Splitn Trace NVM: Leveraging Library OSes for Semantic Memory Tracing

Christian Hakert, Kuan-Hsun Chen, Simon Kuenzer, Sharan Santhanam, Shuo-Han Chen, Yuan-Hao Chang, Felipe Huici and Jian-Jia Chen

Department of Computer Science, TU Dortmund, Germany
https://ls12-www.cs.tu-dortmund.de/

Citation: https://doi.org/10.1109/NVMSA51238.2020.9188136

Split'n Trace NVM: Leveraging Library OSes for Semantic Memory Tracing

Christian Hakert*, Kuan-Hsun Chen*, Simon Kuenzer†, Sharan Santhanam†, Shuo-Han Chen‡, Yuan-Hao Chang‡, Felipe Huici†, Jian-Jia Chen*
*Department of Informatics, TU Dortmund, Germany
† NEC Laboratories Europe GmbH, Germany
‡ Institute of Information Science, Academia Sinica, Taiwan

***Abstract*—With the rise of non-volatile memory (NVM) as a replacement for traditional main memories (e.g. DRAM), memory access analysis is becoming an increasingly important topic. NVMs suffer from technical shortcomings as such as reduced cell endurance which call for precise memory access analysis in order to design maintenance strategies that can extend the memory's lifetime. While existing memory access analyzers trace memory accesses at various levels, from the application level with code instrumentation, down to the hardware level where software is executed on special analysis hardware, they usually interpret main memory as a consecutive area, without investigating the application semantics of different memory regions.**

In contrast, this paper presents a memory access simulator, which splits the main memory into semantic regions and enriches the simulation result with semantics from the analyzed application. We leverage a library-based operating system called Unikraft by ascribing memory regions of the simulation to the relevant OS libraries. This novel approach allows us to derive a detailed analysis of which libraries (and thus functionalities) are responsible for which memory access patterns. Through offline profiling with our simulator, we provide a fine-granularity analysis of memory access patterns that provide insights for the design of efficient NVM maintenance strategies.

***Index Terms*—unikernel, system simulation, memory simulation, memory analysis**

I. Introduction

Beside the total amount of accesses, memory access patterns can have a large impact on the performance and lifetime of emerging memories. For non-volatile memory (NVM) in particular, certain patterns can result in severely reduced memory lifetime due to a very limited cell endurance. To design mitigation solutions, it is imperative to be able to accurately measure and analyze such access patterns.

In this paper, we propose a novel simulator concept, where we combine a library-based operating system called Unikraft [9] with a full-system simulator and subsequent memory trace analysis (the latter based on our previous work [7]). A library operating system splits its functionality into fine-granularity libraries, and often provides clear APIs to be able to seamlessly exchange libraries (e.g., different memory allocators or schedulers). We leverage such fine-grained libraries to split the memory and analyze the memory access behavior of each library in isolation; this allows us to assess each library *individually*, based on its memory access behavior, with respect to arbitrary metrics.

To allow isolated library analysis, we propose two techniques: static memory analysis and dynamic memory analysis. For static memory analysis we investigate the final linked OS image, including all libraries and application. We identify statically allocated main memory regions (i.e. text, data, bss) and map them to the originating library by comparing compiler symbols with the pre-compiled libraries; this results in a per-library memory map which we apply to the memory trace, acquired by our simulator.

For dynamic memory analysis, the aim is to achieve the same output, but for run-time allocated memory (i.e. stack, heap). We extend a full-system simulator to trace-out the program counter of every issued memory access. In combination with the static library-memory map, we can therefore map each memory access to the library that triggered the access.

We conduct a case study by analyzing both, the static and the dynamic allocated memory regions on a *per-library* basis in order to determine the effect of the caused memory access pattern on the memory wear out. This subsequently provides important insights into which libraries to tackle first when designing solutions to improve memory lifetime. Our methodology is not limited to the presented simulation setup, it can also operate with other library based operating and other memory access simulators. In short, our main research contributions are:

- An extended memory access simulator, built on top of a full-system simulator, which leverages Unikraft, a library operating system. Our implementation provides the necessary hardware drivers for Unikraft to be used with the gem5 full-system simulator [3].
- A static analysis module that maps static memory regions of the final linked binary to the originating libraries, allowing for fine-granularity, per library tracing of memory accesses.
- A dynamic analysis module that determines the issuing library for each memory access. This allows us to further analyze dynamic allocated memory (i.e. stack and heap), but also investigate the difference between ownership and usage of static allocated memory.
- A case study that uses both methods to show how individual libraries impact the lifetime of main memory.

