

HSPref: Efficient Software Prefetching for ARM SME Outer-Products

<p>Han Huang CSE School Sun Yat-sen University Guangzhou, China huangh367@mail2.sysu.edu.cn 0009-0009-3774-9115</p>	<p>Lanshu Huang CSE School Sun Yat-sen University Guangzhou, China huanglsh25@mail2.sysu.edu.cn 0009-0006-5673-0493</p>	<p>Xianjie Chen CSE School Sun Yat-sen University Guangzhou, China chenxj275@mail2.sysu.edu.cn 0009-0006-0485-0127</p>	<p>Xianwei Zhang CSE School Sun Yat-sen University Guangzhou, China zhangxw79@mail.sysu.edu.cn 0000-0003-3507-4299</p>
---	---	--	--

Yutong Lu*
CSE School
Sun Yat-sen University
Guangzhou, China
luyutong@mail.sysu.edu.cn
0000-0001-5315-3375

Abstract—The Scalable Matrix Extension (SME) in modern Arm processors delivers very high throughput for tensor computation. However, SME’s outer-product execution model generates structured, multi-dimensional memory access patterns that are poorly handled by existing hardware and software prefetchers, leading to significant memory bottlenecks. As a result, developers often rely on software prefetching techniques. Nonetheless, existing software prefetching approaches are not well-suited for SME computation. We investigate the challenges of enabling efficient software prefetching for SME computation. Our findings show that the performance of SME software prefetching is limited by inaccurate target selection, suboptimal prefetch distances and poor alignment with the cache hierarchy.

We present **HSPref**, a software prefetching framework that systematically bridges SME computation and cache behavior. **HSPref** comprises four key design modules: (1) a computation-pattern analysis that identifies data access patterns and suitable prefetch targets; (2) an analytical model that determines optimal prefetch distances by jointly considering instruction pipelines and memory latency; (3) a hardware-aware instruction-injection strategy that maps prefetches to appropriate cache levels; and (4) an assembly-level pass with lightweight APIs, enabling practical integration into real-world applications. We evaluate **HSPref** on LX2 and Apple M4 platforms. Results show that **HSPref** achieves average speedups of $1.34\times$ and $1.38\times$, respectively, while reducing MPKI by 23% on average and up to 51%.

Index Terms—Arm, Scalable Matrix Extension, Software Prefetch, Outer-Product Computation

I. INTRODUCTION

In recent years, Arm-based processors have seen rapid adoption across cloud and edge platforms, spanning hyperscale datacenters, high-performance computing (HPC) systems, and consumer devices. To meet the increasing demand for computational performance, Arm introduced the Scalable Matrix Extension (SME) in new generation Arm processors [1], which provides native support for outer-product-based matrix

operations. SME has already been integrated into processors from major vendors, including LX2, Apple, and Samsung [2]. SME exposes outer-product matrix instructions that deliver high-throughput tensor computation, conceptually similar to GPU tensor cores. Specifically, an SME outer-product unit consumes two input vectors, computes their outer-products, and then accumulates the result into dedicated on-chip matrix registers. By maximizing data reuse within the register file, such execution model significantly improves compute efficiency for matrix-intensive workloads.

While featuring high compute capabilities, overall performance of SME is often constrained by the memory subsystem. Our analysis shows that, in SME-based workloads, more than 80% of cycles are stalled due to memory accesses (Figure 14). This phenomenon indicates that SME effectively alleviates compute bottlenecks while shifting the performance critical path to memory system. Prefetching is a well-established technique to mitigate such stalls by bringing data into the cache ahead of use, thereby helping overlap memory accesses with computation. However, existing hardware prefetchers in Arm processors are primarily designed for simple access patterns such as sequential or strided streams [3]. These mechanisms are ineffective for SME workloads, where outer-product execution generates complex and multi-dimensional access patterns. As a result, hardware prefetchers frequently fail to provide timely and accurate data, leading to high cache miss rates and substantial stall cycles.

In practice, software prefetching is instead employed as a complementary approach. Unlike hardware mechanism, software prefetchers can adapt to dynamic program behavior without increasing microarchitectural complexity, making it a natural alternative to hardware prefetchers. Accordingly, vendors (e.g. Intel [4] and Apple [3]) often recommend that developers either insert application-specific prefetch instructions or rely on compiler-inserted prefetching. Prior works

*Yutong Lu is the corresponding author.

on software prefetching have primarily focused on predictable streaming and strided accesses, as well as indirect memory patterns such as linked-list traversals [5]–[7]. For prefetching in SME computation, existing works are mainly tailored to memory optimization in specific computation patterns [8] rather than providing a general prefetching framework. Existing compiler-based approaches for Arm SME (e.g., Apple Clang and LX2 Clang) are not well suited to SME outer-product kernels for three main reasons: (1) they lack fine-grained knowledge of tile-level and outer-product access patterns, leading to incorrect prefetch targets; (2) they do not model prefetch distance in a way that properly balances compute-memory overlap for SME’s high-throughput inner loops, thus yielding imprecise prefetch distances; and (3) they insert prefetch instructions without sufficient awareness of the cache hierarchy or instruction-level parallelism (ILP), often resulting in suboptimal scheduling and limited performance gains.

To address these challenges, we present HSPref, a software prefetching mechanism tailored for SME outer-product matrix multiplication. HSPref bridges the gap between SME computation and memory behavior through four key parts: (i) a computation-pattern analysis that identifies the precise targets that need to be prefetched for outer-product kernels; (ii) an analytical prefetch-distance model that jointly accounts for compute latency and memory latency; (iii) architecture-aware prefetch placement and instruction-injection strategies that select appropriate cache levels and schedule prefetches to preserve instruction-level parallelism; and (iv) an assembly-level pass with lightweight APIs, easing the difficulties of practical integration into existing applications. Our evaluation on real Arm SME platforms, including Apple M4 and LX2, demonstrates that HSPref achieves substantial performance improvements and significantly reduces cache miss rates.

This paper makes the following key contributions:

- 1) We identify that SME outer-product kernels generate a multi-dimensional memory access pattern that is not effectively handled by existing hardware prefetchers or vendor compilers. We further characterize how the dominant memory bottleneck varies with computation mode and data size, which motivates the SME-oriented prefetching design.
- 2) We present HSPref, a cross-layer prefetching method that (i) determines appropriate prefetch targets, (ii) computes prefetch distance via DAG-based critical-path analysis that jointly models compute and memory latency, (iii) selects cache levels and instruction placement in accordance with the SME engine’s cache topology.
- 3) We implement HSPref as an assembly-level pass with lightweight APIs and evaluate it on LX2 and Apple M4 using representative benchmarks. HSPref achieves a geometric mean speedup of $1.34\times$ / $1.38\times$ on both platforms (up to $2.50\times$), reducing MPKI by 23% on average, and lowers stall rates from 83.4% to 71%, consistently outperforming existing prefetch designs.

II. BACKGROUND

This section introduces the microarchitectural features of two representative Arm SME platforms and describes the outer-product computation model.

A. Microarchitecture of the CPUs with Arm SME

The latest generation of Arm CPUs introduce the SME units to provide matrix computing power. We study two commercially deployed Arm processors with native SME support: the Apple M4 [9] and the LX2 (CPU of the new exascale LineShine supercomputer deployed at NSCC-SZ in China).

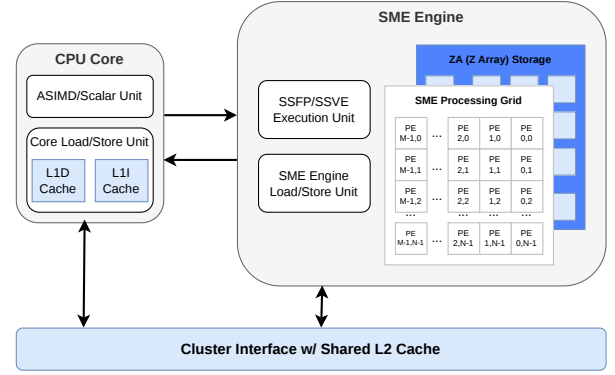


Fig. 1: Microarchitecture of the Apple M4 CPU [3], where SME attaches directly to the L2 cache.

As illustrated in Figure 1, the Apple M4 integrates an SME engine as a dedicated accelerator outside the general-purpose cores. Each CPU cluster includes one SME engine that executes both the Streaming SVE (SSVE) [10] and SME instructions. For the cache hierarchy, the 128 KiB private L1 caches are tightly coupled to the CPU cores and serve the scalar and Advanced SIMD (ASIMD) units, while the SME engine interfaces directly with the shared 16 MB L2 cache. The LX2 platform adopts a different topology, as shown in Figure 2. Each core integrates its own SME engine, which can access the private 32 KiB L1 data cache. Each core is also connected to the shared L2 cache (768 KiB per core). Both platforms equip the SME engine with a streaming hardware prefetcher for simple strided and nested access patterns [11].

Algorithm 1: Outer-Product Computation

Input: vector a, b , matrix C

Output: matrix C

```

1 for  $point \in C$  do
2    $C[i][j] = a[i] \times b[j] + C[i][j]$ 

```

B. Outer-Product Computation in SME

SME units achieve high matrix throughput and excessive computing power through outer-product operations. In the operation (see Algorithm 1), three operands are involved: (i)

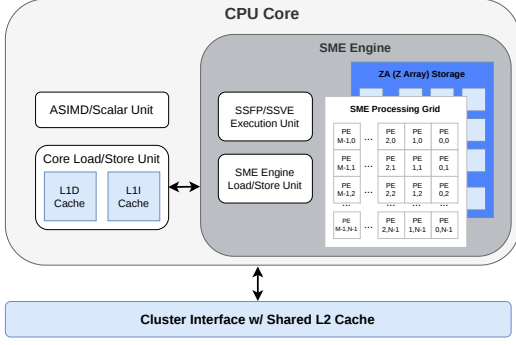


Fig. 2: Microarchitecture of the LX2 CPU [8], where SME attaches to the core L1 cache.

a left input vector $\mathbf{a}$; (ii) a right input vector $\mathbf{b}$; and (iii) an accumulation matrix $\mathbf{C}$. Each operation performs a rank-1 update, $\mathbf{C} += \mathbf{a}\mathbf{b}^\top$, and in FP64 mode it produces an 8×8 tile. Real world computations mainly utilize the rank- k update [12], [13], which repeatedly performs the rank-1 update for k iterations.

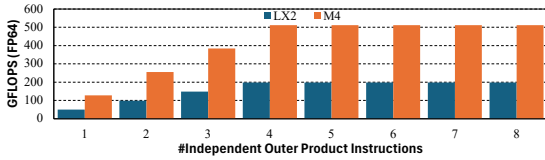


Fig. 3: Outer-product ILP study on LX2 and M4.

