

MemoryIslands: A Federated Approach for Efficient Memory Mappings

Fatemeh Derakhshani* and Mohamed Hassan* †

*McMaster University, Canada

†American Univeristy in Sharjah, UAE

Abstract—Modern computing systems increasingly feature diverse co-running workloads with varying memory access patterns and requirements. However, existing main memory architectures employ rigid, application-agnostic memory mapping strategies, leaving significant performance potential untapped. This paper introduces MemoryIslands, a novel methodology for treating main memory as a federated collection of independent regions, or “islands,” each tailored to the unique memory demands of individual applications. Our contributions include: (1) a profiling-based methodology to identify optimal address mappings for diverse workloads and (2) a software-aware hardware co-design approach that configures memory controllers to leverage these islands without requiring changes to the software-hardware interface. Evaluation across over 80 workloads on single- and multi-core systems demonstrates significant performance improvements—up to 50%—compared to state-of-the-art static mapping techniques. By enabling application-specific memory mappings and partitioning, MemoryIslands provides a scalable and efficient solution to address the limitations of existing memory architectures.

I. INTRODUCTION

Modern CPUs and GPUs have advanced rapidly in computational power, but memory subsystems lag in bandwidth and latency, creating the well-known “memory wall” that limits system performance. Efficient memory allocation is therefore critical. Modern memory systems are organized hierarchically in the form of channels, ranks, bank groups, banks, rows, and columns. As a result, the address mapping dictated by memory controllers (MCs) to allocate and map memory address space strongly affect application performance. Three key observations motivate this paper.

Observation 1: The memory performance of an application depends heavily on its particular memory access pattern and how it is mapped into the aforementioned main memory segments. *Observation 2:* Modern Multiple Processors Systems-on-Chip (MPSoCs) execute multiple applications that exercise diverse memory access patterns. Additionally, in heterogeneous MPSoCs (like those with both GPUs and CPUs), it has been shown that typical GPU and CPU workloads exhibit different behavior that mandates different memory mappings [1]. *Observation 3:* State-of-the-art MCs deploy application-agnostic static techniques that are universally applied to all incoming requests. These techniques typically favor a very particular access pattern, for example, these exhibiting high locality. Nonetheless, even if they are tailored to favor another access pattern, they remain focused only on one pattern at the expense of hurting applications not exhibiting such

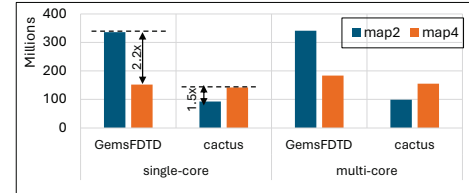


Fig. 1: Application’s performance with two different mappings. map2:RoCoBaRaCh, map4:RoBaRaChCo.

a pattern. Taking address mapping as an example, current MCs deploy one address mapping for all applications. However, we observe that the application’s performance significantly varies with different mappings. Figure 1 shows this observation for two of the SPEC2006 benchmarks with two different mappings. While GemsFDTD seems to have high locality and benefit from having the Dynamic Random Access Memory (DRAM) column bits (CO) at the least significant bits (map4), cactus exhibits better performance when the mapping is interleaving-oriented and maps channel (Ch), rank (Ra), and bank (Ba) bits to the lower address bits (map2). As the figure illustrates, there is a big performance variation based on the selected address mapping (*Observation 1*), and the two applications exhibit better performance using different mappings (*Observation 2*). Finally, in a multi-core SoC, where the two applications execute together, the MC has to choose only one mapping. As Figure 1 also shows, deploying either mapping in this case, one of the two applications suffers significant performance degradation (*Observation 3*).

Several solutions have utilized dynamic address schemes [2] and [3] at run time; however, they suffer from costly data migrations. Instead of relying on dynamic remapping and its costly data migration, this work rethinks off-chip memory as a federation of islands, each with its own address mapping. This perspective *treats memory as a heterogeneous resource that can be tuned to application needs*. Our contributions are as follows:

1) **A methodology to decide address mapping.** We propose two techniques: (i) a stream-aware method that leverages well-defined patterns (e.g., $A[i]$, $A[B[i]]$) to place high-toggle bits where they maximize parallelism and low-toggle bits where they improve locality, and (ii) a lightweight profile-based method that executes workloads under a small set of candidate mappings and directly selects the best. Profiling provides reliable guidance and avoids the mismatch of indirect metrics

such as bit-flip ratios [4] or machine learning predictors [5]. Across more than 80 benchmarks, we find that only a handful of simple mappings consistently capture most of the benefits, making this approach practical.

2) **Federated memory abstraction.** We propose to view main memory as independent *islands*, each composed of one or more banks. Different islands can use distinct mappings, page policies, or scheduling strategies, all without modifications to commodity DRAM devices. This abstraction reduces interference between co-running applications and exposes more parallelism opportunities.

3) **Lightweight software–hardware co-design.** We configure the memory controller with island definitions and mappings using two minor modifications: (i) a scratchpad Static Random Access Memory (SRAM) that holds island and mapping configurations, and (ii) a reverse Translation Lookaside Buffer (rTLB) that links physical requests back to virtual addresses for island identification. Crucially, this requires no changes to the Instruction Set Architecture (ISA), programming model, or Operating System (OS), which makes adoption straightforward.