978-1-7281-8482-1/20/$31.00 ©2020 IEEE

II. Related Work

Work in the literature proposes many different ways how to analyze the memory access behavior of computer applications. Capturing of memory accesses is proposed to be done on various levels. Bao et al. propose HMTT [2], which is a hardware based memory simulator that uses an FPGA between the processor and the memory DIMM to snoop all memory accesses. A less invasive method is provided by the combination of the gem5 full-system simulator [3] and the non-volatile memory simulator NVMain2.0 [13]; this setup simulates an entire system, including CPU, memory and peripherals. The simulator then allows users to trace all memory accesses. Nethercote et al. propose Valgrind [11], which is a method that hooks in at the application level and requires code instrumentation such that every memory access can be traced out.

The aforementioned mechanisms are subsequently used to analyse applications memory behavior. Jiang et al. focus on a memory analysis of computing frameworks (e.g. Hadoop) by investigating hardware characteristics (e.g. cache behavior) [8]. They employ Intel's VTune Amplifier [1] to collect architectural metrics, but also conduct their analysis on the HMTT platform to acquire the full memory trace. Nalli et al. present a benchmark suite with various applications [10], using the gem5 simulator to analyze memory traces and to investigate the memory behavior of these applications. Byma et al. perform an analysis of the usage of heap allocated memory [4] by instrumenting the code, in a similar manner as Valgrind [11] does. Consequently, they analyse the memory behavior of their application based on these data.

To the best of our knowledge, all this work focuses on global analysis of the target applications. They usually compute metrics for the overall memory behavior and do not try to divide the application into separate functional units that can be analyzed isolated. In this work, we propose a novel method to achieve this, where we employ the library based operating system Unikraft within a full-system simulator. We exploit the organization into libraries for the memory analysis to investigate the memory pattern of each library in isolation.

III. Simulator Architecture

In this paper, we propose an entire setup for isolated memory access pattern simulation and analysis. This setup features two important components: 1) the full-system simulator, which allows us to trace out all memory accesses during the execution of an operating system, and 2) the software we run on top of this simulator, which then allows us to trace back every memory access to the software library, which caused the access. As mentioned in Section II, there exists a variety of methods to acquire a memory trace from an executing operating system. It is beyond the scope of this paper to assess memory simulation methods and choose the best among them. We utilize the full-system simulator gem5 [3] in combination with the NVM simulator NVMain2.0 [13] in this work for the following reasons. First, full-system simulations are not invasive and do not influence the memory behavior of the analyzed application. Second, full-system simulations can be run as an application on common linux computers, which eases the processing of simulation results. Third, the combination of gem5 and NVMain2.0 is already well studied for analyzing memory access patterns by us in [7]. We base our methods on top of this work to provide a simulation setup for advanced memory access pattern analysis, while we maintain broad compatibility with the existing methods.

The previous simulation setup [7] ships with a custom operating system, which is dedicated to separate the memory accesses from the application and the operating system. This is crucial to analyze the isolated memory behavior of an application, but not suited to investigate the interplay between the application and the single components of the operating system. Therefore, we use Unikraft as the operating system, which we execute on top of the full-system simulator. Unikraft is a library based operating system and therefor divides the operating system functionalities into small libraries. By analyzing the memory access pattern of each library separately, we can investigate the interplay between the application and all the libraries and the effects on the memory access patterns. Our implementation provides the required code modifications and device drivers to run Unikraft on gem5. Unikraft itself provides a large source repository with a wide choice of libraries, which allow many applications to run within Unikraft. Therefore, the choice of Unikraft not only allows us to perform precise analysis later on, it also allows to run different applications and analyze them. Since we still employ NVMain2.0 in the simulation process, we achieve a precise timing simulation of the underlying NVM.