Achieving peak throughput for outer-product computation requires sufficient instruction-level parallelism. To quantify this behavior, we conduct an ILP study, with results being shown in Figure 3. The results indicate that at least four independent outer-product operations are needed to reach peak performance, and issuing more than four outer-product operations maintains the throughput. Since the Arm SME architecture has up to eight ZA registers in a single SME unit in FP64 [14], developers usually utilize all of the eight registers to guarantee independent instructions and maximal performance.

While exploiting all eight ZA registers maximizes instruction-level parallelism and compute throughput, it simultaneously introduces a structured, multi-dimensional memory access pattern that existing prefetch mechanisms cannot effectively handle. We characterize this pattern in detail and analyze its implications for prefetch design in Section III.

III. DEMYSTIFYING THE MEMORY ACCESS PATTERNS

Building on the SME outer-product patterns, we now characterize these patterns in depth to demystify how and why the prefetchers underperform on Arm SME. First, we conduct a theoretical analysis of data volumes and access structures across computation modes (Section III-A). Then we perform microarchitectural profiling that quantifies how cache behavior

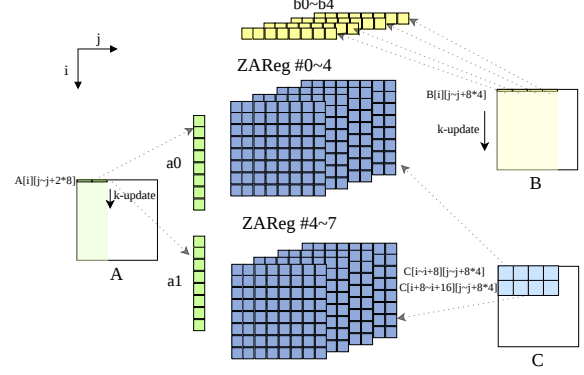


Fig. 4: Register organization in a 2×4 layout.

and backend stalls evolve with data size (Section III-B). Building on these analyses, we further demonstrate why existing hardware and software prefetch approaches fall short for each identified bottleneck (Section III-C).

A. Theoretical Analysis

Different outer-product computation patterns exhibit distinct memory access behaviors, which in turn impose different prefetch requirements. Based on whether each input operand is loaded from memory or held in registers, for outer-product computations, we first identify two representative patterns (see Table I): **data-data**, where both input vectors are streamed from memory and thus both require continuous prefetching; and **coeff-data**, where one operand is a fixed coefficient kept in registers, while only the other operand is streamed from memory. We also note a third case, **coeff-coeff**, in which both input operands are kept in registers. However, this case rarely appears in hotspot kernels, and its memory access behavior is typically well handled by hardware prefetcher. We therefore focus on the more common and performance-critical **data-data** and **coeff-data** patterns.

data-data. In the **data-data** pattern, the computation uses the 2×4 ZA layout (Figure 4) to maximize the compute-to-load ratio. In each iteration, the kernel loads two $\mathbf{A}$ vectors and four $\mathbf{B}$ vectors (each of length SVL_B bytes), performs a rank-1 outer-product update, and accumulates into eight ZA tiles. To be noted, $\mathbf{A}$ is in transposed form in common **data-data** patterns for contiguous memory access. For a rank- k update, the total data volumes are:

$$\begin{aligned} \text{size}(\mathbf{A}) &= 2 \times k \times SVL_B \\ \text{size}(\mathbf{B}) &= 4 \times k \times SVL_B \\ \text{size}(\mathbf{C}) &= 8 \times \frac{SVL_B}{8} \times SVL_B \end{aligned} \quad (1)$$

where SVL_B is the streaming vector length in bytes (at most 64B on LX2 and M4 [3]). The $\mathbf{C}$ term follows from the fact that eight FP64 ZA tiles each occupy $(SVL_B/8) \times (SVL_B)$

TABLE I: Memory access patterns of two outer-product computation modes, classified by the loaded source of the operands.

Outer-Product Pattern	Code Example	Memory Access Pattern	Representative Benchmarks
<i>data-data</i>	<pre> load(C) for (int k=0; k<K; k++) a = load(A[k, i→i+2*8]) b = load(B[k, j→j+4*8]) ZARegs += outer_product(a,b) C[i→i+2*8, j→j+4*8] = store(ZARegs) </pre>		<i>gemm, symm</i>
<i>coeff-data</i>	<pre> load(C) #optional for (int k=0; k<K; k++) a[0→7] = {0,0,1,2,3,2,1,0} b = load(B[k, j→j+8*8]) ZARegs += outer_product(a,b) C[i→i+8, j→j+8*8] = store(ZARegs) </pre>		<i>stencil, conv</i>

double-precision elements, giving a combined storage of SVL_B^2 bytes.¹

coeff-data. In the *coeff-data* pattern, one operand provides fixed coefficients that are assigned statically in each iteration, while the other is streamed from memory. Taking **A** as the coefficient operand and **B** as the streaming operand, the computation maps to the 1×8 layout. A key distinction from *data-data* is that **A** is not loaded from the memory, and therefore it does not constitute a continuous prefetch target. Only **B**, which is loaded in every iteration of $8 \times (SVL_B)$ data, requires continuous prefetching. The data volumes for the inner loop are:

$$\begin{aligned}
 \text{size}(\mathbf{B}) &= 8 \times k \times SVL_B \\
 \text{size}(\mathbf{C}) &= 8 \times \frac{SVL_B}{8} \times SVL_B
 \end{aligned} \quad (2)$$

In practice k is often small for this pattern (e.g., stencil radius or convolution filter size), so the one-time **C** load/store is not negligible relative to the total **B** traffic. Both operands must be treated as prefetch targets in the kernel.

B. Perf-based Microarchitectural Analysis

In addition to the theoretical characterization, we use `perf` [15] to quantify how cache behavior and backend stalls vary with matrix sizes. This analysis identifies the dominant memory bottlenecks in both smaller and larger regimes. For *data-data*, we use *gemm* as the representative benchmark, since it is the typical large- k outer-product workload. For *coeff-data*, we use *jacobi-2d* as the representation, since it is a well-studied stencil kernel with a small radius (k).

Figure 5 compares cache behavior across input sizes. We use 1024×1024 as a small case and 4096×4096 and 8192×8192 as large cases. *gemm* exhibits a moderate hit rate of 60% to 70% under small cases, but falls to 1% to 5% as the working set becomes large. However, *jacobi-2d* exhibits relatively stable L1 cache hit rates, while its L2 hit rate declines gradually. This divergence is consistent with our theoretical analysis in Equation 2. The *data-data* pattern generates substantially more

loads per iteration, making it far more sensitive to working-set growth.

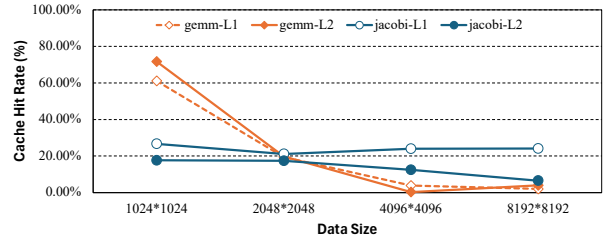


Fig. 5: Cache hit rate for *gemm* and *jacobi-2d* across data sizes. *Gemm*'s L1 hit rate collapses from 60% to below 5% at large sizes, while *jacobi-2d* retains more stable L1 behavior.

Bottleneck analysis. Figure 6 shows the backend stall breakdown for *gemm*. Across all data sizes, stalls attributed to loading and storing **C** account for less than 1% of the total, confirming that the accumulation tile is entirely resident in ZA storage and contributes negligible memory pressure in the large- k case. The bottleneck is dominated by the input operands, with **B** contributing over 90% of stalls at small data sizes and approximately 55% at large data sizes, while **A** rises to roughly 40%. The mechanism behind this shift is structural: in the 2×4 layout, **B** generates four concurrent loads while **A** generates two, so at moderate working-set sizes the **B** vectors are more likely to saturate prefetcher capacity and produce a disproportionate share of misses. As the working set grows well beyond cache capacity, both operands approach a 100% miss rate. Therefore, both **A** and **B** should be loaded from the memory, producing the more balanced 55%/40% split observed at 8192^2 .

Figure 7 shows the corresponding breakdown for *jacobi-2d*. At small data sizes, stores to **C** account for up to 84% of stalls, while loads from **B** contribute only about 10%. In this regime, loads of **B** can be handled by the hardware prefetcher, whereas **C** stores require cache-line allocation and write-back that hardware prefetchers do not cover, leaving them as the bottleneck. As data size increases and the working set exceeds

¹For $SVL = 512$ bit ($SVL_B = 64$ B): each FP64 tile is 8×8 doubles (512 B).

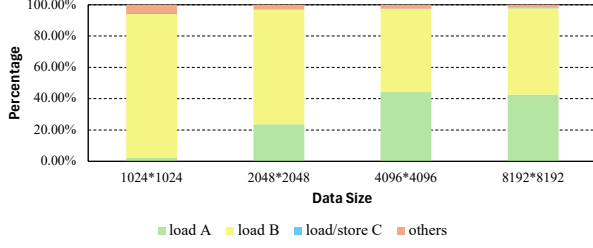


Fig. 6: Backend stall-cycle breakdown for *gemm*. **B** dominates stalls at small sizes ($>90\%$) and **A** rises to $\sim 40\%$ at large sizes. **C** contributes $<1\%$ throughout.

cache capacity, this balance inverts. **B** incurs continuous cache misses whose volume outpaces what the prefetcher can absorb, rising to 73% of stalls, while **C** stores stabilize around 20%.

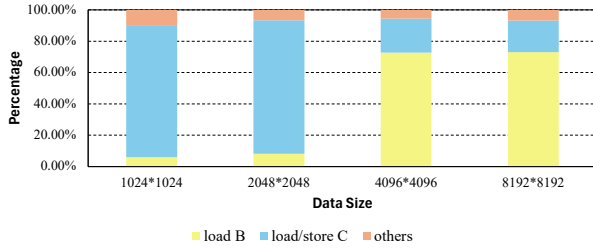


Fig. 7: Backend stall-cycle breakdown for *jacobi-2d*. The dominant bottleneck shifts from **C** stores (84% at 1024^2) to **B** loads (73% at 8192^2) as the working set exceeds cache capacity.

C. Limitations of Existing Prefetch Approaches

The theoretical and empirical analyses above establish that outer-product workloads generate multi-dimensional memory access patterns whose dominant bottlenecks vary significantly across computation modes and data-size regimes. We now examine why existing hardware and software prefetch mechanisms are inadequate for these patterns to motivate the design of HSPref.