Our cycle-accurate simulations on 1–8 core systems show substantial gains, with performance improvements of up to 50% over the best static mapping, demonstrating the promise of federating memory into application-aware islands.

II. BACKGROUND

DRAM Architecture. A key element in the DRAM hierarchy is the row buffer, which serves as a temporary local buffer for accessing rows in a bank. When a memory access request is issued, the associated row is activated and loaded into the row buffer, enabling subsequent column accesses to proceed without additional row activation. This process, termed a *row buffer hit*, minimizes latency and power consumption. Conversely, accessing a different row necessitates a precharge and row activation cycle, increasing access time and energy consumption. Consequently, maximizing the row buffer hit ratio is critical for DRAM performance optimization.

DRAM Address Mapping. The MC maps physical memory addresses to DRAM hierarchy components such as channels, ranks, bank groups, banks, rows, and columns. An efficient address mapping scheme balances locality (to maximize row buffer hits) and parallelism (to utilize multiple banks concurrently). Poor mapping can lead to frequent row conflicts and reduced parallelism, significantly degrading performance. Previous approaches have predominantly relied on static address mapping schemes optimized for specific workloads. However, such fixed mappings are ill-suited for heterogeneous workloads, where diverse memory access patterns lead to conflicting requirements. As illustrated in Figure 1, the performance of an application can vary significantly depending on the chosen mapping, highlighting the need for application-specific strategies.

Streams and Strides. Many applications exhibit repetitive memory access patterns known as *streams*, characterized by sequential or strided access. A *stride* represents the constant address difference between consecutive accesses. For instance,

memory-intensive applications like matrix multiplication and image processing exhibit regular strided patterns, while other workloads such as database queries or linked list traversal may feature irregular, non-strided patterns. Understanding the distinction between strided and non-strided applications is crucial for optimizing address mappings. Strided patterns can benefit from mappings that enhance row locality or interleave across banks for parallelism. In contrast, non-strided patterns often require profiling-based methods to identify optimal mappings due to their irregularity.

Challenges in Memory Allocation. Heterogeneous computing systems exacerbate the challenges of memory allocation. Concurrent workloads compete for limited memory bandwidth, resulting in contention and performance degradation. Traditional scheduling techniques, while effective for mitigating contention, often ignore the impact of address mapping and lack fine-grained control over memory resources. To address these challenges, this work proposes **MemoryIslands**, a methodology that reimagines main memory as a federated collection of independent regions. By tailoring memory mappings and resource allocations to application-specific requirements, **MemoryIslands** aims to optimize performance while minimizing interference across workloads.

III. MOTIVATION

This section explores the key challenges in memory mapping and allocation, motivating our proposed methodology.

A. Application Diversity and Mapping Sensitivity

The performance of an application is strongly influenced by its memory access patterns and the corresponding address mapping. Figure 2 illustrates the diversity of memory access patterns through heatmaps of bit-flip occurrences for non-strided and strided benchmarks. Non-strided workloads, such as SPEC2006 benchmarks (Figure 2a), exhibit irregular and unpredictable patterns, necessitating profiling-based approaches to identify optimal mappings. In contrast, strided benchmarks (e.g., Linpack, Figure 2b) show regular patterns, allowing tailored mappings to leverage stride-based regularities. This diversity underscores a significant challenge: current memory controllers rely on static, application-agnostic mapping strategies, which fail to exploit workload-specific characteristics. A universal mapping approach is inherently insufficient to address the unique needs of all applications, necessitating application-aware strategies to improve performance.

B. Impact of Address Mapping on Performance

To quantify the effects of address mapping, we evaluated four predefined mappings across various benchmarks: map1 (ROW-CH-RNK-BNK-COL), map2 (ROW-COL-BNK-RNK-CH), map3 (ROW-CH-BNK-RNK-COL), and map4 (ROW-BNK-RNK-CH-COL), where ROW, CH, RNK, BNK, COL are the row, channel, rank, bank, and column bits, respectively. Results are shown in Figure 3. Mapping-sensitive applications, such as Sphinx3, Leslie3d, and GemsFDTD,

