The numbers on top of the bars show the memory expansion factor and slowdown vs native when using all of Skadu's techniques except dynamic tag compression. We set the VCache size to cover 1MB of addresses while the list of uncompressed level tables covered 4MB. This represented a decrease in the number of uncompressed level tables compared to the memory footprint profiler. This was a result of a reduced reliance on this list to improve performance: HCPA's use of VCache greatly reduces tag table storage because of nursery effects common in garbage collection infrastructures.

SlimTV not only reduces the memory expansion factor but also improves performance, an outcome similar to what we witnessed in the footprint profiler. SlimTV has a smaller effect on HCPA's memory usage than it did on the footprint profiler's memory usage, with a geomean reduction of 1.7× for HCPA versus 6.6× for the footprint profiler. This difference arises because the baseline HCPA implementation's memory overhead is almost equally split between tags and tag validation; in the memory footprint profiler, almost all the overhead was a result of tag validation. This more equitable split also leads to smaller performance gains for improved tag validation in HCPA.

Skadu's BulkTV and garbage collection with VCache-VStorage architecture (labeled GC in Figure 8) has a significant impact on the memory expansion factor. These benefits ranged from 1.9× (`art`) to 39× (`gzip`) with a geomean of 8.3×. SpecInt benchmarks tend to show greater memory reductions because they move between regions more quickly than the other benchmarks. The runtime overhead is only 15% more than when using only SlimTV.

6.3 Dynamic Tag Compression

In addition to SlimTV and BulkTV, dynamic tag compression can further reduce the memory overhead in VSM. It saves memory by compressing not-recently-used tag tables. However, the saving comes at a cost: increased runtime overhead. This overhead consists of two components: the compression/decompression algorithms and the eviction algorithm used for the list of uncompressed level tables. This list uses a "clock" eviction policy[20] that requires an access bit be updated every time an entry in the list is touched. While a simpler eviction policy may seem desirable (e.g. direct mapped cache), the higher hit ratio of the clock algorithm more than offsets its maintenance costs.

Table 3 shows the impact of dynamic tag compression. It further reduces memory overhead by 25.5% for HCPA on average, achieving a memory expansion factor of 5.2× versus native execution and 11.4× reduction against our baseline. Results are even better for the top six memory intensive applications, where memory overhead is reduced by 37.4%, achieving a final memory expansion factor of 2.8×. For footprint analysis, the memory savings were around 11.9%, increasing Skadu's memory reduction against our baseline from 12.7× to 14.2×. Dynamic tag compression is more effective for reducing HCPA's memory overhead because tag tables in HCPA benefit from our optimization that improves the compression ratio. Rather than compressing raw tag tables, we compressed the differences between two tag tables with adjacent levels. The intuition behind this optimization is that timestamps offsets between two levels are often constants, allowing better compression ratio. The impact on the execution time for HPCA is quite moderate (1.1×).

	Mem. Exp. Factor		Impact	
Bench	No Comp.	With Comp.	Mem. Reduct.	Added Slowdown
mg	5.8X	2.1X	63.7%	1.2X
ft	4.8X	2.3X	52.5%	1.1X
bzip2	4.8X	3.1X	35.7%	1.0X
cg	5.1X	3.5X	31.2%	1.2X
equake	6.0X	4.3X	28.3%	1.2X
is	5.1X	4.0X	22.2%	1.2X
gzip	2.4X	1.9X	22.1%	1.0X
lu	8.0X	6.6X	17.2%	1.4X
mcf	4.4X	3.9X	11.0%	1.9X
art	41.3X	37.9X	8.2%	1.1X
vpr	25.5X	25.5X	0.0%	1.0X
mesa	8.6X	8.7X	0.0%	1.0X
geomean	7.1X	5.2X	25.5%	1.1X
top6	4.4X	2.8X	37.4%	1.2X

Table 3. Performance Impact of Dynamic Tag Compression on HCPA. Dynamic tag compression further reduces the memory overhead at the cost of increased runtime. For example, it reduces the memory expansion factor of HCPA from 7.1× to 5.2×, or by 25.5%. The runtime slowdown is manageable 1.1×. For the top 6 memory intensive programs, the savings average 37.4%.