1) *Hardware prefetchers are ill-suited to large-strided outer-products.*: The SME engine operates in streaming mode and employs a dedicated streaming prefetcher [11], allowing it to avoid being stalled by activities in the NEON or scalar cores. However, the SME prefetcher only supports relatively simple strided and nested access patterns. As shown in Section III-B, the hardware prefetcher provides useful coverage only when the working set is small. At 1024^2 , L1 hit rates remain at 60 to 70%, while at 4096^2 and beyond they collapse below 10% for *data-data* workloads, and fall substantially for *coeff-data* workloads (see Figure 5). The root cause is structural. In the k -update loop, vectors **a** and **b** are loaded from the source matrices at strides of M and N respectively. As these strides grow with matrix size, the SME prefetcher is less capable of tracking more general patterns [3], [11] and

fails to track the resulting multi-load and large-stride structure. The hardware mechanism is therefore insufficient for the SME outer-products.

2) *Software prefetching faces three compounding difficulties.*: Software prefetch instructions (PRFM in Arm) are the most immediately deployable alternative, and prior work [8], [16] demonstrates their effectiveness for specific SME stencil kernels. However, applying them correctly to general outer-product kernels introduces three challenges that existing tools do not adequately address.

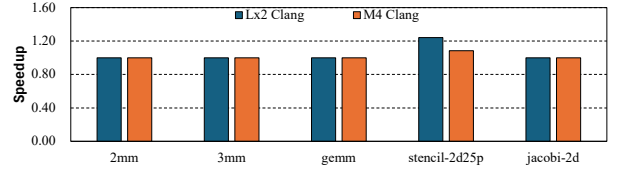


Fig. 8: Speedup of compiler-automatic prefetch. The compiler emits useful PRFM instructions only for *stencil-2d25p*, and fails on all remaining benchmarks.

Prefetch target identification depends on computation pattern and data-size regime. Section III-A establishes that the set of memory streams requiring prefetching is not uniform. For *data-data* kernels, only **A** and **B** require continuous prefetching: as confirmed in Figure 6, **C** resides in ZA storage and contributes negligible memory pressure across all data sizes. For *coeff-data* kernels, however, Figure 7 shows that the dominant bottleneck shifts from **C** boundary traffic at small sizes to **B** streaming loads at large sizes, requiring both operands to be treated as prefetch targets with size-dependent priorities. This computation-dependent structure is beyond the reach of current compiler heuristics. We evaluate both the LX2 Clang [17] on LX2 and Apple Clang [18] on M4 (see Figure 8). Across our benchmark suite, both compilers can identify prefetch opportunities in the *stencil-2d25p* kernel, but fail to emit any useful PRFM instructions for the remaining benchmarks.

TABLE II: Sensitivity of *gemm* performance to prefetch distance on LX2.

Prefetch	IPC Speedup	Prefetch Coverage	Execution Time Speedup
None	1.0×	0%	1.0×
Distance = 1	1.42×	91.6%	0.84×
Distance = 32	2.06×	93.8%	1.20×
Distance = 256	0.91×	61.7%	0.67×

Prefetch distance must jointly account for compute and memory latency. Even when the correct targets are identified, the prefetch must arrive in cache at the right time. Table II illustrates this sensitivity on the *gemm* benchmark: a distance of 32 successfully covers 93.8% of demand loads and achieves a speedup of $1.20\times$, whereas a distance of 1 causes prefetches to arrive too late and thus stall the pipeline,

and a distance of 256 instead over-prefetches and evicts still-useful data, inflating execution time to $0.67\times$. Existing distance models [5]–[7], [19] have been designed for indirect memory access workloads, where the loop body consists primarily of pointer-chasing loads and iteration latency, which is well approximated by the L1 cache access latency. This simplification is not suitable for SME outer-product kernels, which contain high-latency FMOPA instructions executing on the SME engine pipeline in parallel with load/store operations. Thus, SME outer-product kernels require a distance model that jointly accounts for pipeline depth and memory latency.

```

1      ldld { z1.d }, p0/z, [x30, x2, lsl #3]
2      prfm pldl1keep, [x12, #64]
3      ldld { z2.d }, p0/z, [x30, x4, lsl #3]
4      prfm pldl1keep, [x12, #128]
5      ldld { z3.d }, p0/z, [x30, x5, lsl #3]
6      prfm pldl1keep, [x12, #192]
7      ldld { z4.d }, p0/z, [x30, x6, lsl #3]
8      prfm pldl1keep, [x12, #256]

```

Listing 1: Prefetch instructions generated by the Apple Clang compiler [18], incorrectly targeting L1 on the M4 platform.

TABLE III: Performance impact of the mis-targeted L1 prefetch.

Data Size	Prefetch Speedup
4096 ²	1.07×
6144 ²	0.88×
8192 ²	0.85×

Prefetch cache level must match the platform’s SME topology. As established in Section II, the SME engine’s cache attachment point differs across platforms. On LX2 it connects directly to the private 32 KiB L1 cache, while on Apple M4 it connects to the shared 16 MB L2 [3]. Issuing prefetches to the inappropriate level either pollutes useful cache capacity unnecessarily or fails to stage data close enough to the execution unit. Despite this, Apple Clang emits PRFM instructions targeting L1 (see Listing 1). Since the M4 SME engine does not benefit from L1 residency, these miss-targeted prefetches can degrade rather than improve performance at certain data sizes (see Table III). To this end, a correct and portable prefetch solution must treat cache topology awareness as a first-class design factor.

IV. METHOD

The analyses in Section III expose three concrete requirements for effective prefetching in SME outer-product kernels: prefetch distance, prefetch cache level and prefetch targets. In this section, we address each requirement in turn, presenting the analytical models and assembly pass that together constitute HSPref (see Figure 9).

A. SME-Aware Prefetch Distance Model

The first step is to determine how far ahead each prefetch should be issued. As established in Section III, the prefetch distance must be large enough to fully hide memory latency,

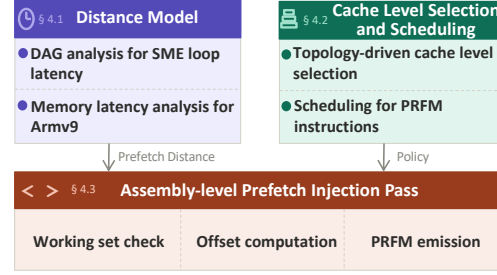


Fig. 9: Overview of the HSPref Design. HSPref consists of an analytical prefetch distance model, a cache-level selection strategy, and an assembly pass that emits prefetch instructions.

TABLE IV: Symbol definitions used in the prefetch distance model.

Notation	Value
$L_{L1\$}$	Access Latency of L1 Cache
L_{Memory}	Access Latency of Memory
L_{iter}	Latency of one iteration
D_{pf}	Prefetch Distance

yet bounded by the loop iteration count to remain effective. Although the Arm CPU with SME is a new CPU platform with matrix units, we observe that the core principle of the prefetch distance is similar. Using the symbols in Table IV, we can formulate the prefetch distance. A prefetch issued D_{pf} iterations ahead must satisfy:

$$D_{pf} \times L_{iter} \geq L_{Memory}, \quad (3)$$

so that D_{pf} iterations of execution are sufficient to cover the round-trip memory latency before the corresponding demand load issues. In the indirect-memory-access workloads, L_{iter} is approximated as the L1 access latency $L_{L1\$}$, reducing the formula to $D_{pf} \approx \lceil L_{Memory}/L_{L1\$} \rceil$.

This simplification is not suitable for Arm SME outer-product kernels, where matrix and vector instructions have higher latency and execute on dedicated pipelines alongside the load/store unit. The true L_{iter} for an SME kernel is substantially larger than $L_{L1\$}$ once compute latency is included. Therefore, we retain the principle of Equation 3 but replace the simplified L_{iter} with an explicit model that accounts for both compute and memory operations:

$$D_{pf} = \left\lceil \frac{L_{Memory}}{L_{iter}} \right\rceil, \quad (4)$$

where L_{iter} is derived from a DAG-based critical-path analysis of the loop body and L_{Memory} captures the effective memory latency under concurrent loads.

1) Iteration Latency via DAG Critical-Path Analysis:

A naive sum of all instruction latencies overestimates L_{iter} because instructions dispatched to different execution units overlap in time. We model one loop iteration as a directed acyclic graph (DAG) $G = (V, E)$, where each node $v_i \in V$ represents one instruction with latency lat_i , and each directed

edge $(v_j, v_i) \in E$ encodes a true (RAW) data dependency. Instructions are assigned to three execution domains:

- **SME/SVE instructions**, executing on the SME engine;
- **Integer, floating-point, and ASIMD instructions**, executing on the CPU core pipeline;
- **Load/store instructions**, handled by the load/store unit (LSU) of the SME engine.

The DAG captures data-dependency and cross-domain pipeline overlap under two simplified assumptions. First, we model each domain as issuing multiple instructions per cycle based on the domain capability. Second, cache-miss penalties are not incorporated into the DAG. Instead, they are captured separately by L_{Memory} to avoid double-counting. Let $\text{pred}(i)$ denote the immediate predecessors of v_i . The start time S_i is:

$$S_i = \begin{cases} 0, & \text{pred}(i) = \emptyset, \\ \max_{j \in \text{pred}(i)} (S_j + \text{lat}_j), & \text{otherwise,} \end{cases} \quad (5)$$

and the iteration latency is the DAG makespan:

$$L_{\text{iter}} = \max_{v_i \in V} (S_i + \text{lat}_i). \quad (6)$$

As shown in Figure 10, SME/SVE, LD/ST, and INT/FP instructions execute on independent pipelines and overlap substantially. Figure 11 illustrates the corresponding DAG, where the load nodes v_1 and v_2 have no predecessors and execute immediately, feeding into the `fmul` nodes v_3 and v_4 , which in turn feed the `fropa` node v_5 . The critical path $v_1 \rightarrow v_3 \rightarrow v_5 \rightarrow v_6$ determines L_{iter} .

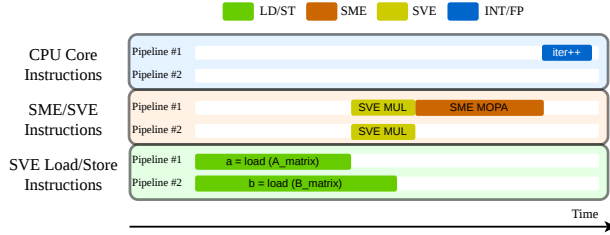


Fig. 10: Instruction-level parallelism across execution domains in outer-product kernels.