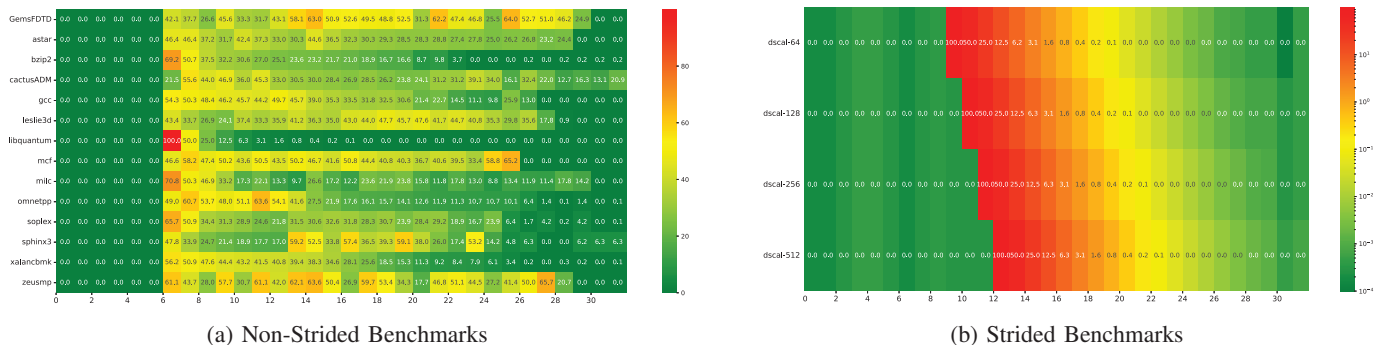


Fig. 2: Heatmap of Bit-flip Patterns for Strided and Non-strided Benchmarks.

exhibited execution time variations of up to 45.57%, 43.77%, and 38.36%, respectively, depending on the mapping used. Conversely, mapping-insensitive workloads, such as GCC and Libquantum, were relatively unaffected. These results lead to two key observations: 1) Different applications benefit from different mappings, with no single strategy performing optimally across all workloads. 2) Mapping-sensitive applications can achieve substantial performance improvements through tailored mappings.

C. Challenges in Multi-Core Systems

In multi-core systems, co-running workloads compete for shared memory resources, leading to contention and performance degradation. Existing solutions, such as scheduling policies [6], [7], aim to mitigate interference by prioritizing requests or partitioning memory channels based on workload intensity and row-buffer hit rates. However, these approaches face two significant limitations: 1) The finite number of memory channels limits scalability in high-core systems. 2) These methods often require non-trivial hardware or scheduling modifications. To address these challenges, we propose partitioning memory at a finer granularity, specifically at the bank level. This approach, which isolates workloads into independent *memory islands*, reduces interference while enabling static, application-aware mappings for improved performance.

From these observations, we identify three critical insights that motivate our proposed methodology: **1) Mapping Sensitivity:** Performance varies significantly across applications, necessitating tailored address mappings. **2) Partitioning for Interference Reduction:** Fine-grained memory partitioning can isolate workloads and reduce contention in multi-core systems. **3) Scalable, Static Solutions:** Static, pre-configured mappings eliminate the overhead of dynamic adjustments while ensuring scalability. Our proposed **MemoryIslands** methodology leverages these insights to transform memory into federated regions, each customized to the specific requirements of individual applications. This approach provides a robust and scalable solution for optimizing modern heterogeneous systems.

IV. METHODOLOGY

The presented methodology revolves around an innovative approach to memory allocation, focusing on a static method

that customizes memory mapping for individual applications and assigns a tailored partition to each application. This approach starts with the application analysis. This stage involves a detailed examination to identify key features influencing optimal memory mapping, using the stride size of the strided applications. Static profiling is employed when stride size is not evident, ensuring personalized mapping for each application’s needs. By *profiling* we refer to the analysis of application characteristics during compile time as well as running it with various mappings to study its memory performance. Figure 4 illustrates a higher-level design of our proposed solution.

Our approach begins by conducting a thorough analysis of the application to pinpoint features that significantly influence the selection of optimal memory mapping. These identified features play a crucial role in tailoring the memory allocation strategy for improved performance. One key feature that emerged from various experiments is the stride size of the main (memory-intensive) stream. This critical feature, representing the distance between consecutive elements accessed in the main data stream, directly influences the efficiency of memory access patterns. Building upon this insight, our approach introduces a set of mapping schemes corresponding to each stream within an application. This static mapping approach allows for personalized mapping for individual streams, enhancing adaptability to diverse characteristics within a single application. The main idea is to assign the channel bits to the address bits starting from the bit corresponding to the stride of the stream as it toggles the most as noted in Figure 2(b). The subsequent higher order bits after that are assigned to ranks, then groups, then banks. For example, for a system with 2 channels, 2 ranks per channel each of which has 4 bank groups and each group encompassing 4 banks, in Figure 2(b) for dscal-256, bit 11 will be assigned to the channel, bit 12 is for rank,..etc. This ensures these toggles cause maximum parallelism. Another potential mapping is to assign the hot bits to columns to maximize locality. One key takeaway from understanding this pattern of the stream is to assign the address bits that do not toggle at all to the lower row bits (which is usually avoided by commodity address mappings). This will increase row hits, and hence improve performance. In case of the previous dscal-256 example, these will be bits 6 to 10.