IV. Modular Analysis

As mentioned before, we utilize the library structure of unikraft to analyze the memory access pattern of every library separately. This allows the generation of single memory traces for libraries like memory allocation, boot code, scheduling all the way to libraries such as openssl and boost [14]. OS primitive libraries reside in the main Unikraft repository, while all others have own repositories that can be separately added to a build configuration. Once configured, Unikraft's build system builds all libraries and links them together, generating a single executable binary file with a single memory address space.

1) Binary Analysis and Memory Trace Indication: As mentioned before, the linker merges all separate libraries and modules into one single address space, meaning that, for instance, the text segments of all libraries, are merged into one global text segment to keep all the compiled binary code spatially close. While the applications performance may benefit from this technique, it makes its analysis more complex. To overcome this, we extend the existing simulation setup with a post-simulation analysis module which traces back the original software component for every memory section.

In detail, during compilation debug symbols are added to the final, single address-space binary file. These symbols indicate, among others, the memory location of variables, arrays or functions. The symbols are each placed at the first byte of the corresponding variable, array or function. Therefore, the memory region between two symbols belongs to the high level construct from the first symbol. The symbols are generated during compilation and not changed (only placed) during linking. As a result, symbols can be found in the single precompiled libraries as well as in the final linked binary file. Due to the fact that symbols are unique, symbols from the final linked binary can be linked to a single library.

With the help of these symbols we scan the final, compiled binary for all symbols and check, for each symbol, from which library it comes from. This tells us that the memory region between this and the next symbol belongs to that library. We subsequently create a *library-memory map* that we can use for static analysis (e.g., to determine how much memory each library consumes). Once we run the simulator and receive the memory trace, we take the library-memory map into account to indicate the libraries in the memory trace[1]. The memory trace itself contains access to the main memory, therefore accesses which may be covered by caches are not considered, since they also have no influence on the memory. Basically, all analysis modules from [7] can be still combined with this library map as long as they output a result which includes memory addresses.

To analyze memory wear-out later in this paper, we focus on the cumulative number of write accesses per byte to analyze the lifetime impact of software on main memory. Consequently, we first extend the analysis script to calculate wear-out indicators for each library by only taking their corresponding memory cells into account; this provides a quantitative value of the impact on memory lifetime of every library. We further extend the analysis script to produce a plotted memory write count map, coloured by single libraries, allowing for separate analysis of the memory access patterns of each library.

2) Dynamic Memory Analysis: As detailed above, we utilize the compiler-generated symbols to identify the memory regions of each library. Naturally, this only works for static allocated memory (text, data, bss), but not for dynamically allocated memory (heap and stack). To overcome this, we introduce a specialized analysis module that collects further information during the simulation. To estimate, for a given memory access, which library triggeres it, we investigate the current program counter.

Note that this does not necessarily reveal the memory ownership, since libraries can access memory of other libraries. This method rather indicates how frequent which library uses a certain memory region. Additionally, the current program counter only indicates the currently executing library and not the calling library, which is responsible the access. Although stack frames (caused by function calls) could be investigated to track down a memory access to the most outer calling library, it is unclear how far the calls should be traced back. For instance, the most outer calling function would always be the

[1]Note that we use an identity mapping between virtual and physical memory addresses. When an arbitrary mapping is used, the virtual memory address has to be traced out next to the physical address by the simulator to lookup a write access in the library-memory map

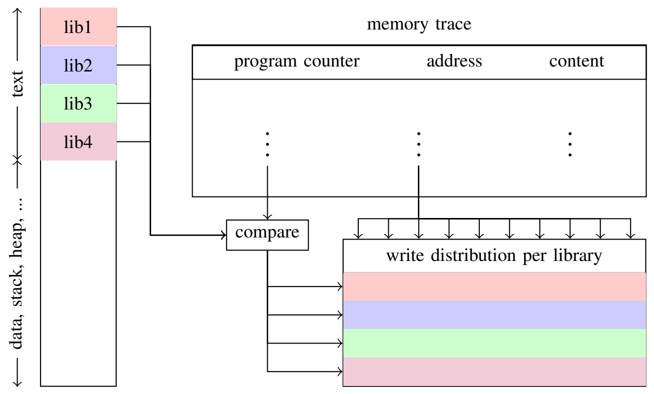


Fig. 1: Dynamic memory trace analysis[2]

initialization function of the thread, which would not deliver any new insight. Since it is not clear how to properly figure out the responsible function call, we only focus on the current program counter.