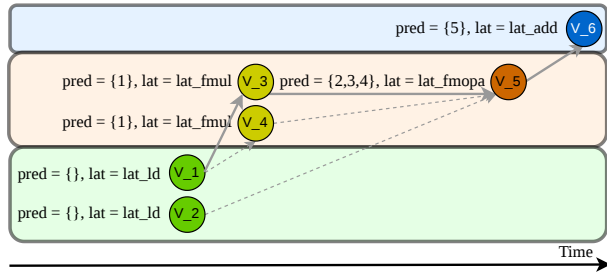


Fig. 11: DAG for one inner-loop iteration. The critical path, which determines L_{iter} , is highlighted in bold.

To validate our model, Table V compares the DAG-predicted L_{iter} against cycles-per-instruction (CPI) measured via `perf stat` across the representative kernels. On both LX2 and M4, the model prediction falls within 12% of the measured value for every kernel, confirming that the single-issue-per-domain approximation is adequate for the purpose of distance estimation.

TABLE V: DAG-predicted L_{iter} vs. measured CPI (cycles).

Kernel	LX2		M4	
	Predicted	Measured	Predicted	Measured
GEMM	1.38	1.54	0.96	1.02
JACOBI-2D	0.84	0.87	1.06	1.11

2) Effective Memory Latency under Concurrent Vectors:

To obtain the latencies, we measure raw cache and DRAM latencies via pointer-chasing microbenchmarks [20]. We use a working set of 512 MB to ensure accesses are served from DRAM. The DDR latency is respectively 260 and 380 cycles on LX2 and M4.

We define c as the hardware's capability to handle multiple outstanding DRAM requests in parallel. We measure it by issuing an increasing number of independent sequential vectors, and record the aggregated memory latency and throughput. Throughput scales linearly until c vectors are active, and we identify the knee of the curve as the c value. When a kernel issues N_s concurrent prefetch demands, the hardware can service at most c in parallel. We model the excess as serialized in batches of c , giving:

$$L_{\text{Memory}} = \left\lceil \frac{N_s}{c} \right\rceil \times L_{\text{DDR}}. \quad (7)$$

The distance study in Figure 19a and Figure 19b confirms that the formula's predictions can bring better performance on both platforms. Substituting Equation 7 into Equation 4 gives the final distance formula:

$$D_{\text{pf}} = \left\lceil \frac{[N_s/c] \times L_{\text{DDR}}}{L_{\text{iter}}} \right\rceil, \quad (8)$$

where N_s is determined by the register organization and computation mode, L_{DDR} and c are platform-specific constants measured as above, and L_{iter} is computed from the DAG analysis of the kernel's inner loop.

B. Topology-Driven Cache Level Selection and PRFM Scheduling

Having determined the prefetch distance D_{pf} , the remaining design questions are *where* in the cache hierarchy to stage the data, and the scheduling in the instruction stream to issue each PRFM.

1) *Topology-Driven Cache Level Selection:* For selection, the guiding principle is simple: prefetch to the cache level that the SME engine reads from, not the level closest to the CPU core. On LX2, each core's SME engine is directly coupled to its private 32 KiB L1D. Consequently, prefetching **a** and **b** into L1D minimizes the load-to-use latency for the SME engine.

To verify that prefetching does not pollute L1D, we compute the worst-case resident volume. At the prefetch distance of 16, the 2×4 organization brings in $16 \times 6 \times 64 \text{ B} = 6 \text{ KiB}$ of prefetched data simultaneously, occupying less than 19% of L1D capacity and leaving ample space for the accumulation tile and control data.

On **Apple M4**, the SME engine is decoupled from the per-core L1D and connects directly to the cluster-shared 16 MB L2 cache [3]. Consistent with the topology, we target L2 (PLDL2STRM) for all SME operand prefetches, reserving L1D exclusively for the integer, floating-point, and NEON pipelines that execute alongside the SME engine. At the same D_{pf} , the 6 KiB resident prefetch footprint occupies less than 0.04% of the 16 MB L2, confirming that pollution is negligible at this level.

Algorithm 2: HSPref PRFM placement within one rank-k update iteration.

```

1 for  $k \in \{kStart, \dots, kEnd\}$  do
    /* Step 1: Load  $B[k]$  and prefetch  $B[k+D_{pf}]$  */
2   load  $B_k$ 
3   prefetch  $B_{k+D_{pf}}$ 
4   (SME/SVE Computation)
    /* Step 2: Prefetch  $C$  */
5   prefetch  $C$ 
6   (Remaining SME/SVE Computation)

```

2) *PRFM Scheduling*: Even with the correct distance and cache level, the placement of PRFM instructions within the loop body affects the overall performance. Since we prefetch the data of the $(k + D_{pf})$ th iteration, compute instructions of the k th iteration do not have dependencies with the PRFM instructions. We exploit this observation to derive the scheduling policy described below and formalized in Algorithm 2.

Load prefetches. For each streaming operand, we emit the corresponding PRFM immediately after the current-iteration load of the same operand but before the subsequent SME/SVE computation instructions. Doing this has two advantages. First, positioning PRFM directly after the load allows it to reuse the already-computed base address, eliminating redundant address calculation overhead. It also exposes strided access patterns, which are friendly to the memory and cache. Second, the PRFM consumes only the load/store unit pipeline, which is separate from the SME and SVE compute pipelines. Therefore, it overlaps with the SME/SVE instructions in the same cycle, improving overall performance.

Store prefetches. The accumulation matrix C is written back to memory either at the end of the rank-k update, or at the end of the current iteration if the stores can be pipelined. In the pipelined case, we prefetch the data to be stored in the current iteration. In the non-pipelined case, we prefetch one row of C registers per iteration (up to 8 rows in FP64), beginning at iteration $kEnd - 8$. In both cases, the PRFM instructions are placed between the current-iteration compute

instructions (step 2 in Algorithm 2), allowing them to overlap with in-flight computation and avoid introducing pressure on the critical path.

C. Assembly-Level Prefetch Implementation

The implementation of HSPref combines lightweight user APIs with an assembly-level injection pass. The workflow (see Figure 12) consists of three stages: 1) *Annotation and assembly generation*: the developer marks outer-product loops with HSPref functions, and the source is compiled normally to assembly; 2) *Analysis and decision*: HSPref scans the generated assembly to decide whether to prefetch and to determine the prefetch distance and cache hierarchy; and 3) *Injection*: HSPref emits prefetch instructions at the appropriate locations, after which the standard assembler and linker produce the final binary.

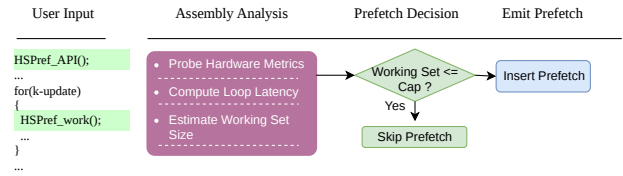


Fig. 12: Overview of the HSPref implementation. Given user-annotated code, HSPref estimates working-set size, computes D_{pf} , and emits PRFM instructions when beneficial.

```

1 mem_handle_t A_mem = m_handle_create(A, M, K, FP64);
2 mem_handle_t B_mem = m_handle_create(B, K, N, FP64);
3 mem_handle_t C_mem = m_handle_create(C, M, N, FP64);
4 for(size_t i = blk_i; i < blk_i + blk_size_i; i+=16)
5 {
6     for(size_t j = blk_j; j < blk_j + blk_size_j; j+=32)
7     {
8         load(C);
9         for(size_t k = blk_k; k < blk_k + blk_size_k; k+=1)
10        {
11            HSPref_work(A_mem, B_mem, C_mem);
12            load(A);
13            load(B);
14            compute();
15        }
16        store(C);
17    }
18 }

```

Listing 2: HSPref API usage on a GEMM kernel.

In the first stage, HSPref exposes a **memory handle** that encapsulates the base address and matrix size. The APIs are used for obtaining the current working set size, and users can invoke `HSPref_work()` for prefetch targets (see Listing 2). After users annotate the source code, the code is compiled into assembly code.

In the second stage, HSPref conducts a lightweight pass over the assembly text to decide whether to prefetch, and to determine prefetch distances and cache hierarchy. Before emitting any prefetch, HSPref derives the aggregate working set directly from the matrix dimensions in the memory handle. As established in Section III-A, at most two matrices

are simultaneously active in both *data-data* and *coeff-data* patterns, so the aggregate reduces to the sum of the two active matrix sizes. Prefetching is enabled only when this aggregate exceeds the **LLC capacity**. On Apple M4, whose LLC is 16 MB, the two active matrices at 1024^2 occupy approximately 16 MB in total and reside within the LLC. Prefetching is therefore suppressed, avoiding the $0.86\times$ slowdown that would otherwise result (see Figure 17b). At 2048^2 , the aggregate matrix volume grows to approximately 64 MB, well beyond LLC capacity. Prefetching is enabled and the speedup rises to $1.15\times$. On LX2, whose LLC per core is 768 KiB, the same criterion suppresses prefetching only when the total active matrix size falls below 768 KiB. Once prefetching is enabled, HSPref reads the probed hardware metrics, and obtains the prefetch cache hierarchy and prefetch distance. HSPref computes L_{iter} via DAG critical-path analysis of the loop body, and evaluates Equation 8 to obtain D_{pf} .

In the final stage, HSPref lowers each function call into PRFM instructions. Each PRFM instruction is placed at the injection site determined by the scheduling policy in Section IV-B2, enabling data to be staged ahead of its corresponding demand access. The instrumented assembly then proceeds through the standard assembler and linker to produce the final binary.

D. Discussion: Optimization Scope

We discuss two aspects of the optimization scope of HSPref: 1) its behavior under multi-core execution, and 2) its effectiveness in real-world applications.

Prefetching in multi-core scenario. The prefetch analysis in this work targets single-core SME computation. On M4, each cluster exposes one SME unit that occupies the full shared L2, so single-core analysis suffices in this case. On LX2, each core has an independent SME unit, and multi-core execution introduces additional challenges. First, concurrent prefetches from multiple cores can evict the live data from shared LLC, which may cause a cache pollution problem. Second, the prefetch traffic can congest the memory bus, increasing the effective L_{DDR} seen by all cores. Third, a prefetched line may be invalidated by a remote write before it is consumed, generating unnecessary coherence traffic. We identify multi-core SME prefetch as future work, and note that our per-core distance and cache-level analysis provides the necessary baseline from which such a cooperative extension can be built.

Generalization in real-world applications. A natural concern is whether HSPref remains effective in real-world applications. First, HSPref only instruments the annotated SME outer-product loops, leaving the surrounding program code entirely unaffected. Second, SME outer-product kernels (e.g., GEMM, stencil, and convolution) constitute the dominant computational hotspots in realistic HPC and AI workloads. Although our model is not specifically tuned for different operations in real-world applications, we observe consistent performance gains in larger and more complex workloads, as shown in Section 5.