For applications where the stride size is not readily apparent or the program is not necessarily strided, static profiling is

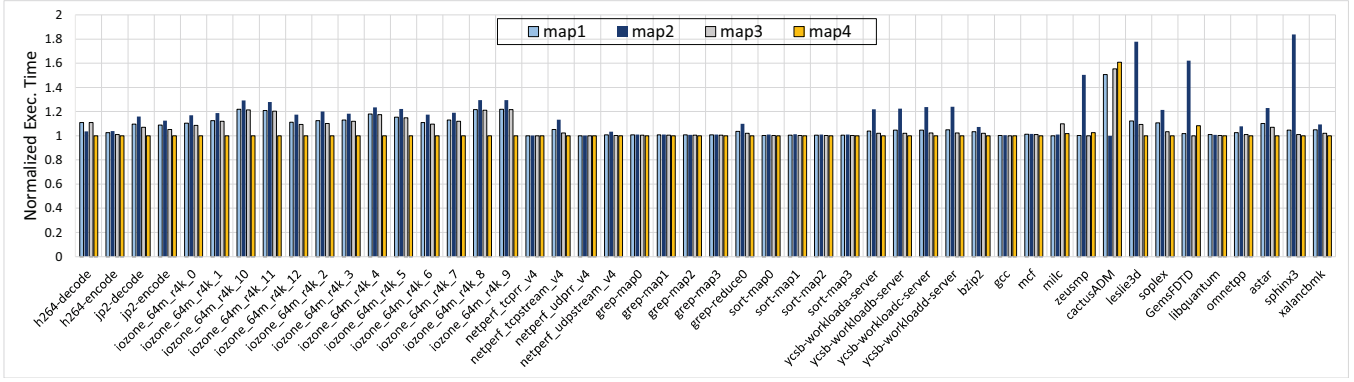


Fig. 3: Comparison of Execution Time for Single-Core Systems Over Non-stream-based Applications.

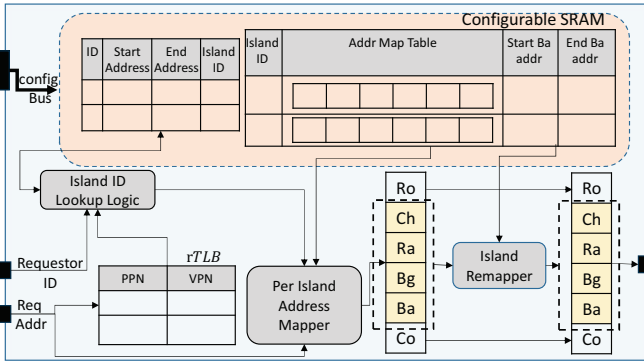


Fig. 4: A high-level design of our island-based approach

applied to the entire or a part of the program. This involves experimenting with a pool of potential base mappings to identify the best option. Incorporating partitioning (where as aforementioned, each partition would be called an island) guarantees that different applications or streams do not interfere with each other. Figure 4 illustrates the architecture of our proposed solution, a pivotal component integrated within the memory controller. Preceding the processing of any request, our system relies on a critical resource known as the TLB inverse (rTLB) table. This table serves as a gateway to retrieve the virtual address. The virtual address is indispensable for accessing the island ID, as all the data stored in the configurable SRAM table during compile time is intricately linked to the virtual address.

Each of the main streams and applications possesses a specific identifier, which is stored within a configurable SRAM table (denoted as the left table of Configurable SRAM in Figure 4). For non-strided applications, this ID corresponds to the thread ID, whereas for strided applications, it comprises a combination of the thread ID and the stream’s ID. Additionally, in the case of strided applications, the start and end addresses of the streams are retained to facilitate stream differentiation. Our objective is to allocate each of these applications and streams to distinct memory islands each assigned a unique island ID. The mappings obtained from the application analysis, along with its Island ID, are stored in another table, which is denoted as Address Map Table in

Figure 4. Upon receiving a request, the Island ID Lookup Logic utilizes the rTLB and the left table of Configurable SRAM (in Figure 4) to extract the corresponding island ID. This step uniquely requires the virtual address. Following this crucial stage and the subsequent extraction of the island ID, our operations will exclusively rely on the physical address. With the island ID determined, we employ the right table of Configurable SRAM (in Figure 4) to identify the appropriate mapping for the given address and retrieve its corresponding mapping using the Island Remapper logic.

A. Illustrative Example

Let’s consider a scenario where we receive a request from the stream “A” with a physical address of 0x24C6A40E43F8 and the corresponding virtual address retrieved from the rTLB as 0x7FFD819DE008. Let’s say in Figure 4, stream “A” is stored with the island ID “5” and starts from address 0x7FFD819DD010 to 0x7FFD819DE048. Using the “Island ID Lookup Logic” and then “Per Island Address Mapper” and its corresponding table in Configurable SRAM, we find the suitable mapping, which, for this example, let’s say is “ROW-COL-BNK-RNK-CH”.

Considering we have 15 bits for rows, 6 bits for columns, 2 bits for banks, 2 bits for bank groups, 1 bit for rank, 1 bit for channel, and 6 bits for the cache line size, the channel would be 1, the rank would be 1, the bank group would be 3, the bank would be 0, the column would be 36, and the row would be 10499. Next step is the fine-grained partitioning (bank-level partitioning). This partitioning is crucial for dividing the entire memory into small, distinct partitions (islands), with one bank serving as the smallest possible unit in our case. Each application or stream then claims one or more partitions based on its specific requirements, thereby preventing other applications or streams from using these partitions. The number of banks required for each application is determined statically based on the analysis of the application patterns and requirements. We assign corresponding banks across different channels, then ranks, and then bank groups, facilitating efficient parallel processing. The Island Remapper using the Addr Map Table computes the modulo of dividing the generated bank number by the corresponding island size and adds the result to the start bank number. The resulting new bank number is then

rearranged to its original order based on its original mapping. Finally, in this step, we reattach the row and column, resulting in the final location of the request.

