Address Scaling: Architectural Support for Fine-Grained Thread-Safe Metadata Management

Deepanjali Mishra, Konstantinos Kanellopoulos, Ashish Panwar, Akshitha Sriraman, Vivek Seshadri, Onur Mutlu, and Todd C. Mowry

***Abstract*—In recent decades, software systems have grown significantly in size and complexity. As a result, such systems are more prone to bugs which can cause performance and correctness challenges. Using run-time monitoring tools is one approach to mitigate these challenges. However, these tools maintain metadata for every byte of application data they monitor, which precipitates performance overheads from additional metadata accesses. We propose *Address Scaling*, a new hardware framework that performs fine-grained metadata management to reduce metadata access overheads in run-time monitoring tools. Our mechanism is based on the observation that different run-time monitoring tools maintain metadata at varied granularities. Our key insight is to maintain the data and its corresponding metadata within the same cache line, to preserve locality. *Address Scaling* improves the performance of `Memcheck`, a dynamic monitoring tool that detects memory-related errors, by 3.55× and 6.58× for sequential and random memory access patterns respectively, compared to the state-of-the-art systems that store the metadata in a memory region that is separate from the data.**

***Index Terms*—Virtual memory, intermediate address space, metadata management, dynamic program monitoring tools.**

I. Introduction

MODERN software systems are increasingly growing in size and complexity, making them more prone to software bugs that cause performance and correctness challenges [1]. To detect bugs and improve programmer productivity, one approach is to use run-time monitoring tools to automatically analyze and monitor software [2], [3], [4], [5]. These tools monitor the code dynamically during execution and flag unsafe scenarios.

Some popular run-time monitoring tools include Memcheck [2] (built using Valgrind [5] framework), AddressSanitizer [6], and Transactional Memory [3]. Memcheck [2] and AddressSanitizer [6] detect memory-related bugs, and Transactional Memory [3] enables effectively managing locks to protect critical sections.

Run-time monitoring tools typically track some information about application data in a fine-grained manner, i.e., they maintain a few bits/bytes of information about every byte of data used by the application. We call these additional information bits *metadata*. The metadata gets 1) initialized when its corresponding data is allocated and accessed, and 2) updated on every memory operation to the corresponding data byte. For example, Memcheck [7] maintains two bits of metadata for each application data byte. These bits indicate whether the corresponding application data byte is addressable and contains a valid value, i.e., is initialized.

Despite their effectiveness, the performance overhead of run-time monitoring tools often limits their use in production systems. We find that these tools introduce slowdowns ranging from approximately 2× to 20× for applications. The main reasons for these overheads are due to: 1) the overhead from executing additional instructions to monitor and update the metadata, and 2) the increased pressure on the memory subsystem due to additional metadata accesses [8].

Prior works [1], [8], [9], [10] on enhancing dynamic monitoring tools' performance have three limitations. First, despite having hardware support, these approaches have a high overhead for locating and accessing metadata, since it is maintained in a separate region from the data [8]. Second, these mechanisms exhibit high complexity to maintain consistency between data and metadata in parallel applications. Third, some of these mechanisms are designed for a specific monitoring application and cannot be generalized.

Our goal is to design a hardware framework that efficiently implements diverse run-time monitoring tools with low hardware complexity. To achieve this, we propose a new hardware framework for fine-grained metadata management, which we call *Address Scaling*. The key insight behind our framework is to store the metadata corresponding to a data element in the same cache line that holds the data. To achieve this locality, *Address Scaling* introduces an intermediate address space between the virtual and physical address spaces named *Scaled Address Space*, where the data and its corresponding metadata are co-located in the same cache line. Therefore, further translation ensures that they are mapped to the same physical cache line.

We evaluate *Address Scaling* by comparing it against an approach where the data and its metadata are not co-located in the same cache line [5]. For our evaluations, we run microbenchmarks that perform sequential and random memory accesses with `Memcheck`. In both of these cases, *Address Scaling* outperforms the existing approach by 3.55× and 6.58×, respectively.

Manuscript received 15 January 2024; revised 20 February 2024; accepted 3 March 2024. Date of publication 6 March 2024; date of current version 29 March 2024. *(Corresponding author: Deepanjali Mishra.)*

Deepanjali Mishra was with the Microsoft Research India, Bengaluru, Karnataka 560001, India. She is now with Carnegie Mellon University, Pittsburgh, PA 15213 USA (e-mail: mishradeepanjali25@gmail.com).

Konstantinos Kanellopoulos and Onur Mutlu are with ETH Zurich, 8092 Zürich, Switzerland.