7. Related Work

Skadu's related work is in four areas: shadow memory applications and design, compression, and garbage collection.

Applications of Shadow Memory Shadow Memory is a generally applicable technique that has been used in many different areas. These areas include security [5, 17, 23], data race detection [18], copy profiling [22], cache contention detection [26], and memory debugging [4, 19] among many others. Skadu's target applications differ from these previous applications in that they are analyses that seek to answer questions about many independent program scopes; previous applications of memory shadowing seek to answer questions about the program as a whole or at least some defined portion of it. Skadu therefore enables a whole new class of scope-based dynamic program analyses, demonstrated in this paper through two case studies with memory footprint profiling and hierarchical critical path analysis [6, 8].

Shadow Memory Design Wide applicability of shadow memory has led to a wide range of shadow memory architectures. Some of these approaches use only a single-level implementation [5, 18], relying on assumptions about the size of address space (e.g. 32-bit addresses) and often allocating half of the address space for shadow memory. This single-level approach is not robust: it often fails in the face of programs that make assumptions about memory placement and often clash with operating systems which have assumptions about object locations. MemCheck [19], pinSEL [12], and an array of tools [10, 13, 16] built using Valgrind [15] use a two-level translation table similar to the one described in Section 2. This approach works well for 32-bit address spaces but does not scale well to 64-bit spaces. Recent work has expanded this basic structure to three-levels to better support 64-bit address spaces [24, 25].

While Skadu's VStorage uses a three-level address translation organization similar to that of Umbra [25], Skadu's overall architecture is optimized to meet the needs of region-based analysis and vector shadow memory operations. Skadu introduces novel shadow memory features such as the VCache, level tables, garbage collection, and tag compression. These additions are unnecessary in traditional memory shadowing applications but are critical in meeting the exacting demands of region-based analysis.

A number of tools propose specialized hardware to reduce the overhead of memory shadowing [11, 21, 27]. Specialized hardware could also potentially be used to accelerate Skadu further.

Our previous work [6] implemented a basic version of vector shadow memory that is comparable to the baseline numbers reported in this paper. The prior version is not practical outside of supercomputer environments because of extreme memory requirements. This paper introduces techniques to reduce memory requirements so as to make poly-scopic analyses available to a broader range of developers. Skadu introduces a number of techniques (e.g. SlimTV, BulkTV, and garbage collection) to reduce the memory overhead of VSMs, formally defines VSM, and defines a class of analyses, poly-scopic analyses, that rely on VSMs. This work additionally introduces memory footprint analysis as an additional example of poly-scopic analysis.

Memory Compression Skadu's basic compression architecture is influenced by compressed memory systems such as IBM's Memory Expansion Technology (MXT) [2] and Connectix RamDoubler. In these systems, caches usually contain uncompressed data while main memory is compressed. Skadu has uncompressed data in its VCache but its VStorage is a mix of compressed and uncompressed data. While a majority of the VStorage is compressed, a frequently used subset is left uncompressed; this "uncompressed buffer" greatly reduces the overhead of compression while incurring only moderate memory overhead. Skadu also increases the effectiveness of compression by exploiting the regularity *across* levels: the differences between tag tables are more regular than the tag tables themselves, leading to an increased compression ratio.

8. Conclusion

This paper has demonstrated a set of techniques for enabling poly-scopic dynamic analyses through a set of space-efficient Vector Shadow Memory optimizations including lightweight tag validation, garbage collection, and dynamic compression. Our results, using two dynamic analyses, are strong. We show that Skadu reduces the footprint analysis' space overhead from 17.8× to 1.3×. Furthermore, it reduces HCPA's space overhead from 59.0× to 5.2×. In both cases, Skadu makes the poly-scopic dynamic analysis practicable where otherwise it would not be.

Acknowledgment

This research was funded by the US National Science Foundation under CAREER Award 0846152, Awards 0725357 and 1018850, and by a gift from Advanced Micro Devices.