To provide this functionality, we first extend gem5 / NVMain2.0 to pass the program counter along with a memory request issue to the memory subsystem and to the memory trace writer. Because gem5 is a full system simulator, this information can be extracted easily. This leads to an additional column in the tracefile of our simulation setup that records the program counter address together with each memory access.

As a second step, we provide an analysis module that interprets the additional information in the tracefile. Leveraging the static library-memory map and the program counter, we can identify the library that issued the memory request. The text segment is static and distinctly separated by functions by the compiler, thus any address within the text segment (i.e. every value the program counter can have) belongs unambiguously to one function of one library. We scan the memory trace access by access, identify the calling library of each access and aggregate the number of writes per cell for every library separately. This mechanism is illustrated in Figure 1. This mechanism has no further limitations: it can be applied to every memory region and can analyze the memory traces separately. While such an analysis is crucial to analyze dynamic managed memory regions (e.g. stack, heap) for each library, it can also be applied to statically-allocated memory regions (e.g. data, bss). By doing this, the analysis can give insights into which library is actually using memory owned by other libraries.

V. Case Study: SQLite Database

The previous section details our setup for an advanced simulation environment, which allows further and detailed analysis of single operating system components when it comes to recording accesses to non-volatile main-memory. To illustrate which types of insights can be gathered by using

[2]The figure illustrates how the memory trace from the simulator is processed to analyze the memory write distribution for each library separately. For each access (i.e. line in the memory trace), the issuing program counter is compared with the text segment fragments of the libraries and in therefore, the issuing library is identified. The access address is then used to increment the corresponding counter in the write distribution for this specific library.

such a simulator, we provide a case study on the memory lifetime impact of an SQLite database application, analyzing the memory access patterns of each individual library and noting which libraries have the biggest impact on memory wear out and thus memory lifetime; this kind of analysis can provide key insights into the design of specialized, minimal wear-leveling mechanisms.

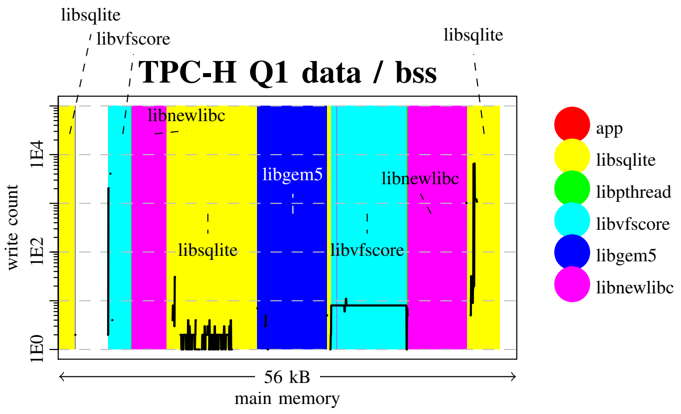
A. *SQLlite TPC-H Implementation*

As the application, we run the TPC-H [5] queries Q1 and Q6 on a scaled lineitem table (500 rows) in a Unikraft SQLite image. To provide this functionality, the image includes an sqlite3 implementation [12], the required system libraries (libc and libpthread) for sqlite, a virtual file system layer (libvfscore) used to store the database, and an application that populates the table and executes the query. These libraries are all available from the Unikraft source repository[3]. All these are configured into one instance of Unikraft and executed on top of our simualtion setup.