Ashish Panwar and Vivek Seshadri are with the Microsoft Research India, Bengaluru, Karnataka 560001, India.

Akshitha Sriraman and Todd C. Mowry are with Carnegie Mellon University, Pittsburgh, PA 15213 USA.

Digital Object Identifier 10.1109/LCA.2024.3373760

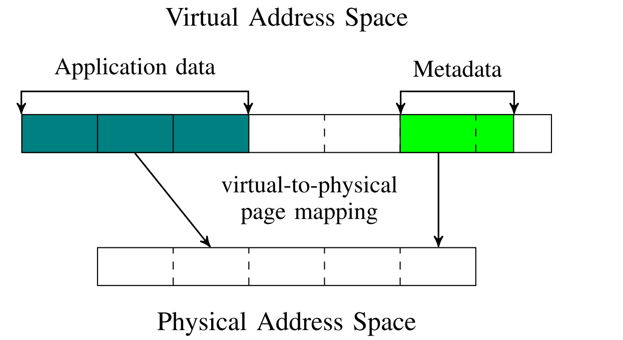

Fig. 1. Metadata management in existing approaches [5], [6]. The dotted lines indicates page boundaries. The white space in the virtual address space is unused by the application. The virtual-to-physical page mapping is shown only for one data page and one metadata page.

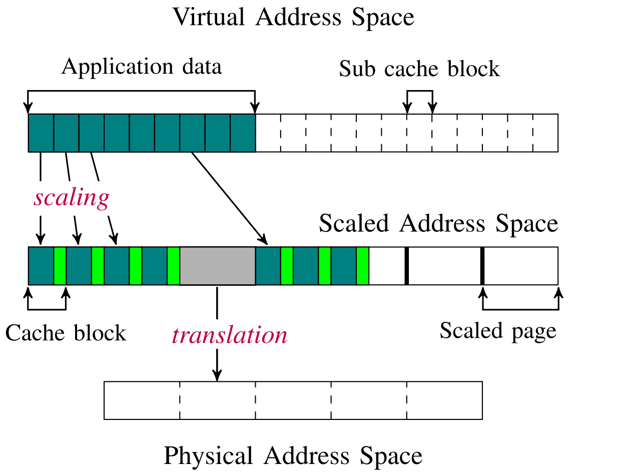

Fig. 2. Conceptual design of *Address Scaling*. The dotted lines in the virtual address space indicate sub-cache-line boundaries. Each cache line in the scaled address space contains a sub-cache-line and metadata associated with it.

II. Address Scaling

We design *Address Scaling*, a hardware framework that provides efficient fine-grained metadata management. Our framework leverages the fact that modern processors access main memory at cache line granularity. The *key idea* behind *Address Scaling* is to store the metadata corresponding to a data element in the same cache line that holds the data.

Overview: As shown in Fig. 1, existing systems typically store metadata in a separate region in the virtual address space which maps to different page frames in physical memory. Consequently, a piece of data and its metadata reside, by design, in two different cache lines.

To bring data and its corresponding metadata into the same cache line, *Address Scaling* introduces an intermediate address space between the virtual address space (VAS) and the physical address space (PAS), called the *Scaled Address Space* (SAS). Unlike existing systems, where a virtual address is directly translated into a physical address, *Address Scaling* first scales an address in VAS into an address in SAS and then translates that into an address in PAS.

Fig. 2 shows the overview of *Address Scaling*. We divide the VAS into small chunks called sub-cache-lines. The size of a sub-cache-line is smaller than that of a cache line and determines how much metadata is stored for each byte of data. For example, if we want to store 16 bytes of data in each cache line of size 64B, then the size of a sub-cache-line would be 16 bytes. *Address Scaling* maps the *i*th sub-cache-line in the VAS to the *i*th *cache line* in the SAS. Thus, each cache line in SAS contains a sub-cache-line worth of data followed by a *hole*. This hole can be used to store the metadata for the sub-cache-line worth of data in the cache line. Further translations from SAS to PAS ensures that data and metadata are mapped to the same physical cache line.

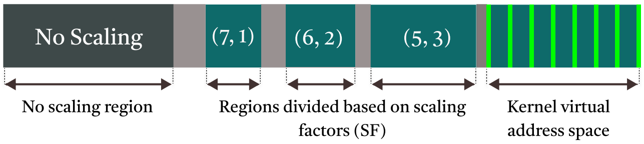
No Scaling
(7, 1)
(6, 2)
(5, 3)
No scaling region
Regions divided based on scaling factors (SF)
Kernel virtual address space

Fig. 3. Example virtual address space (VAS) with different scaling factors used for different memory regions. Gray regions denote unused memory.