V. EVALUATION

A. Experimental Setup

TABLE VI: Platform configurations.

Item	LX2	Apple M4
Cores	38	4 (per cluster)
L1D	32 KiB	128 KiB
L2 (per SME unit)	768 KiB	16 MB
SME attachment	L1D	L2

Platform. We evaluate on two modern Arm SME platforms as listed in Table VI. On LX2, each of the 38 cores contains a private SME unit attached to its private 32 KiB L1D. Each core in a NUMA node has a 768 KiB L2. On Apple M4, four cores share one SME unit that accesses the 16 MB shared L2 cache. L1D serves only the NEON and integer pipelines. On LX2, we use the *perf* tool for profiling and analysis. On M4, we can only use XCode Instruments [21], which is not able to track the L2 cache metrics. Therefore, we can only report the cache metrics on the LX2 CPU.

Compiler and configurations. We use Clang on LX2 and Apple Clang on M4, both with `-O3`. In the evaluation, all the speedups are normalized to the speedup against the no-prefetch version. For comparison, we select the configurations below:

- Clang: Automatic prefetch by the Clang compiler.
- HSPref-c: HSPref with **correct** targets and cache level, the prefetch distance is set to 1.
- HSPref-ct: HSPref-c with the **timely** prefetch distance.
- HSPref-cts: HSPref-ct with the PRFM **scheduling**.

We run each experiment five times, and we report the average speedup.

Benchmarks. Table VII lists 10 SME kernels from PolyBench [22] and 2 kernels from SparStencil [23], all run at FP64 precision.

TABLE VII: Evaluated benchmarks.

Benchmark	Source	Outer-Product Type	
		<i>data-data</i>	<i>coeff-data</i>
2mm, 3mm, doitgen	Polybench [22]	✓	
symm, syrkc, syr2k	Polybench	✓	
trmm, gemm	Polybench	✓	
conv, stencil-2d25p	HStencil [8] and SparStencil [23]		✓
jacobi-2d, heat-3d	Polybench		✓

B. Overall Performance

Figure 13 reports per-benchmark and average speedup over the no-prefetch baseline at 4096×4096 on both LX2 and M4. For *heat-3d*, we use a comparable data size of 256^3 . Clang benefits only *stencil-2d25p* ($1.24\times$ on LX2, $1.08\times$ on M4), where the access pattern happens to match the compiler’s strided-stream heuristic. On all remaining benchmarks,

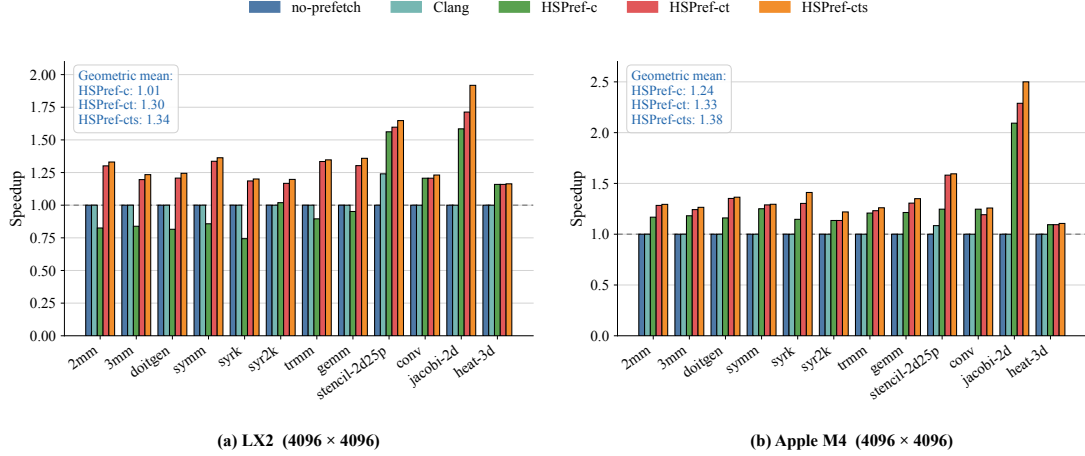


Fig. 13: Per-benchmark and average speedup over the no-prefetch baseline on LX2 and M4 (4096×4096). HSPref achieves geomean speedups of $1.34\times$ and $1.38\times$, consistently outperforming Clang and all ablation variants.

it provides negligible gain or slight regression, consistent with the target-identification failures documented in Section III-C.

For the *data-data* kernels, HSPref-c achieves a geometric mean of $0.86\times$ on LX2 and $1.18\times$ on M4. This indicates that prefetching to the correct operands and cache levels alone is insufficient to bring substantial benefits for *data-data* kernels. HSPref-ct improves to $1.25\times / 1.27\times$. This is because *data-data* kernels sustain long k -loops with both a and b loads being active simultaneously, so the effective memory latency is high and distance sensitivity is pronounced. HSPref-cts reaches **$1.28\times$ on LX2 and $1.31\times$ on M4** with scheduling. The additional gain demonstrates that in compute-bound FMOPA loops, placing PRFM instructions correctly reduces pipeline backpressure. For the *coeff-data* kernels, HSPref-c yields $1.36\times$ on LX2 and $1.37\times$ on M4. Since the *coeff-data* kernels utilize the SVE instructions to concatenate the vectors, the memory accesses are reduced. Thus, for the compute-intensive *coeff-data* kernels, directing prefetches to the correct operands and to the SME-attached cache level is the dominant contributor to performance. HSPref-ct adds distance tuning and reaches $1.40\times / 1.47\times$. This shows that distance tuning can provide even more performance within the *coeff-data* kernels. HSPref-cts reaches **$1.46\times$ on LX2 and $1.53\times$ on M4** with instruction scheduling.

Among the benchmarks, *jacobi-2d* shows the largest gain with $1.92\times / 2.50\times$. This is because loads and stores constitute significant bottlenecks concurrently in *jacobi-2d* (see Section III). HSPref eliminates both bottlenecks by issuing load prefetches for the streaming operand and store hints to hide the store latencies on the accumulation matrix. Also, we observe that *heat-3d* achieves the lowest speedups with $1.16\times / 1.11\times$. This is due to the computation patterns of *heat-3d*, where the three-dimensional arrays incur even more complicated memory access patterns. Even under such circumstances, HSPref still can provide speedups over 10%.

C. Prefetch Efficiency Metrics

We evaluate prefetch efficiency through three metrics: backend stall rate, MPKI, and prefetch coverage, showing that HSPref effectively reduces memory bottlenecks across all benchmarks.

Stall rate. Figure 14 shows backend stall rates of the baseline version without prefetch and HSPref on LX2. The geometric mean stall rate falls from 83.4% to 71%. *stencil-2d25p* shows the largest reduction from 75% to 51%, since HSPref solves the prefetch of both the loads and stores. The reduction of stall rate demonstrates that HSPref can help alleviate the memory bottlenecks.

MPKI. Figure 15 shows MPKI (Miss Per Kilo Instructions) reduction. Most benchmarks achieve 10% to 30% reduction, with a geometric mean of 23%. *3mm* and *doitgen* achieve approximately 50% reduction, since they contain heavy matrix multiplication operations in their computations patterns, and HSPref can help alleviate the memory bottlenecks.

Prefetch coverage. Figure 16 reports the fraction of demand loads successfully prefetched. HSPref achieves a geometric mean coverage of 76%. For the *3mm*, *gemm*, and *stencil-2d25p* benchmarks, the coverage exceeds 90%. Specially, *heat-3d* achieves 34%, which is consistent with its overall performance. Meanwhile, HSPref without scheduling attains a lower average coverage of 73%. Beyond preserving ILP, scheduling matters most when the prefetch distance is small. With only a few iterations of lead time, the exact injection site determines whether a prefetch arrives before the corresponding demand load. By placing each prefetch instruction appropriately, HSPref ensures more requests complete in time.

D. Sensitivity to Data Size and Prefetch Distance

To evaluate the effects of HSPref under different data sizes, we conduct experiments as shown in Figure 17. At 1024^2 , HSPref yields a $1.15\times$ speedup on LX2. On M4, the larger 16 MB L2 keeps more data resident and makes

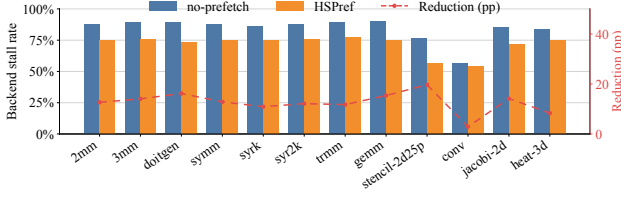


Fig. 14: Backend stall rate without and with HSPref on LX2. The geometric mean falls from 83.4% to 71%.

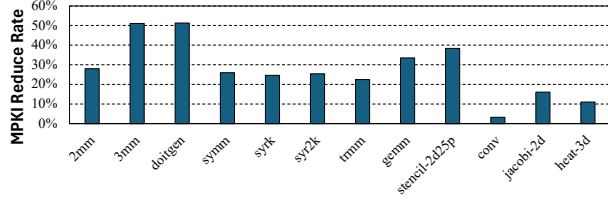


Fig. 15: MPKI reduction rate on LX2. HSPref achieves an average reduction of 23%.

prefetch overhead non-negligible. Thus, HSPref chooses not to emit prefetches for M4 at 1024^2 (Section IV-C). If HSPref emits the prefetches, it will bring a $0.85\times$ speedup. Once the working set exceeds cache capacity at 2048^2 to 4096^2 , speedups rise to $1.30\text{--}1.31\times$ on LX2 and $1.14\text{--}1.38\times$ on M4. At 8192^2 , both platforms sustain these gains ($1.33\times$ and $1.40\times$), confirming that HSPref scales well with working-set growth.

We further study the sensitivity to prefetch distance (Figure 18), by comparing three static distance settings (4, 16, 64) against the prefetch distance selected by HSPref. The LX2 CPU, which has a smaller cache, favors shorter distances, where the speedup drops from $1.18\times$ (dist-4) to $1.09\times$ (dist-16). On the contrary, the M4 CPU, with the larger cache, tolerates longer prefetch distances, achieving similar speedups of $1.25\text{--}1.26\times$ on both dist-4 and dist-16. A distance of 64 leads to over-prefetching on both platforms, resulting in performance of only $0.96\times/1.00\times$. By deriving the distance analytically rather than relying on a fixed value, HSPref matches or exceeds the static choices on each platform ($1.34\times/1.38\times$), confirming the necessity of a platform-aware distance model.