In this case study, we utilize our proposed simulation setup to separate memory traces for each library and identify memory access hotspots, which finally have the biggest influence on the memory lifetime, when they target NVM with low write endurance. As a consequence of this result, specific software components can be replaced by alternative implementations or can be mapped to volatile memory or be tackled by wear-leveling, thus the memory lifetime can be increased. The most drastic influence on memory lifetime is the hotspot where the highest amount of writes is applied to the same memory region, because this memory region would wear-out first. In the following, we analyze static allocated memory regions (data and bss), as well as a dynamic allocated memory region (stack) for such memory hotspots and determine, from which operating system library they stem from.

B. *Static Memory Analysis*

We begin by investigating the statically-allocated memory, i.e. the data and bss segments. Although the text segment is statically allocated, it only experiences read accesses during execution and thus it is not meaningful to analyze it when considering memory wear out. The data and bss segment both store program data, such as variables, arrays or datatype instances. Each library therefore has a different memory usage on the data / bss segment and therefore has different semantics for accessing the memory, leading to different memory access patterns for each library. Figure 2 shows the output of our simulation setup for the data and bss segment of the TPC-H Q1 and Q6 benchmark. As a first observation, it can be seen that the application (red) has no present memory regions in the data / bss segment. This stems from the fact that the application only coordinates the sql calls and printing of the results; as



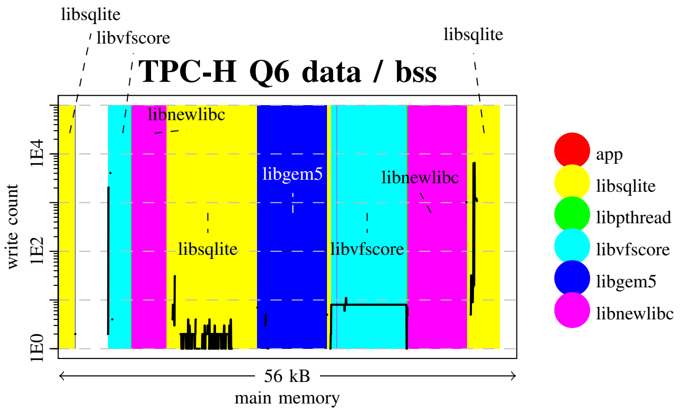


Fig. 2: data / bss section for TPC-H Q1 and Q6[4]

a result, no program memory is needed, since everything can be stored locally on the stack. As expected, the sql library (libsqlite) takes a central role during the benchmark execution, which also can be seen in the memory hotspot at the end of the data / bss segment. Analyzing the data / bss segment, libsqlite and libvfscore result in the most evident memory hotspots and thus represent prime candidates for wear-leveling mitigation (or for being relocated to volatile memory).

C. *Dynamic Memory Analysis*

As we noted in [7], dynamic memory (i.e. the stack) causes much more intensive write hotspots than the statically-allocated segments (i.e. data, bss). Nevertheless, the stack may be allocated to non-volatile memory with limited write-endurance for various technical reasons, therefore the hotspots should be identified.

We apply our method for dynamic memory analysis (Section IV-2) to analyze write accesses to the main stack. As our benchmark is a single-core benchmark, only one stack is required. If, however, multiple stacks are used, they can be all analyzed separately. To know the position of the stack, we instrument Unikraft with a simple debug output on the allocation of thread's stacks. In Figure 3, we illustrate the amount of write accesses to the stack by each library. Note that the lines only count the accesses each library performs, therefore the "real" amount of write accesses to the stack is given by the sum of all lines. This illustration allows to easily identify the library that would degrade stack memory the fastest: libpthread. Although the static analysis shows that

[3]https://www.github.com/unikraft

[4]The figures show the cumulative number of writes (y axis) over the memory bytes of the data and bss segment (x axis). The ownership of the memory regions by the libraries is indicated by the colours (legend on the right). Note that the y axis is in logarithmic scale, thus memory regions with zero write accesses cannot be indicated.