Continuing our example, using the Island Remapper, we would compute the bank number, which in our example would be 0b001111 or 15. Assume that the start and end bank numbers for island 5, stored in the SRAM table, are 7 to 18. Using the Island Remapper, we would compute the new (remapped) bank number as $7 + (15\% \cdot (18 - 7 + 1))$ or 10. Then, we would extract the original channel, rank, bank group, and bank from this new bank number; row and column remain unchanged. According to this, the channel would change to 0, rank would stay 1, bank group would change to 2, and bank would stay 0. Row would remain 10499 and column would remain 36. This is the new location that our approach maps the request to, which is more application-aware and leads to more optimized memory mapping.

V. EVALUATION

This section evaluates our memory mapping methodology. We used Ramulator, a standalone memory simulator, for all evaluations. The base memory configuration in our evaluation is composed of a 4Gb 2400MHz DDR4 memory comprising 2 channels, 2 ranks, 4 bank groups, and 4 banks per bank group. The cache line size is 64B. To generate traces for different benchmarks, we employed an Intel Pin-based tool integrated with Ramulator. Experiments were conducted across both single-core and multi-core scenarios using five benchmark suites: SPEC, MemBen, PolyBench [8], Rodinia [9], and Linpack [10]. The evaluation is divided into two parts: non-stream-based applications and stream-based applications. The benchmarks and their combinations are detailed in Tables I and II, respectively.

A. Profile-based approach

This section evaluates the impact of static profiling and memory mapping techniques on non-stream-based applications. Figure 5 shows results for 2, 4, and 8-core systems. Each application was assigned a dedicated island with its best mapping from profiling, and compared against four base mappings: `map1`, `map2`, `map3`, and `map4` (Section III).

The results demonstrate clear improvements: average gains of 8.3% for 2-core, 10.5% for 4-core, and 0.51% for 8-core setups, with peak improvements of 23.82%, 19.61%, and 5.81%, respectively. These findings highlight the effectiveness of profiling-based mapping for optimizing performance.

B. Stream-aware approach

We next evaluate the stream-aware approach on both single-core and multi-core systems.

1) *Single-core systems*: For stream-based applications, we used benchmarks from PolyBench, Rodinia (Hotspot, Lud, Needle, Srad), and Linpack (Daxpy, DDot, Dscal, Scusumkbn), configured with varying stride sizes (Figure 6). The stride size indicates loop configurations before considering data type byte size. We compared three setups: base

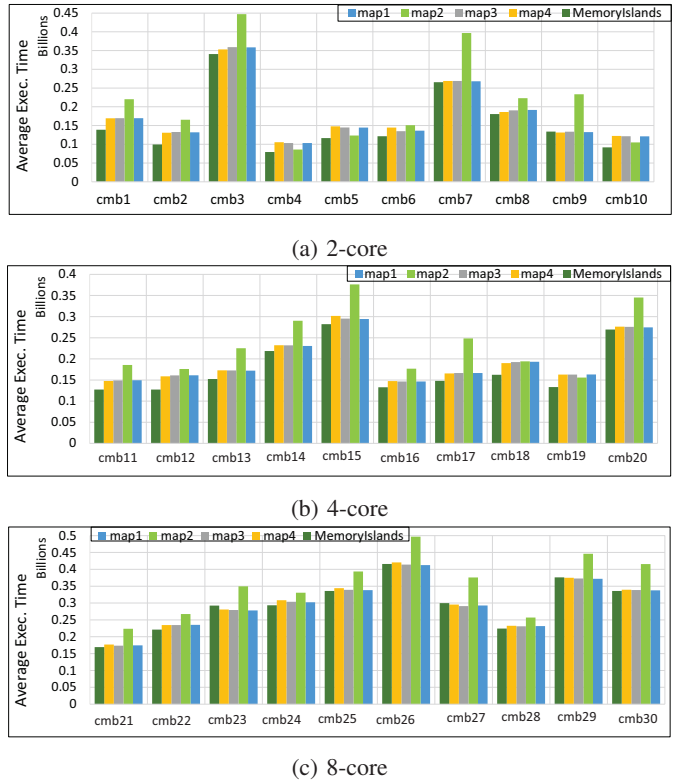


Fig. 5: Effect of suitable mapping on execution time for multi-core systems (non-stream-based applications)

mappings without partitioning, the best mappings identified via stride analysis, and fair partitioning with those mappings.

Results show that identifying the best mapping per stream yields an average improvement of 10.66%, with gains up to 50.55% compared to base mappings.

2) *Multi-core systems*: We extended the analysis to four-core systems (Figure 7), using random subsets of the strided benchmarks. Comparisons were made between base mappings (`map2`, `map4`) and the best mappings identified through stride analysis with fair partitioning. The results reveal average improvements of 21.19%, with maximum gains of 27.56% compared to base mappings. This confirms that the stream-aware methodology significantly improves performance in multi-core environments. Overall, the evaluation demonstrates that MemoryIslands effectively improves performance for both non-stream and stream-based workloads by enabling application-aware mappings and partitioning, without requiring dynamic migration or intrusive system modifications.