Advantages: Address Scaling addresses all the three challenges involved in run-time monitoring outlined in the previous section. First, since the data and its associated metadata reside in the same cache line, it is easy to locate the metadata. Second, as the processor accesses memory at the granularity of a cache line, accessing the data also fetches its corresponding metadata into the L1 cache. Hence, the metadata access following the data access will result in a cache hit. Third, ensuring atomicity across the data and the metadata accesses now requires providing *single cache line atomicity*, which can be implemented using *delayed coherence*.

Design Considerations: There are five components in the design of hardware support for *Address Scaling*: 1) determining how much to scale — i.e., determining data-metadata ratio, 2) computing the scaled address, 3) semantics for accessing metadata, 4) semantics for ensuring atomicity between data and metadata accesses, and 5) support for different scaling factors. We describe each of the components below.

A. How Much to Scale?

When the process encounters a load/store instruction, it should first determine how much to scale the virtual address. Depending on the use case, this can be determined based on how much metadata needs to be stored for a given region of the virtual address space. For example, the region of the address space that stores kernel data or shared libraries may not require any metadata.

To determine the ratio between the size of data and metadata, we divide the virtual address space into a number of *segments*. Each segment is associated with a parameter called *scaling factor*, which determines the size of sub-cache-line for that segment. The scaling factor is represented using a pair of integers *(d, m)*, which indicates that every m bytes of metadata is associated with every d bytes of data. Since *Address Scaling* stores a piece of data and the associated metadata in the same cache line, it requires $d + m$ to be a factor of the cache line size. For example, for a system using a 64-byte cache lines, a scaling factor of (6, 2) will store 48 bytes of data followed by 16 bytes of metadata in every cache line.

B. Scaled Address Computation

In this section, we describe how we can compute the scaled address by taking an example of Memcheck on top of our framework. We will assume that the size of a cache line is 64 bytes. An optimized version of Memcheck stores two bits of metadata for each byte of data [7]. Therefore, *Address Scaling* uses a scaling factor of (6, 2) i.e., a sub-cache-line size of 48 bytes, leaving a hole of 16 bytes, which is sufficient to store all the metadata for the 48 bytes. When the processor receives an access for a virtual address X_v, it computes the base address of the scaled cache line, X_s, using the following equation:

$$X_s = \left\lfloor \frac{X_v}{48} \right\rfloor 64 \tag{1}$$

The offset within the virtual cache line is used to locate both the data and the metadata within the scaled cache line. The above notion of scaling can be generalized to different sizes of metadata per byte.

Since computing the scaled address is on the critical path of data and metadata access. Based on our analysis of different monitoring tools, practical values for the sub-cache-line size for a 64-byte cache line are 56, 48, 40, 32, 16, and 8. Similar to (1), computing the scaled address requires integer division by the sub-cache-line sizes. While such division is trivial for 8, 16, and 32, for sub-cache-line sizes 56, 48, and 40, this computation reduces to division by 7, 3, and 5. For our mechanism to perform well, we need to build fast hardware to perform these divisions.

C. Support for Different Scaling Factors

To support varied metadata granularities, we partition the virtual address space into regions with different scaling factors (including regions without support for *Address Scaling*, as shown in Fig. 3). A few approaches to determining the scaling factor are: 1) using the upper bits of the virtual address, this enables the processor to quickly check the scaling factor by simply comparing the base-bound pair of each region with the virtual address, 2) storing it in the page table entry (PTE), and 3) a separate hardware structure. We leave the exploration of determining the scaling factor as part of future work.

D. Application Interface

In this section, we describe how an application can utilize *Address Scaling*. Let us take the example of AddressSanitizer [6]. It maintains one byte of metadata for every 8 bytes of application data. When AddressSanitizer tries to allocate memory using our modified version of memory allocators such as `malloc`, it indicates to the operating system (OS) that it wants to use a specific scaling factor for that particular data structure. The OS then allocates virtual memory for that data structure in the corresponding region in the virtual memory which has a 8:1 scaling ratio. Subsequently, the hardware begins scaling the virtual memory accesses to the data structure, thereby generating space for one byte of metadata for every 8 bytes of application data within each cache line. AddressSanitizer accesses this metadata using new ISA instructions. We leave the semantics of the instructions for a longer write-up.