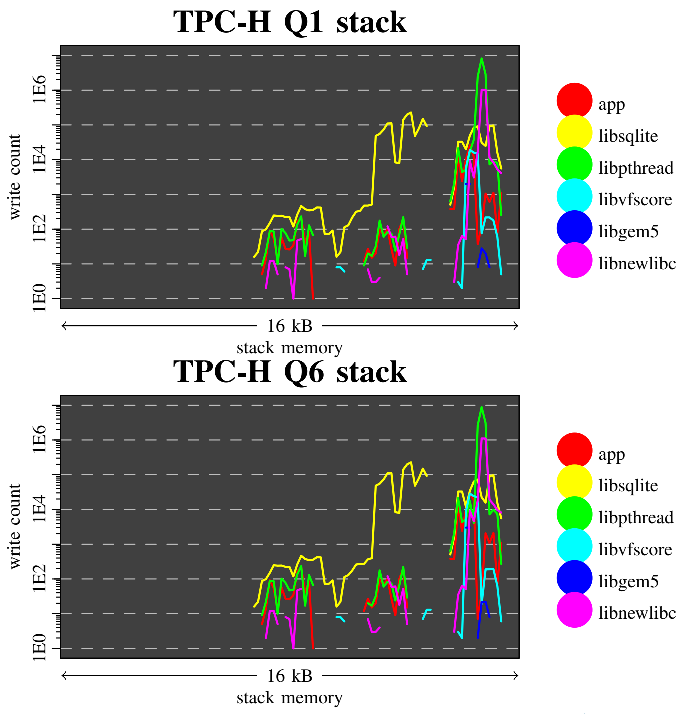


Fig. 3: stack section for TPC-H Q1 and Q6[5]

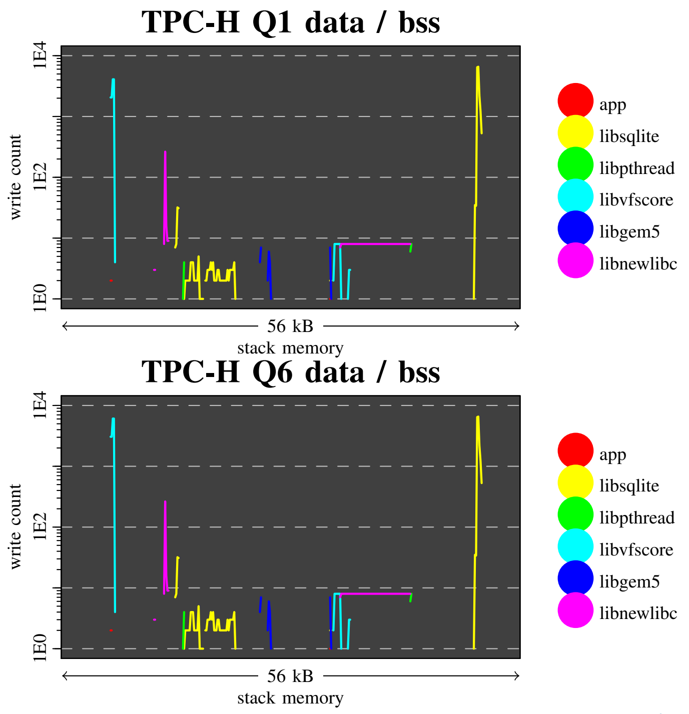


Fig. 4: dynamic data / bss section for TPC-H Q1 and Q6[6]

the sql library (libsqlite) performs the most dense memory accesses in the data / bss segment, this is different for the stack. The threading abstraction (libpthread), as well as the c library (libnewlibc) both cause a memory hotspot larger by two orders of magnitude than the highest peak usage for libsqlite. As a result, and perhaps somewhat counter-intuitively, any wear-out mitigation mechanism should first look into modifying these libraries before tackling the ”obvious’ libsqlite library.

To finalize this case study, we combine the static and dynamic analysis. To achieve this, we run the dynamic analysis on the static allocated memory region (data / bss). This then results in a figure that indicates which library accesses correspond to which memory region. The results are shown in Figure 4. As can be seen, for most memory regions the owning library accesses its own memory segments. However, it is also clear that the virtual file system (libvfscore) practially does not access the second portion of data / bss memory. Instead, this is only accessed by the c library, due to library functions which operate on this memory.