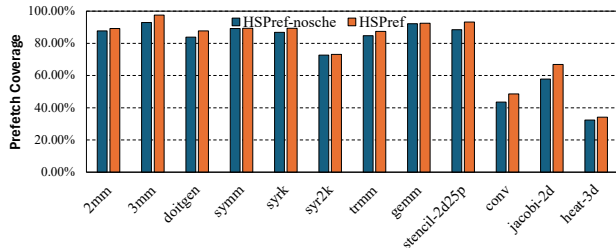
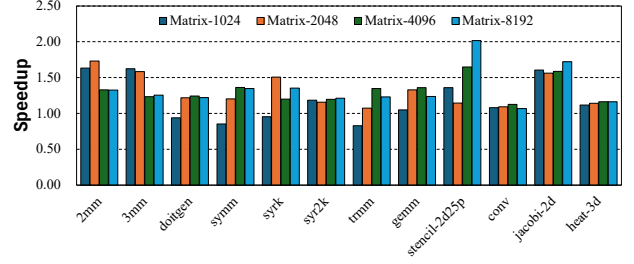
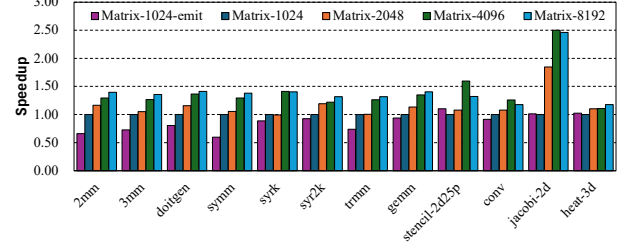


Fig. 16: Fraction of demand loads successfully prefetched on LX2, with and without PRFM scheduling.

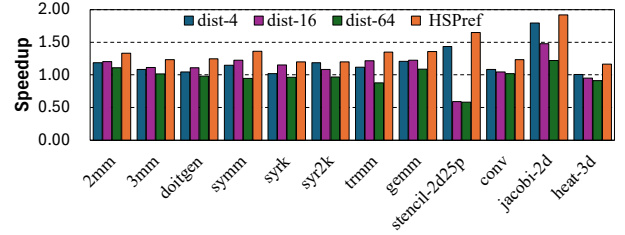


(a) Speedup across matrix sizes (LX2).

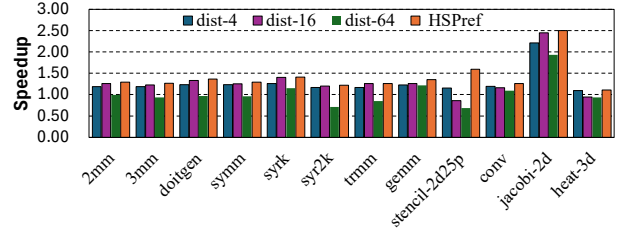


(b) Speedup across matrix sizes (M4).

Fig. 17: Speedup of HSPref across matrix sizes on LX2 and M4.



(a) Sensitivity to prefetch distance (LX2).

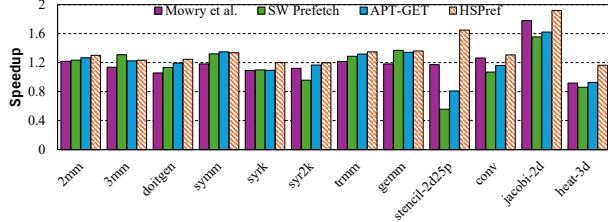


(b) Sensitivity to prefetch distance (M4).

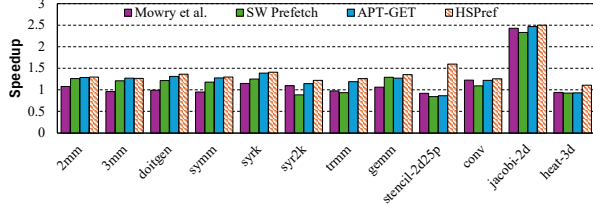
Fig. 18: Sensitivity to prefetch distance on LX2 and M4. "dist- x " sets the static prefetch distance to x .

E. Comparison with Prior Distance Models

To evaluate the effects of the distance model of HSPref, we compare four prefetch distance models, as shown in Figure 19a and Figure 19b. We hold prefetch targets and cache-level selections constant, isolating the contribution of the distance model alone. The distance model of [24] only considers the instructions count of the critical path and the memory latency. Thus, it has a larger distance than HSPref, and



(a) Speedup of prefetch distance models (LX2, 4096×4096).



(b) Speedup of prefetch distance models (M4, 4096×4096).

Fig. 19: Comparison of prefetch distance models on LX2 and M4. HSPref outperforms all the other models, achieving average speedups of $1.34\times$ and $1.38\times$.

achieves a geometric mean speedup of $1.18\times$ on LX2 and $1.10\times$ on M4. SW Prefetch [19] achieves $1.21\times$ on *data-data* benchmarks on LX2 and $1.14\times$ on M4, but degrades on *coeff-data* benchmarks since it only considers the memory latencies, neglecting the computation latencies. APT-GET [6] has a more refined model that considers both the cache access latency and the memory latency, obtaining an average speedup of $1.19\times$ on LX2 and $1.26\times$ on M4. However, since no computation overhead is involved in APT-GET, the model still falls short for the compute-intensive kernels. HSPref achieves an average speedup of $1.34\times$ on LX2 and $1.38\times$ on M4, outperforming all the other SOTA models.

F. Further Explorations

Beyond single-core micro-kernels, we evaluate HSPref generality in two settings it was not specially optimized for: multi-core execution and hand-optimized operators.

Multi-core exploration. Although HSPref targets single-core execution, we evaluate its portability on multiple cores on LX2 with 2, 16, and 32 threads (Figure 20). At 2 threads, HSPref achieves an average speedup of $1.23\times$. At 32 threads, it rises modestly to $1.29\times$. Although HSPref does not contain optimizations specialized for multi-core scenes, we still obtain substantial speedups and identify multi-core SME prefetch optimizations as future work.

Hand-optimized operators exploration. We further evaluate HSPref on a highly optimized FlashAttention operator to demonstrate its portability. Table VIII reports performance of FlashAttention across sequence lengths from 1024 to 3072 on both LX2 and M4. Despite being already hand-tuned, FlashAttention still benefits from HSPref, which consistently improves performance on both platforms, achieving speedups of up to $1.11\times$ on LX2 and $1.07\times$ on M4. Moreover, the

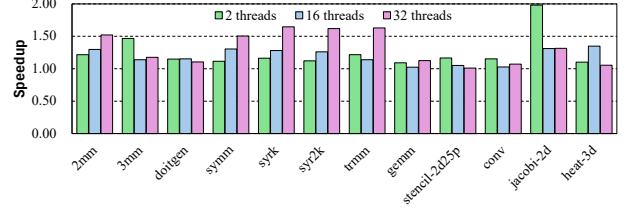


Fig. 20: Multi-core speedups on LX2.

speedup increases with sequence length, as longer sequences expand the attention working set beyond cache capacity, exposing more memory stalls that HSPref can effectively mitigate.

TABLE VIII: Performance of FlashAttention operator with and without HSPref. Performance is measured in GFLOPS (FP32).

Seq. Len.	Perf. w/o HSPref		Perf. w/ HSPref		Speedup	
	LX2	M4	LX2	M4	LX2	M4
1024	118.38	564.03	126.57	577.25	1.07x	1.02x
1536	132.29	708.97	145.07	740.88	1.10x	1.05x
2048	145.85	812.54	158.50	852.69	1.09x	1.05x
2560	148.33	862.53	164.55	919.97	1.11x	1.07x
3072	151.82	876.71	168.83	939.20	1.11x	1.07x

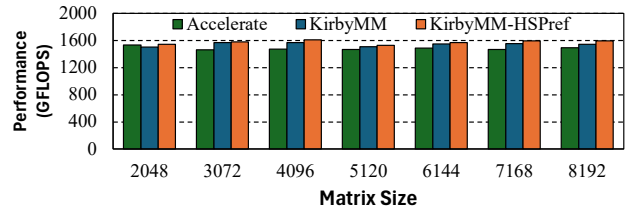


Fig. 21: Comparison with Apple Accelerate on M4.

We additionally apply HSPref to an optimized FP32 GEMM implementation (KirbyMM) [25] on M4 and compare it against Apple Accelerate, a highly optimized vendor library (Figure 21). With HSPref, the optimized kernel achieves up to 3.6% improvement over the non-prefetched version across square matrices ranging from 2048^2 to 8192^2 . Performance reaches up to 1608.32 GFLOPS and surpasses Accelerate by as much as $1.09\times$. These results confirm that HSPref is effective not only on common kernels but also on expert-optimized implementations of multiple practical operations, and that its benefits scale with problem size in real-world workloads.

VI. RELATED WORK

We review existing approaches on prefetching and memory optimizations on Arm SME.

A. Existing Prefetching Methods

Prefetching has been studied extensively across a wide range of access patterns and implementation strategies. Hardware prefetching has been well-studied, and contains stream prefetcher and other spatial [26], [27] or temporal prefetchers [28]–[31] for the complex memory patterns. Recent work has further explored ML-guided prefetching [32]–[34] and hybrid hardware-software schemes [35]–[39]. Although these are effective and continue to be actively improved by many researchers, software prefetching remains a more practical and accessible approach for most programmers, as it does not require any hardware modifications.

Software prefetching addresses this portability constraint by inserting prefetch instructions directly into the instruction stream. Compiler-directed techniques [24], [40]–[42] automate this process for regular loop nests, while analytical models [5], [6], [19] have been developed to optimize prefetch distance and coverage for indirect memory access workloads. These approaches are limited by lacking knowledge about the patterns of outer-products, the memory bottlenecks and the Arm SME microarchitecture. HSPref closes this gap by jointly modeling compute pipeline depth and memory latency, and by incorporating cache topology awareness into a lightweight, portable prefetch framework.

B. Memory Optimizations on Arm SME

Several prior works optimize memory access for specific SME workloads. Studies on GEMM [12], [13] propose blocking and packing strategies that improve data reuse in the cache hierarchy. Other research [43]–[45] introduces memory-friendly algorithms for stencil computation, Fast Fourier Transforms, and sparse matrix multiplication on SME. Most closely related to our work, recent studies [8], [16] apply explicit software prefetching to accelerate SME stencil kernels on LX2 and Apple M4. However, these prefetches are hand-tuned to specific stencil patterns, target only a single computation mode, and do not model prefetch distance analytically. HSPref generalizes these efforts into a computation-aware, topology-driven framework that supports both *data-data* and *coeff-data* outer-product patterns across platforms, freeing developers from platform-specific manual tuning.