III. Evaluation

To quantitatively evaluate the benefits of *Address Scaling*, we implement our framework within Virtuoso [11], a new virtual memory system simulator based on Intel's Sniper [12] for the x86 architecture. Table I shows some of the key configurations of the simulated system. We have maintained small cache sizes to stress the cache hierarchy and highlight the worst-case overheads of *Address Scaling*. We demonstrate performance using two microbenchmarks that compute the sum of a large integer array — one with sequential and the other with random memory access pattern. In both these situations, the memory access is the critical bottleneck. Since we are executing out of order, the only critical path in the program are the memory accesses, and with our mechanism, for every memory access we are doing metadata checks in software, leading to a significant overhead.

TABLE I
Main Parameters of Our Simulated System

Processor	x86, out-of-order, 2.6 GHz
L1 cache	16 KB D-cache, 32 KB I-cache, LRU policy
L2 cache	64 KB, LRU policy, 64B cache line size
L3 cache	2 MB, LRU policy
Main memory	DDR4-3200, 3-channel, 2-rank, 16 banks

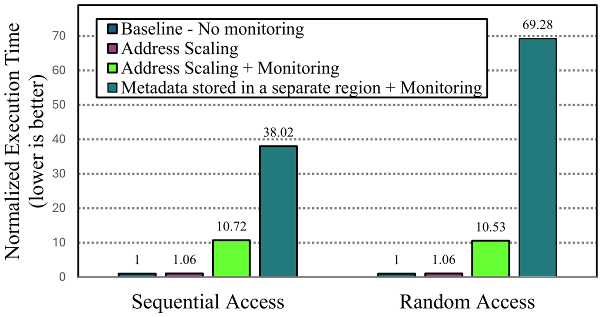

Fig. 4. Execution time for running *Address Scaling* with Memcheck (monitoring) compared to existing systems where metadata is stored elsewhere (normalized to a baseline that does not perform monitoring). Each cache line stores 48B data + 16B metadata; i.e., a scaling factor of (6, 2).

A. Memcheck With Address Scaling

Address Scaling's benefits come from eliminating cache misses for the metadata, making metadata accesses almost free, albeit at the cost of reduced cache space for the data. We show that this trade-off results in significant performance improvement compared to existing approaches.

We quantitatively show the benefit of our approach by comparing it with Memcheck. For these experiments, we run the microbenchmarks under four settings: 1) a baseline system that does not perform any monitoring of the application, 2) application running with our implementation of Memcheck along with a software mechanism that stores metadata in a separate shadow memory region, 3) *Address Scaling* mechanism in the hardware that reserves the space for metadata but does not perform any monitoring, and 4) *Address Scaling* in the hardware along with metadata operations in software that Memcheck requires.

Fig. 4 shows that Memcheck incurs very high overheads, slowing down the application by as much as 38× (sequential access) and 69× (random access). In contrast, the overhead of *Address Scaling* is 3× and 6× lower than Memcheck, respectively. Since our microbenchmarks involve only memory accesses and metadata checks, the 10× overhead in some sense serves as an upper bound on the potential slowdown induced by *Address Scaling*. We are working on further reducing these slowdowns by simulating those software metadata instructions using ISA changes. Therefore, this experiment shows the promise of

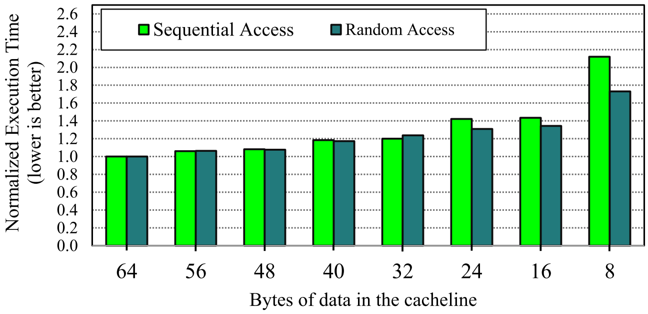

Fig. 5. Performance impact of varying the amount of data bytes in the cache line (normalized to a 64B cacheblock size). The overhead is moderate as long as the metadata size is less than data.

co-locating data and the corresponding metadata in the same cache line.

B. Address Scaling Characterization

Since *Address Scaling* reduces the effective cache space available for an application (as it allocates a portion of each cache line for storing metadata), it can introduce some performance overheads. To evaluate these overheads, we capture the run-time of our microbenchmarks by varying the amount of metadata stored in each cache line and compare it with the baseline system i.e., a system without any monitoring or *Address Scaling*. Fig. 5 shows that the amount of metadata has a direct impact on application run-time. However, the overhead is moderate (<20%) as long as the size of metadata is modest i.e., when most of the cache space is allocated to application data. Note that we expect this to be a common case for *Address Scaling* based tools.