In addition to the graphical illustration, our simulation setup also allows to perform arbitrary analysis on the collected and separated data. For instance, lifetime indicators can be calculated for every single library to assess the memory usage in terms of lifetime. This can also be used to identify libraries with the worst impact on memory lifetime. To illustrate this, we take the achieved endurance (AE) metric (Equation (1)) [6] into account:

$$AE = \frac{\text{mean\_write\_count}}{\text{max\_write\_count}} \tag{1}$$

This metric gives an intuition as to which level of improvement can be achieved by wear-leveling. For instance, if the achieved endurance is low, wear-leveling has high potential to improve the memory lifetime. When calculated for every single library, this metric can be used as an indicator of which library may be worth the effort for performing wear-leveling. We determine the achieved endurance for every library for the stack (compare Figure 3) and the data / bss (compare Figure 4). Both data sets are acquired by our dynamic analysis mechanism. The achieved endurance is calculated in all cases across the entire segment and is not limited to the memory regions from the library only. Table I shows the resulting achieved endurance numbers. As shown, libnewlibc has the highest optimization potential in the data segment, while libpthread shows the highest optimization potential in the stack segment. It is also worth noting that the optimization potential is not always correlated for each library: the sql library (libsql), for instance, has relatively high optimization potential in the data segment, but not in the stack segment.

	app	libsqlite	libpthread	libvfscore	libgem5	libnewlibc
Q1						
stack	2.44%	7.80%	1.53%	2.74%	1.87%	1.89%
data	1.17%	0.69%	0.62%	0.79%	1.27%	0.47%
Q6						
stack	2.53%	7.47%	1.52%	2.73%	1.85%	1.90%
data	1.17%	0.69%	0.62%	0.79%	1.27%	0.40%

TABLE I: achieved endurance after dynamic analysis

[5]This figure illustrates the cumulative number of write accesses (y axis) to the stack (x axis) per library (coloured lines). The lines each indicate the number of accesses per library, thus the sum of all lines would be the real amount of writes.

[6]This figure depicts the same memory accesses as Figure 2. In contrast, we use the dynamic analysis method (Section IV-2) to show which library accesses the memory during runtime. In comparison to Figure 2, this depicts the difference in memory ownership and usage.

Summarizing this case study briefly, it can be observed that the per-library modular analysis, we propose in this paper, delivers important insights into how to design efficient wear-leveling and memory lifetime improvement techniques. The results are counter-intuitive for some libraries, as well as they point out the need to distinguish between memory regions and libraries and assess and tackle them with separate means regarding their memory wear out.

VI. Conclusion

In this paper, we propose a novel module-based memory access simulator for NVM which utilizes the features of a configurable library operating system to provide separation between single OS components. We base our full-system simulator on our previously proposed combination of gem5 and NVMain2.0 [7]. In addition to this setup, we replace the original custom run-time system with Unikraft, a configurable library OS [9]. On this basis, we extended our simulation setup with analysis modules that keep track of these libraries and indicate main memory regions on a per-library basis.

With this in place, we first provide a static analysis mechanism which investigates debug symbols in the compiler output to find statically-allocated main memory for every library. This information could be used on its own to, for instance, estimate the library's memory consumption, but we also combine it with the memory access simulator. Thus, for each memory access, the target memory region can be ascribed to one operating system component. Second, we provide a dynamic analysis mechanism, which separates the run-time behavior of each library. Due to a specific extension of gem5, we acquire the causing program counter for every memory access. In combination with the static analysis, we can clearly identify the program counter within the compiled code of one library. Therefore we can unambiguously identify the causing library for every memory access and collect a separate memory access trace for every library.

In the case study of this paper, we provide an intuition for the use cases and benefit of our proposed simulation setup. We focus on the analysis of the cumulative number of writes per memory cell, which is an important indicator for the lifetime of NVMs with limited write endurance. We show how memory write hotspots can be identified and tracked back to an individual library. This knowledge then can be used to resolve these hotspots by specifically targeting the library. We further point out a difference in the ownership and usage of static allocated memory. Finally, we analyze potential lifetime improvement for each library separately and discover that for each distinct memory segment (i.e. data or stack segment), different libraries present the highest potential for lifetime improvement.