VI. RELATED WORK

Address Mapping. A wide range of techniques have been proposed to optimize DRAM address mapping. Early works such as [4] and [11] used permutation- and XOR-based schemes to enhance bank utilization, but relied only on trace features. Dynamic schemes [2] and [3] adapt mappings at runtime but require costly data migration. Entropy-based methods (e.g., [1], [12]) provide indirect metrics and are mostly GPU-focused. [13] models mapping as a graph-cut problem,

TABLE I: Benchmark names and corresponding IDs

ID	Benchmark	Suite	ID	Benchmark	Suite	ID	Benchmark	Suite	ID	Benchmark	Suite
p1	2mm-256	PolyBench	p17	lu-256	PolyBench	h1	grep-map0	Hadoop	y1	ycsb-workloada-server	YCSB + Redis
p2	adi-256	PolyBench	p18	mvt-2048	PolyBench	h2	grep-map1	Hadoop	y2	ycsb-workloadb-server	YCSB + Redis
p3	atax-2048	PolyBench	p19	nussinov-512	PolyBench	h3	grep-map2	Hadoop	y3	ycsb-workloadc-server	YCSB + Redis
p4	bicg-2048	PolyBench	p20	symm-128	PolyBench	h4	grep-map3	Hadoop	y4	ycsb-workloadd-server	YCSB + Redis
p5	cholesky-512	PolyBench	p21	syrk-256	PolyBench	h5	grep-reduce0	Hadoop	s1	astar	SPEC2006
p6	correlation-256	PolyBench	11	daxpy-128	Linpack	h6	sort-map0	Hadoop	s2	cactusADM	SPEC2006
p7	covariance-256	PolyBench	12	ddot-128	Linpack	h7	sort-map2	Hadoop	s3	gcc	SPEC2006
p8	deriche-512	PolyBench	13	dscal-128	Linpack	i1	iozone_64m_r4k_0	IOzone	s4	GemsFDTD	SPEC2006
p9	doitgen-64	PolyBench	14	scusumkbn-128	Linpack	i2	iozone_64m_r4k_6	IOzone	s5	leslie3d	SPEC2006
p10	fdtd-256	PolyBench	r1	lud-512	Rodinia	i3	iozone_64m_r4k_8	IOzone	s6	libquantum	SPEC2006
p11	floyd-256	PolyBench	r2	needle-1024	Rodinia	i4	iozone_64m_r4k_9	IOzone	s7	mcf	SPEC2006
p12	gemver-512	PolyBench	r3	srad-1024	Rodinia	i5	iozone_64m_r4k_10	IOzone	s8	milc	SPEC2006
p13	gesummv-1024	PolyBench	n1	netperf_tcprr_v4	Netperf	i6	iozone_64m_r4k_11	IOzone	s9	omnetpp	SPEC2006
p14	gramshmidt-256	PolyBench	n2	netperf_tcpstream_v4	Netperf	i7	iozone_64m_r4k_12	IOzone	s10	soplex	SPEC2006
p15	heat3d-64	PolyBench	n3	netperf_udpr_v4	Netperf				s11	sphinx3	SPEC2006
p16	jacobi-2d-512	PolyBench	n4	netperf_udpstream_v4	Netperf				s12	zeusmp	SPEC2006

TABLE II: Benchmark ID of combinations

Cmb	Exp	BMs	Cmb	Exp	BMs	Cmb	Exp	BMs	Cmb	Exp	BMs
cmb1	Fig 5(a)	s4, s2	cmb2	Fig 5(a)	s11, s2	cmb3	Fig 5(a)	y2, s6	cmb4	Fig 5(a)	i3, s2
cmb5	Fig 5(a)	n2, s2	cmb6	Fig 5(a)	i2, s12	cmb7	Fig 5(a)	y1, s11	cmb8	Fig 5(a)	s10, s6
cmb9	Fig 5(a)	s12, s4	cmb10	Fig 5(a)	s1, s2	cmb11	Fig 5(b)	s1, s6, s12, s11	cmb12	Fig 5(b)	s2, s6, s12, s11
cmb13	Fig 5(b)	s2, s12, s4, s11	cmb14	Fig 5(b)	h5, y4, s2, s11	cmb15	Fig 5(b)	i2, y1, s6, s4	cmb16	Fig 5(b)	i4, s2, s12, s4
cmb17	Fig 5(b)	s3, s6, s12, s11	cmb18	Fig 5(b)	s10, s2, s6, s12	cmb19	Fig 5(b)	n2, s2, s6, s12	cmb20	Fig 5(b)	y3, s2, s12, s4
cmb21	Fig 5(c)	s1, h5, i3, i4, s6, s12, s4, s11	cmb22	Fig 5(c)	s1, h5, s10, s2, s6, s12, s4, s11	cmb23	Fig 5(c)	s1, i3, i4, n2, y2, s12, s4, s11	cmb24	Fig 5(c)	i1, i2, i3, n2, y3, s2, s6, s12
cmb25	Fig 5(c)	s1, i2, y3, s2, s6, s12, s4, s11	cmb26	Fig 5(c)	i2, y2, y4, s2, s6, s12, s4, s11	cmb27	Fig 5(c)	i3, i4, s5, y4, s2, s12, s4, s11	cmb28	Fig 5(c)	i3, i4, s10, n2, s2, s6, s12, s4
cmb29	Fig 5(c)	i3, s10, y1, y2, s2, s12, s4, s11	cmb30	Fig 5(c)	s5, n2, y2, s2, s6, s12, s4, s11	cmb67	Fig 7	p13, p15, r1, p19	cmb68	Fig 7	p4, i1, i4, r3
cmb69	Fig 7	p3, p18, p20, p21	cmb70	Fig 7	p2, p5, p6, p9	cmb71	Fig 7	i2, p14, p17, r2	cmb72	Fig 7	p3, p6, p18, p19
cmb73	Fig 7	p3, p8, p14, r2	cmb74	Fig 7	i3, p11, p10, p16	cmb75	Fig 7	p7, p12, p14, p20	cmb76	Fig 7	p1, p5, i2, p19