VII. CONCLUSION

This paper presents HSPref, a software prefetching mechanism that systematically improves cache utilization for outer-product kernels on modern Arm SME processors. HSPref tackles the prefetch challenges by jointly determining the prefetch targets, the prefetch distances and cache-level placement. To facilitate practical adoption, HSPref integrates these techniques into easy-to-use APIs along with assembly-level transformation passes that enable developers to effectively leverage prefetch instructions and enhance overall performance. From our study, we derive the following key insights:

- **SME-oriented prefetching.** Existing hardware and software prefetchers are insufficient for outer-product op-

erations on Arm SME CPUs, motivating prefetching dedicated to SME.

- **Joint modeling of prefetch distance.** Effective prefetch distance for SME outer-product kernels must account for both compute and memory latency.
- **Importance of cache-level selection.** Prefetch efficiency critically depends on selecting the appropriate cache level, which must align with the SME engine’s attachment point on the memory hierarchy.

Overall, HSPref demonstrates that cross-layer, software-driven prefetching is essential for unlocking the performance potential of SME-based outer-product computations.

VIII. ACKNOWLEDGMENT

We sincerely thank the anonymous reviewers for their constructive comments and suggestions. This research was supported by Guangdong S&T Program under Grant No. 2024B0101040005, NSFC Grants (62461146204, 62472462). Han Huang and Lanshu Huang contribute equally to this work.

REFERENCES

- [1] F. Wilkinson and S. McIntosh-Smith, “An initial evaluation of arm’s scalable matrix extension,” in *2022 IEEE/ACM International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, 2022, pp. 135–140.
- [2] Samsung Semiconductor, “Expanding cpu capabilities for on-device ai with arm sme2,” <https://semiconductor.samsung.com/news-events/tech-blog/expanding-cpu-capabilities-for-on-device-ai-with-arm-sme2/>, 2023, accessed: 2023-10-25.
- [3] Apple Inc. (2024, May) CPU Optimization Guide. Accessed on 2025-10-29. [Online]. Available: <https://developer.apple.com/documentation/apple-silicon/cpu-optimization-guide>
- [4] Intel Corporation, *Intel® 64 and IA-32 Architectures Optimization Reference Manual*, May 2020. [Online]. Available: <https://www.intel.com/content/www/us/en/content-details/671488/intel-64-and-ia-32-architectures-optimization-reference-manual-volume-1.html>
- [5] G. Fu, T. Xia, M. Yin, P. J. Nair, M. Lis, and P. Ren, “Magellan: A high-performance loop-guided prefetcher for indirect memory access,” in *Proceedings of the 52nd Annual International Symposium on Computer Architecture*, ser. ISCA ’25. New York, NY, USA: Association for Computing Machinery, 2025, p. 601–615. [Online]. Available: <https://doi.org/10.1145/3695053.3731054>
- [6] S. Jamilan, T. A. Khan, G. Ayers, B. Kasikci, and H. Litz, “Apt-get: profile-guided timely software prefetching,” in *Proceedings of the Seventeenth European Conference on Computer Systems*, ser. EuroSys ’22. New York, NY, USA: Association for Computing Machinery, 2022, p. 747–764. [Online]. Available: <https://doi.org/10.1145/3492321.3519583>
- [7] R. Jain, S. Cheng, V. Kalagi, V. Sanghavi, S. Kaul, M. Arunachalam, K. Maeng, A. Jog, A. Sivasubramaniam, M. T. Kandemir, and C. R. Das, “Optimizing cpu performance for recommendation systems at-scale,” in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ser. ISCA ’23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: <https://doi.org/10.1145/3579371.3589112>
- [8] H. Huang, J. Xie, G. Feng, X. Zhang, D. Huang, Z. Chen, and Y. Lu, “Hstencil: Matrix-vector stencil computation with interleaved outer product and mla,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC ’25. New York, NY, USA: Association for Computing Machinery, 2025, p. 1816–1829. [Online]. Available: <https://doi.org/10.1145/3712285.3759769>
- [9] Apple, “Apple introduces m4 pro and m4 max,” October 2024, apple Newsroom. [Online]. Available: <https://www.apple.com/newsroom/2024/10/apple-introduces-m4-pro-and-m4-max/>

- [10] N. Stephens, S. Biles, M. Boettcher, J. Eapen, M. Eyole, G. Gabrielli, M. Horsnell, G. Magklis, A. Martinez, N. Prémillieu, A. Reid, A. Rico, and P. Walker, "The ARM scalable vector extension," *CoRR*, vol. abs/1803.06185, 2018. [Online]. Available: <http://arxiv.org/abs/1803.06185>
- [11] M. H. Noor Mohamed, A. Patil, L. Ionkov, and E. Van Hensbergen, "Improving compiler support for simd offload using arm streaming sve," in *High Performance Computing: ISC High Performance 2025 International Workshops, Hamburg, Germany, June 10–13, 2025, Revised Selected Papers*. Berlin, Heidelberg: Springer-Verlag, 2025, p. 339–348. [Online]. Available: https://doi.org/10.1007/978-3-032-07612-0_26
- [12] C. Deng, W. Yang, J. Fang, and D. Dong, "Demystifying arm sme to optimize general matrix multiplications," 2025. [Online]. Available: <https://arxiv.org/abs/2512.21473>
- [13] S. Remke and A. Breuer, "Hello sme! generating fast matrix multiplication kernels using the scalable matrix extension," 2024. [Online]. Available: <https://arxiv.org/abs/2409.18779>
- [14] ARM, "Streaming sve mode and za storage," <https://developer.arm.com/documentation/109246/0100/Introduction/The-Scalable-Matrix-Extensions/Streaming-SVE-mode-and-ZA-storage>, 2023, accessed: 2023-10-24.
- [15] PerfWiki Contributors, "Perfwiki," <https://perfwiki.github.io/main/>, 2026, accessed: 2026-04-07.
- [16] Y. Wang, T. Mao, L. Gan, W. Wan, Z. Song, J. Fu, L. He, W. Wang, Z. Yin, W. Xue, and G. Yang, "Smestencil: Optimizing high-order stencils on arm multicore using sme unit," *IEEE Transactions on Parallel and Distributed Systems*, vol. 37, no. 3, pp. 651–665, 2026.
- [17] (2023) Clang compiler user's manual. LLVM Project. Version 17.0.1. [Online]. Available: <https://releases.llvm.org/17.0.1/tools/clang/docs/index.html>
- [18] Apple, "Command line tools for xcode 16.2 beta 2," 2024, apple Developer Tools. [Online]. Available: https://download.developer.apple.com/Developer_Tools/Command_Line_Tools_for_Xcode_16.2_beta_2/Command_Line_Tools_for_Xcode_16.2_beta_2.dmg
- [19] S. Ainsworth and T. M. Jones, "Software prefetching for indirect memory accesses," in *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2017, pp. 305–317.
- [20] Timoheimonen, "macOS-memory-benchmark: Low-level apple silicon memory benchmark for measuring cache latency and memory behavior on macOS (ARM64)," <https://github.com/timoheimonen/macos-memory-benchmark>, 2025.
- [21] Apple Inc., *Xcode*, 2026, accessed: 2025-04-05. [Online]. Available: <https://developer.apple.com/xcode/>
- [22] L.-N. Pouchet, "PolyBench/C: the Polyhedral Benchmark suite," <https://www.cs.colostate.edu/~pouchet/software/polybench/>, Ohio State University, 2012, version 3.2.
- [23] Q. Li, K. Li, H. Han, L. Yuan, Y. Zhang, Y. Chen, J. Chen, H. An, T. Cao, and M. Yang, "Sparstencil: Retargeting sparse tensor cores to scientific stencil computations via structured sparsity transformation," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 1495–1509. [Online]. Available: <https://doi.org/10.1145/3712285.3759820>
- [24] T. C. Mowry, M. S. Lam, and A. Gupta, "Design and evaluation of a compiler algorithm for prefetching," in *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS V. New York, NY, USA: Association for Computing Machinery, 1992, p. 62–73. [Online]. Available: <https://doi.org/10.1145/143365.143488>
- [25] L. Huang, H. Huang, Z. Chen, and Y. Lu, "Kirbymm: Outer-product based matrix multiplication on armv9 processor," in *2026 Design, Automation & Test in Europe Conference (DATE)*, 2026, pp. 1–7.
- [26] R. Bera, A. V. Nori, O. Mutlu, and S. Subramoney, "Dspatch: Dual spatial pattern prefetcher," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-52. New York, NY, USA: Association for Computing Machinery, 2019, p. 531–544. [Online]. Available: <https://doi.org/10.1145/3352460.3358325>
- [27] M. Bakhshalipour, M. Shakerinava, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Bingo spatial data prefetcher," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019, pp. 399–411.
- [28] M. Li, Q. Zhang, Y. Gao, W. Fang, Y. Lu, Y. Ren, and Z. Xie, "Profile-guided temporal prefetching," in *Proceedings of the 52nd Annual International Symposium on Computer Architecture*, ser. ISCA '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 572–585. [Online]. Available: <https://doi.org/10.1145/3695053.3731070>
- [29] F. Xue, C. Han, X. Li, J. Wu, T. Zhang, T. Liu, Y. Hao, Z. Du, Q. Guo, and F. Zhang, "Tyche: An efficient and general prefetcher for indirect memory accesses," *ACM Trans. Archit. Code Optim.*, vol. 21, no. 2, Mar. 2024. [Online]. Available: <https://doi.org/10.1145/3641853>
- [30] H. Wu, K. Nathella, J. Pusdesris, D. Sunwoo, A. Jain, and C. Lin, "Temporal prefetching without the off-chip metadata," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-52. New York, NY, USA: Association for Computing Machinery, 2019, p. 996–1008. [Online]. Available: <https://doi.org/10.1145/3352460.3358300>
- [31] H. Wu, K. Nathella, D. Sunwoo, A. Jain, and C. Lin, "Efficient metadata management for irregular data prefetching," in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 449–461. [Online]. Available: <https://doi.org/10.1145/3307650.3322225>
- [32] C. Xu, X. Sun, W. Li, C. Bai, B. Wang, M. Li, Z. Xie, and Y. Xie, "Pf-llm: Large language model hinted hardware prefetching," in *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS '26. New York, NY, USA: Association for Computing Machinery, 2026, p. 1430–1444. [Online]. Available: <https://doi.org/10.1145/3779212.3790202>
- [33] C. Block, G. Gerogiannis, and J. Torrellas, *Micro-MAMA: Multi-Agent Reinforcement Learning for Multicore Prefetching*. New York, NY, USA: Association for Computing Machinery, 2025, p. 884–898. [Online]. Available: <https://doi.org/10.1145/3725843.3756096>
- [34] M. Li, Q. Zhang, Y. Ren, and Z. Xie, "Integrating Prefetcher Selection with Dynamic Request Allocation Improves Prefetching Efficiency," in *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. Los Alamitos, CA, USA: IEEE Computer Society, Mar. 2025, pp. 204–216. [Online]. Available: <https://doi.ieeecomputersociety.org/10.1109/HPCA61900.2025.00026>
- [35] Y. Chen, J. Feng, Z. Wang, C. J. Hughes, and J. Huang, "Software prefetch multicast: Sharer-exposed prefetching for bandwidth efficiency in manycore processors," in *Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 156–169. [Online]. Available: <https://doi.org/10.1145/3725843.3756049>
- [36] T. Zhang, B. Grot, W. He, Y. Lv, P. Qu, F. Su, W. Wang, G. Zhang, X. Zhang, and Y. Zhang, *Hierarchical Prefetching: A Software-Hardware Instruction Prefetcher for Server Applications*. New York, NY, USA: Association for Computing Machinery, 2025, p. 529–544. [Online]. Available: <https://doi.org/10.1145/3676641.3716260>
- [37] H. Li, K. Liu, T. Liang, Z. Li, T. Lu, H. Yuan, Y. Xia, Y. Bao, M. Chen, and Y. Shan, "Hopp: Hardware-software co-designed page prefetching for disaggregated memory," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2023, pp. 1168–1181.
- [38] C. Zhang, Y. Zeng, J. Shalf, and X. Guo, "Rnr: A software-assisted record-and-replay hardware prefetcher," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020, pp. 609–621.
- [39] N. Talati, K. May, A. Behrooz, Y. Yang, K. Kaszyk, C. Vasiladiotis, T. Verma, L. Li, B. Nguyen, J. Sun, J. M. Morton, A. Ahmadi, T. Austin, M. O'Boyle, S. Mahlke, T. Mudge, and R. Dreslinski, "Prodigy: Improving the memory latency of data-indirect irregular workloads using hardware-software co-design," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021, pp. 654–667.
- [40] D. Callahan, K. Kennedy, and A. Porterfield, "Software prefetching," in *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS IV. New York, NY, USA: Association for Computing Machinery, 1991, p. 40–52. [Online]. Available: <https://doi.org/10.1145/106972.106979>
- [41] E. H. Gornish, E. D. Granston, and A. V. Veidenbaum, "Compiler-directed data prefetching in multiprocessors with memory hierarchies," in *Proceedings of the 4th International Conference on Supercomputing*, ser. ICS '90. New York, NY, USA: Association for Computing