References

[1] "LZO data compression library." `www.oberhumer.com/opensource/lzo/`.

[2] Abali, Franke, Shen, Poff, and Smith. "Performance of hardware compressed main memory." *HPCA*, 2001.

[3] Bailey et al. "The NAS parallel benchmarks." In *SC*, 1991.

[4] Bruening, and Zhao. "Practical memory checking with dr. memory." In *CGO*, 2011.

[5] Cheng, Zhao, Yu, and Hiroshige. "Tainttrace: Efficient flow tracing with dynamic binary rewriting." In *Computers and Communications, 2006. ISCC '06.*, june 2006.

[6] Garcia, Jeon, Louie, and Taylor. "Kremlin: Rethinking and rebooting gprof for the multicore age." In *PLDI*, 2011.

[7] Garcia, Jeon, Louie, and Taylor. "The Kremlin Oracle for Sequential Code Parallelization." *Micro, IEEE*, July/Aug 2012.

[8] Jeon, Garcia, Louie, and Taylor. "Kismet: Parallel Speedup Estimates for Serial Programs." In *OOPSLA*, 2011.

[9] Lattner, and Adve. "LLVM: A compilation framework for lifelong program analysis & transformation." In *CGO*, 2004.

[10] Mhlenfeld, and Wotawa. "Fault detection in multi-threaded c++ server applications." *Electronic Notes in Theoretical Comp Sci*, 2007.

[11] Nagarajan, and Gupta. "Architectural support for shadow memory in multiprocessors." In *VEE*, 2009.

[12] Narayanasamy, Pereira, Patil, Cohn, and Calder. "Automatic logging of operating system effects to guide application-level architecture simulation." In *SIGMETRICS*, 2006.

[13] Nethercote, and Mycroft. "Redux: A dynamic dataflow tracer." *Electronic Notes in Theoretical Comp Sci*, 2003.

[14] Nethercote, and Seward. "How to shadow every byte of memory used by a program." In *VEE*, 2007.

[15] Nethercote, and Seward. "Valgrind: A framework for heavyweight dynamic binary instrumentation." In *PLDI*, 2007.

[16] Newsome. "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software." In *NDSS*, 2005.

[17] Qin, Wang, Li, Kim, Zhou, and Wu. "Lift: A low-overhead practical information flow tracking system for detecting security attacks." In *MICRO*, 2006.

[18] Savage, Burrows, Nelson, Sobalvarro, and Anderson. "Eraser: a dynamic data race detector for multithreaded programs." *ACM Trans. Comput. Syst.*, November 1997.

[19] Seward, and Nethercote. "Using valgrind to detect undefined value errors with bit-precision." In *USENIX ATC*, 2005.

[20] Tanenbaum. *Modern Operating Systems*. Prentice Hall Press, 3rd ed., 2007.

[21] Venkataramani, Roemer, Solihin, and Prvulovic. "Memtracker: Efficient and programmable support for memory access monitoring and debugging." *HPCA*, 2007.

[22] Xu, Arnold, Mitchell, Rountev, and Sevitsky. "Go with the flow: profiling copies to find runtime bloat." In *PLDI*, 2009.

[23] Xu, Bhatkar, and Sekar. "Taint-enhanced policy enforcement: a practical approach to defeat a wide range of attacks." In *USENIX Security*, 2006.

[24] Zhao, Bruening, and Amarasinghe. "Efficient memory shadowing for 64-bit architectures." In *ISMM '10*, Jun 2010.

[25] Zhao, Bruening, and Amarasinghe. "Umbra: Efficient and scalable memory shadowing." In *CGO*, 2010.

[26] Zhao, Koh, Raza, Bruening, Wong, and Amarasinghe. "Dynamic cache contention detection in multi-threaded applications." In *Proceedings of the International Conference on Virtual Execution Environments*, VEE '11, 2011.

[27] Zhou, Teodorescu, and Zhou. "Hard: Hardware-assisted lockset-based race detection." In *HPCA*, 2007.