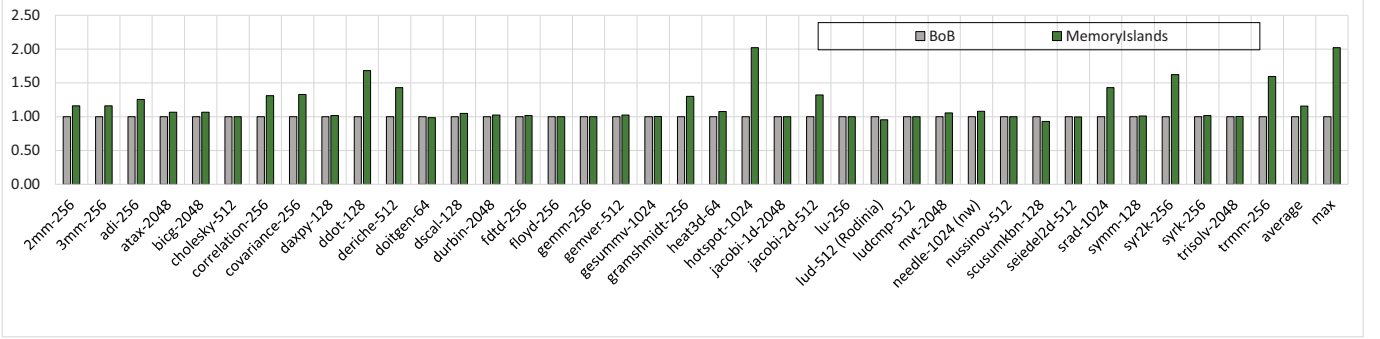


Fig. 6: Execution time for single-core systems for stream-based applications, normalized to Best of Baseline mapping (BoB).

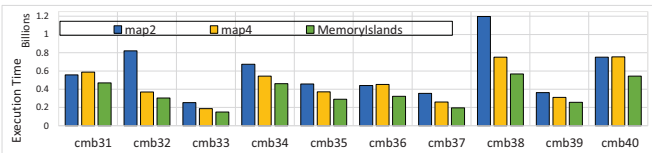


Fig. 7: Stream-based applications in a 4-core setup

while [5] applies reinforcement learning, but both retain a single mapping and lack support for interference reduction. More recently, [14] proposed programmer-defined mappings via *malloc* and OS extensions. While flexible, it is tailored to 3D memories, requires intrusive software/OS modifications, and depends on bit-flip metrics that fail to capture DRAM timing constraints. In contrast, our approach supports multiple mappings concurrently, applies to commodity DRAM hierarchies, and avoids software interface changes.

Memory Partitioning. Partitioning has been explored at banks [15]–[18], ranks [19], [20], and channels [7] to mitigate interference, provide predictability, or block timing channels. Examples include [7], [15] for inter-core isolation, [17] for real-time guarantees, and [18] for side-channel protection. However, most approaches statically allocate resources.

Parallelism Metrics. Several studies estimated memory-level parallelism (MLP) [21], [22] or bank-level parallelism (BLP) [23], [24], and more recently [25] introduced bank parallelism utilization. These metrics are effective for runtime tracking but remain indirect indicators of performance. Since our approach is offline, we instead rely on profiling to directly capture the impact of mappings and allocations, yielding more accurate optimization.

VII. CONCLUSION

MemoryIslands is a methodology that treats DRAM as federated regions with tailored mappings and allocations to different workloads. By combining profiling-based and stream-aware mapping selection with bank-level partitioning formulated as an optimization problem, our approach reduces interference and improves performance without requiring intrusive hardware or software modifications. Evaluation across diverse benchmarks shows significant execution time gains (up to 50%) over state-of-the-art static mapping. *MemoryIslands* thus offers a scalable, application-aware framework for addressing the memory wall in modern heterogeneous systems.