- Machinery, 1990, p. 354–368. [Online]. Available: <https://doi.org/10.1145/77726.255176>
- [42] G. Ayers, H. Litz, C. Kozyrakis, and P. Ranganathan, “Classifying memory access patterns for prefetching,” in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’20. New York, NY, USA: Association for Computing Machinery, 2020, p. 513–526. [Online]. Available: <https://doi.org/10.1145/3373376.3378498>
 - [43] R. Zhang, H. Jia, Y. Zhang, B. Yan, P. Ma, L. Wang, and W. Zhao, “Openfft-sme: An efficient outer product pattern fft library on arm sme cpus,” in *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2024, pp. 938–949.
 - [44] W. Zhao, L. Yuan, B. Yan, P. Ma, Y. Zhang, L. Wang, and Z. Wang, “Stencil computation with vector outer product,” in *Proceedings of the 38th ACM International Conference on Supercomputing*, ser. ICS ’24. New York, NY, USA: Association for Computing Machinery, 2024, p. 247–258. [Online]. Available: <https://doi.org/10.1145/3650200.3656611>
 - [45] K. Lei, H. Yang, K. Zhang, K. Ma, Y. Wang, X. You, Y. Xu, E. S. Quintana-Orti, Z. Luan, Y. Liu, and D. Qian, “Low-cost yet high-performant sparse matrix-matrix multiplication on arm sme architectures,” 2025. [Online]. Available: <https://arxiv.org/abs/2511.08158>

A. Abstract

This artifact provides **HSPref**, a software prefetching framework designed for Arm SME outer-product kernels, together with the source code, benchmark suite, compilation scripts, and evaluation scripts required to reproduce the experimental results presented in the paper.

B. Description

1) *Overview of Contributions and Artifacts*: This artifact enables reproduction of the main techniques and experimental results presented in the paper. It supports the following contributions:

- C_1 The artifact includes representative SME outer-product kernels and benchmark workloads used to analyze the memory-access characteristics of SME execution.
- C_2 The artifact provides the implementation of HSPref, including prefetch target selection, DAG-based prefetch-distance estimation, and cache-level selection.
- C_3 The artifact provides the evaluation framework for the performance and efficiency results on LX2 and Apple M4. The workflow measures execution performance, cache behavior, memory stalls, and portability across different SME platforms.

The computational artifact is archived under the DOI below:

A_1 <https://doi.org/10.5281/zenodo.21548721>

Artifact ID	Contributions	Paper Elements
A_1	C_1, C_2, C_3	Figure 13–21

2) *How to access*: The artifact is distributed as a single Zenodo archive (A_1). The directory structure is organized as follows:

- `HSPref/src/` contains the assembly-level prefetch injection pass and the HSPref APIs.
- `HSPref/benchmarks/` contains the benchmark suite, including 10 PolyBench kernels, 2 SparStencil kernels, and additional applications.
- `HSPref/dependencies/LX2/` and `HSPref/dependencies/M4/` provide platform-specific dependency installation scripts.
- `HSPref/scripts/` contains experiment automation scripts.
- `HSPref/figures/` contains the data template (`data.xlsx`) and Python scripts (`plot.py`) required to regenerate the figures reported in the paper.
- `HSPref/tests/` contains hardware detection utilities used to verify hardware metrics.

3) *Hardware dependencies*: HSPref targets Arm processors supporting the Scalable Matrix Extension (SME).

LX2 The LX2 platform consists of 8 NUMA nodes, with 38 CPU cores per NUMA node. Each core integrates an SME unit connected to a private 32 KiB L1 data cache and a 768 KiB L2 cache. The system uses DDR memory.

M4 The Apple M4 platform contains CPU cores sharing an SME unit within each cluster. The SME unit is connected to a shared 16 MB L2 cache, while each core has a private 128 KiB L1 data cache. The system uses DDR memory.

4) *Software dependencies*: The artifact supports two software environments corresponding to the two evaluation platforms. All required dependencies are provided through platform-specific installation scripts.

LX2 The LX2 evaluation environment requires:

- **LX2 Clang**: the Arm SME-enabled compiler provided with the evaluation environment.
- **CMake v3.31.0**: used to configure and build the artifact.
- **perf v5.10.0**: the Linux performance analysis tool used to collect hardware performance counters.

M4 The Apple M4 evaluation environment requires:

- **Homebrew v6.0.12**: used for dependency management.
- **Apple Clang v21.0.0**: provided by Xcode Command Line Tools.
- **CMake v4.3.3**: used for build configuration.
- **Xcode v26.6**: used for performance profiling with Xcode Instruments.

5) *Data sets*: The artifact includes GEMM-like workloads where both input operands are dynamically loaded from memory (data-data kernels). It also includes workloads where one operand behaves as a relatively small coefficient array while the other operand contains the primary data stream (coeff-data kernels). The portability evaluation additionally includes FlashAttention and an optimized FP32 GEMM implementation compared against Apple Accelerate on Apple M4.

For cache-resident evaluations, the artifact uses matrix sizes around 1024^2 . For memory-intensive evaluations, the matrix sizes range from 2048^2 to 8192^2 .

6) *Models*: HSPref does not require machine-learning models or external model files.

C. Installation

The artifact provides platform-specific scripts for dependency installation and compilation.

- **LX2**. Run `dependencies.sh` to install LX2 Clang and CMake.
- **Apple M4**. Install Homebrew and run `dependencies.sh` to install CMake and Xcode Command Line Tools.

The benchmark suite can be compiled using `compile.sh`, which configures the required SME compilation options, including `-mcpu=lx2` for LX2 and `-mcpu=armv8-a+sme2` for Apple M4.

D. Experiment workflow

The artifact provides scripts under `HSPref/scripts` for the main experiments. Before execution, users should source

`env.sh`. On LX2, `numactl` is used to control CPU and memory affinity.

- 1) **Performance evaluation (Fig. 13).** Build and run the HSPref-enabled and baseline kernels. The scripts measure execution time and report speedup over the no-prefetch implementation. Each experiment is repeated five times.
- 2) **Efficiency analysis (Fig. 14–16).** On LX2, users can run `perf stat` to collect hardware metrics, including cache misses, MPKI, and stall cycles.
- 3) **Sensitivity and model comparison (Fig. 17–19).** Users can evaluate different problem sizes, fixed prefetch distances (4/16/64), and previous prefetch-distance models including Mowry, SW Prefetch, and APT-GET.
- 4) **Scalability and portability (Fig. 20–21).** The scripts evaluate multi-core performance with 2/16/32 threads and reproduce the FlashAttention and GEMM portability studies on LX2 and Apple M4.

E. Evaluation and expected results

After running the provided workflow, users should obtain results consistent with those reported in the paper. Minor variations may occur due to compiler versions and system conditions.

Performance (Fig. 13). HSPref achieves geometric-mean speedups of $1.34\times$ on LX2 and $1.38\times$ on Apple M4 compared with the no-prefetch baseline. Automatic compiler prefetching provides limited improvements.

Efficiency (Fig. 14–16). HSPref reduces backend stall rate from 83.4% to 71% and decreases MPKI by 23% on average. The average prefetch coverage reaches 76%, with over 90% coverage for several memory-intensive kernels.

Sensitivity and model comparison (Fig. 17–19). For large memory workloads, HSPref achieves approximately 1.30 – $1.40\times$ speedup over no-prefetch. Fixed prefetch distances and previous models provide lower performance than HSPref, demonstrating the effectiveness of the analytical distance selection.

Scalability and portability (Fig. 20–21). HSPref maintains performance benefits in multi-core execution, improving from $1.23\times$ speedup with 2 threads to $1.29\times$ with 32 threads. FlashAttention and GEMM evaluations further demonstrate portability across different SME platforms.

F. Experiment customization

The artifact allows users to customize the evaluation configuration through provided scripts and benchmark parameters.

- **Matrix and problem size.** Users can modify the size parameters in each benchmark program, such as N , M , and K , to reproduce the size sensitivity evaluation.
- **Prefetch distance.** HSPref automatically derives the prefetch distance using its analytical model. Users can also configure fixed prefetch-distance baselines (4/16/64) to reproduce the comparison experiments.
- **Cache level selection.** HSPref automatically selects the target cache level according to the SME topology of the

evaluated processor. Users can modify this policy for architecture exploration.

G. Notes

Cache-level performance metrics, including L2 hit rate, MPKI, and backend stall breakdown, are reported only for LX2. This limitation is due to the restricted cache performance counter availability provided by Xcode Instruments on Apple M4.