VII. Outlook

To the best of our knowledge, we are the first to show the potential combination of library operating system and memory access simulators and explore this for designing maintenance for non-volatile-memory. Although we show in this work that crucial insights can be acquired by this novel simulation setup, we believe that this setup has high potential for further memory analysis as well. The library separation, combined with the fact that for each library multiple implementations may exist and be interchanged, allows for a high grade of configurability, which yields the need to assess the quality of every configuration and to optimize configurations. For future work, we plan to apply further analysis, such as cache replacement analysis and memory deduplication analysis to our simulator to also improve among these.

Since we believe novel memory behavior analysis to be crucial to designing future memory mechanisms, we plan to publish all our implementation. The analysis implementation and the modified simulator are provided at https://github.com/tu-dortmund-ls12-rt/splitntrace_analysis, the gem5 port of unikraft will be upstreamed in the official unikraft repository.

Acknowledgement

This paper has been supported by Deutsche Forschungsgemeinschaft (DFG), as part of the project OneMemory (project number 405422836) and the project SFB876 A1 (project number 124020371). It is partly funded by the Horizon 2020 Framework Programme of the European Union under agreement No 825377 (UNICORE).

References

[1] Intel Vtune Amplifier. Intel vtune amplifier, 2019.

[2] Yungang Bao, Mingyu Chen, Yuan Ruan, Li Liu, Jianping Fan, Qingbo Yuan, Bo Song, and Jianwei Xu. Hmtt: a platform independent full-system memory trace monitoring system. In *Proceedings of the 2008 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 229–240, 2008.

[3] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.

[4] Stuart Byma and James R Larus. Detailed heap profiling. In *Proceedings of the 2018 ACM SIGPLAN International Symposium on Memory Management*, pages 1–13, 2018.

[5] Transaction Processing Performance Council. Tpc-h benchmark specification. *Published at http://www. tcp. org/hspec. html*, 21:592–603, 2008.

[6] Christian Hakert, Kuan-Hsun Chen, Paul R. Genssler, Georg Brüggen, Lars Bauer, Hussam Amrouch, Jian-Jia Chen, and Jörg Henkel. Softwear: Software-only in-memory wear-leveling for non-volatile main memory. *CoRR*, abs/2004.03244, 2020.

[7] Christian Hakert, Kuan-Hsun Chen, Mikail Yayla, Georg von der Brüggen, Sebastian Bloemeke, and Jian-Jia Chen. Software-based memory analysis environments for in-memory wear-leveling. In *25th Asia and South Pacific Design Automation Conference ASP-DAC 2020, Invited Paper*, Beijing, China, 2020.

[8] Tao Jiang, Qianlong Zhang, Rui Hou, Lin Chai, Sally A Mckee, Zhen Jia, and Ninghui Sun. Understanding the behavior of in-memory computing workloads. In *2014 IEEE International Symposium on Workload Characterization (IISWC)*, pages 22–30. IEEE, 2014.

[9] S. Kuenzer, S. Santhanam, Y. Volchkov, F. Schmidt, F. Huici, Joel Nider, Mike Rapoport, and Costin Lupu. Unleashing the power of unikernels with unikraft. In *Proceedings of the 12th ACM International Conference on Systems and Storage*, SYSTOR 19, page 195, New York, NY, USA, 2019. Association for Computing Machinery.

[10] Sanketh Nalli, Swapnil Haria, Mark D Hill, Michael M Swift, Haris Volos, and Kimberly Keeton. An analysis of persistent memory use with whisper. *ACM SIGPLAN Notices*, 52(4):135–148, 2017.

[11] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan notices*, 42(6):89–100, 2007.

[12] Chris Newman. *SQLite (Developers Library)*. Sams, USA, 2004.

[13] M. Poremba, T. Zhang, and Y. Xie. Nvmain 2.0: A user-friendly memory simulator to model (non-)volatile memory systems. *IEEE Computer Architecture Letters*, 14(2):140–143, July 2015.

[14] Boris Schäling. *The boost C++ libraries*. Boris Schäling, 2011.