REFERENCES

- [1] Y. Liu, X. Zhao, M. Jahre, Z. Wang, X. Wang, Y. Luo, and L. Eeckhout, "Get out of the valley: Power-efficient address mapping for gpus," in *ACM Annual International Symposium on Computer Architecture (ISCA)*. IEEE Press, 2018. [Online]. Available: <https://doi.org/10.1109/ISCA.2018.00024>
- [2] M. Ghasempour, A. Jaleel, J. D. Garside, and M. Luján, "Dream: Dynamic re-arrangement of address mapping to improve the performance of drams," in *International Symposium on Memory Systems (MEMSYS)*, 2016.
- [3] S. Adavally and K. Kavi, "Towards application-specific address mapping for emerging memory devices," in *International Symposium on Memory Systems (MEMSYS)*. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: <https://doi.org/10.1145/3422575.3422785>
- [4] Z. Zhang, Z. Zhu, and X. Zhang, "A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2000.
- [5] X. Li, Z. Yuan, Y. Guan, G. Sun, T. Zhang, R. Wei, and D. Niu, "Flatfish: A reinforcement learning approach for application-aware address mapping," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2022.
- [6] Y. Kim, D. Han, O. Mutlu, and M. Harchol-Balter, "Atlas: A scalable and high-performance scheduling algorithm for multiple memory controllers," in *International Symposium on High-Performance Computer Architecture (HPCA)*, 2010.
- [7] S. P. Muralidhara, L. Subramanian, O. Mutlu, M. Kandemir, and T. Moscibroda, "Reducing memory interference in multicore systems via application-aware memory channel partitioning," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2011.
- [8] L.-N. Pouchet *et al.*, "Polybench: The polyhedral benchmark suite," <http://www.cs.ucla.edu/pouchet/software/polybench>, vol. 437, pp. 1–1, 2012.
- [9] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *IEEE International Symposium on Workload Characterization (IISWC)*, 2009.
- [10] A. Petitet, R. Whaley, J. Dongarra, and A. Cleary, "Hpl 2.0 – a portable implementation of the high-performance linpack benchmark for distributed-memory computers," 2008.
- [11] N. Chatterjee, M. O'Connor, G. H. Loh, N. Jayasena, and R. Balasubramonia, "Managing dram latency divergence in irregular GPGPU applications," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2014.
- [12] W. Chen, Z. Li, L. Liu, and S. Wei, "Dynamically reconfigurable memory address mapping for general-purpose graphics processing unit," in *IEEE International Conference on Integrated Circuits, Technologies and Applications (ICTA)*, 2022.
- [13] M. Jung, D. M. Mathew, C. Weis, N. Wehn, I. Heinrich, M. V. Natale, and S. O. Krumke, "Congen: An application specific dram memory controller generator," *International Symposium on Memory Systems (MEMSYS)*, 2016.
- [14] J. Zhang, M. Swift, and J. J. Li, "Software-defined address mapping: A case on 3d memory," in *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2022.
- [15] H. Yun, R. Mancuso, Z.-P. Wu, and R. Pellizzoni, "PALLOC: Dram bank-aware memory allocator for performance isolation on multicore platforms," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2014.
- [16] M. K. Jeong, D. H. Yoon, D. Sunwoo, M. Sullivan, I. Lee, and M. Erez, "Balancing DRAM locality and parallelism in shared memory cmp systems," in *IEEE International symposium on high-performance comp architecture (HPCA)*, 2012.
- [17] J. Reineke, I. Liu, H. D. Patel, S. Kim, and E. A. Lee, "PRET DRAM controller: Bank privatization for predictability and temporal isolation," in *IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis (CODES+ISSS)*, 2011.
- [18] Y. Wang, A. Ferraiuolo, and G. E. Suh, "Timing channel protection for a shared memory controller," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2014.
- [19] J. H. Ahn, N. P. Jouppi, C. Kozyrakis, J. Leverich, and R. S. Schreiber, "Future scaling of processor-memory interfaces," in *The Conference on High Performance Computing Networking, Storage and Analysis (SC)*, 2009.
- [20] H. Zheng, J. Lin, Z. Zhang, E. Gorbato, H. David, and Z. Zhu, "Mini-rank: Adaptive DRAM architecture for improving memory power efficiency," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2008.
- [21] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt, "A case for MLP-aware cache replacement," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 2, pp. 167–178, 2006.
- [22] J. Tuck, L. Ceze, and J. Torrellas, "Scalable cache miss handling for high memory-level parallelism," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2006.
- [23] C. J. Lee, V. Narasiman, O. Mutlu, and Y. N. Patt, "Improving memory bank-level parallelism in the presence of prefetching," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2009, pp. 327–336.
- [24] X. Tang, M. Kandemir, P. Yedlapalli, and J. Kotra, "Improving bank-level parallelism for irregular applications," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–12.
- [25] S. Ghose, T. Li, N. Hajinazar, D. S. Cali, and O. Mutlu, "Demystifying complex workload-dram interactions: An experimental study," *ACM on Measurement and Analysis of Computing Systems (POMACS)*, 2019.