IV. Related Work

Prior works [1], [3], [4], [8], [10] primarily differ in 1) how and where they maintain the metadata, and 2) their target monitoring application. For example, TaintTrace [13] simply splits the virtual address space into two parts, where one part is used to store the data while the other stores the metadata. Hardbound [10] proposes a mechanism that extends every word of memory in the virtual address space and the registers to augment the metadata. These mechanisms provide ISA support to locate and access metadata.

Sasaki et al. [1] propose a mechanism called Califorms with the aim of providing byte-granular memory safety. They achieve this by storing metadata in unused memory spaces within the cache line. While this approach is effective, it necessitates disruptive modifications to the cache hierarchy. In contrast, *Address Scaling* stores metadata at an offset from the data within the cache line, maintaining consistency throughout the cache hierarchy. Additionally, *Address Scaling* aims to support use-cases beyond memory safety.

V. Future Work

While we evaluated Memcheck [2] to demonstrate the potential benefits of *Address Scaling*, the goal of our framework is to enable a variety of runtime monitoring tools. To better demonstrate this, we will implement AddressSanitizer (ASan) [6] on top of our *Address Scaling* framework, and compare it to hardware-assisted ASan [14].

Address Scaling as a framework can enable several new use-cases to improve system reliability, security, and memory optimizations. We describe one such use-case below.

Heterogenous ECC: Modern processors maintain ECC bits alongside cache blocks in their on-chip caches. However, not all cache blocks may require the same level of ECC. For example, cache blocks that are accessed multiple times often contain critical data and may require a more robust ECC than cache blocks that are accessed less frequently. Hence, depending on the desired level of reliability the software needs to maintain varying amounts of metadata. Therefore, ECC serves as a perfect use-case for our proposed framework. Our approach to ECC can also protect sensitive data against RowHammer style attacks. For example, embedding a 64-bit MAC with a cache line containing 56 bytes of data enables quick verification of the data integrity in the cache line.

Acknowledgment

The authors would like to thank Ian Glen Neal for the brainstorming sessions.

References

[1] H. Sasaki, M. A. Arroyo, M. T. I. Ziad, K. Bhat, K. Sinha, and S. Sethumadhavan, "Practical byte-granular memory blacklisting using califorms," in *Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2019, pp. 558–571.

[2] J. Seward and N. Nethercote, "Using Valgrind to detect undefined value errors with bit-precision," in *Proc. USENIX Annu. Tech. Conf., Gen. Track*, 2005, pp. 17–30.

[3] M. Herlihy and J. E. B. Moss, "Transactional memory: Architectural support for lock-free data structures," in *Proc. 20th Annu. Int. Symp. Comput. Architecture*, 1993, pp. 289–300.

[4] S. Nagarakatte, M. M. Martin, and S. Zdancewic, "Watchdog: Hardware for safe and secure manual memory management and full memory safety," *ACM SIGARCH Comput. Architecture News*, vol. 40, no. 3, pp. 189–200, 2012.

[5] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," *ACM SIGPLAN Notices*, vol. 42, no. 6, pp. 89–100, 2007.

[6] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A fast address sanity checker," in *Proc. USENIX Annu. Tech. Conf.*, 2012, pp. 309–318.

[7] N. Nethercote and J. Seward, "How to shadow every byte of memory used by a program," in *Proc. 3rd Int. Conf. Virtual Execution Environ.*, 2007, pp. 65–74.

[8] V. Nagarajan and R. Gupta, "Architectural support for shadow memory in multiprocessors," in *Proc. ACM SIGPLAN/SIGOPS Int. Conf. Virtual Execution Environ.*, 2009, pp. 1–10.

[9] V. Seshadri et al., "Page overlays: An enhanced virtual memory framework to enable fine-grained memory management," *ACM SIGARCH Comput. Architecture News*, vol. 43, pp. 79–91, 2015.

[10] J. Devietti, C. Blundell, M. M. Martin, and S. Zdancewic, "HardBound: Architectural support for spatial safety of the C programming language," *ACM SIGOPS Operating Syst. Rev.*, vol. 42, no. 2, pp. 103–114, 2008.

[11] K. Kanellopoulos, K. Sgouras, and O. Mutlu, "Virtuoso: An open-source, comprehensive and modular simulation framework for virtual memory research," *arXiv*.

[12] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2011, pp. 1–12.

[13] W. Cheng, Q. Zhao, B. Yu, and S. Hiroshige, "TaintTrace: Efficient flow tracing with dynamic binary rewriting," in *Proc. IEEE 11th Symp. Comput. Commun.*, 2006, pp. 749–754.

[14] S. Kostya, S. Evgenii, S. Aleksey, T. Vlad, and V. Dmitry, "HWASAN: An AArch64-specific compiler-based tool," 2018.