The Bicameral Cache: a split cache for RISC-V vector architectures

Susana Rebolledo[1*], Borja Perez[1], Jose Luis Bosque[1] and Peter Hsu[2]

[1]Department of Computer Science and Electronic Engineering, University of Cantabria, Santander, Spain
[2]Peter Hsu Consulting, Inc. Barcelona, Spain and San Francisco, USA

Abstract

This paper presents the design and evaluation of the Bicameral Cache, a memory hierarchy for vector processors that separates scalar and vector references into distinct partitions. This design aims to enhance vector application performance by reducing scalar instruction interference and ensuring the continuity of vector elements. Additionally, a prefetching option is included to improve performance by exploiting spatial locality in vector references. The Cavatools simulator, supporting the RISC-V vector extension, was used for evaluation. Simulations of eight benchmark types across various architectural vector lengths show that the proposed cache significantly benefits sequential memory access patterns while having minimal impact on non-contiguous ones. Moreover, the prefetching feature consistently enhances performance.

Introduction

Vectorization, which exploits data-level parallelism, is widely used in both supercomputers and commercial general-purpose processors. While it optimizes performance by operating on multiple data simultaneously, its effectiveness depends on memory performance, specifically the speed of data availability. Reducing memory access latency is crucial to bridging this performance gap. Given the different access patterns and locality of scalar and vector references, dedicated memory solutions can significantly enhance performance and energy efficiency in vector architectures. Common strategies to exploit locality include the memory hierarchy and prefetching.

We propose the Bicameral Cache, a segregated cache architecture for vector processors that splits data based on access type –vector or scalar. The design, evaluated through simulation, seeks to boost the performance of vector applications by leveraging the locality of vector data. Its dedicated partitions preserve the temporal locality of each data type, preventing scalar interference in vector data. Additionally, our proposal incorporates a prefetching feature that fills vector cache lines with data in advance to exploit their spatial locality.

The Bicameral Cache

The Bicameral Cache (BC) is a data cache memory system for vector architectures composed of two different cache structures; the Scalar Cache (SC) and the Vector Cache (VC). Its aim is to preserve the data locality inherent in vector computation by preventing potential interference from the scalar one. Following a similar approach to [1], the lines in both caches are sectorized (i.e. organized in sectors). A sector is defined as the minimum data transfer unit between the main memory and the caches. Its size is fixed to the length of the SC lines. VC lines are significantly longer, therefore composed of several sectors, to better exploit the spatial locality on the vector data. Lines have an tag for identification, while each sector uses 2 bits to determine the state of their data; valid (v) and dirty (d). Both caches use LRU replacement and a write-back policy. Despite using specific sizes for modelling, such as 64 B sectors and up-to-8-lines write buffers (WB), the proposed cache is size-agnostic.

Scalar Cache

The Scalar Cache (SC) stores data referenced by scalar memory instructions. It has a set-associative structure of 256 4-way sets which, with its 64 B-long lines (sector size), sums up a total capacity of 64 KB. Figure 1a shows a graphical representation of its structure. To avoid processor stalls on store operations, the Scalar Cache includes an additional 8-line write buffer (WB) that stores the evicted lines with modified sectors.

Vector Cache

The Vector Cache (VC) stores data referenced by vector memory instructions in a fully-associative structure, where each 1024 B-long line fits 16 sectors. Hence, the 8 write buffer rows to store whole cache lines containing modified sectors when evicted require significantly larger storage capacity than in SC. To mitigate this, the VC embeds the WB, enabling the use of its free lines as regular cache lines. Consequently, the VC capacity varies dynamically depending on the write buffer occupancy. Figure 1b depicts such organization.

*Corresponding author: susana.rebolledo@unican.es

Set	Line	Sector
0	0	
	1	
	2	
	3	
. . .	. . .	. . .
255	0	
	1	
	2	
	3	

WB0	
WB1	
WB2	
WB3	
WB4	
WB5	
WB6	
WB7	

(a) *Scalar Cache (left) and its write buffer (right).*

	Sector				
Line	0	1	···	14	15
0			···		
. . .			. . .		
63			···		
WB0			···		
WB1			···		
WB2			···		
WB3			···		
WB4			···		
WB5			···		
WB6			···		
WB7			···		

(b) *Vector Cache. Shadowing represents modified sectors from WB lines awaiting write-back to memory (i.e. valid and dirty).*

Figure 1: *Representation of the Bicameral Cache.*

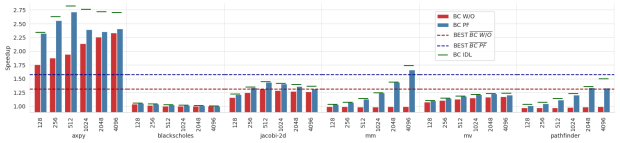
Other features

We use mutual exclusivity to ensure a sector is never in both caches simultaneously. After a miss in the cache corresponding to the instruction type (native lookup), the opposite cache is probed first (cross lookup) before accessing memory. For vector data, if found in the SC after a miss (cross hit), the sector is migrated to VC. This migration policy prioritizes vector data in VC to enhance vector instruction performance, allowing scalar instructions to access vector data without disrupting VC's continuity. Valid sectors from WB are either restored to cache if referenced again or written back to memory if dirty. Our approach includes memory-side prefetching to proactively fill VC lines, reducing miss rates and improving performance.

Methodology

Baseline. We evaluate against a single conventional cache or white cache (WC) with same capacity (128 KB), whose structure is similar to SC.

Prefetching effectiveness is compared to an ideal version that fetches all sectors in the VC line simultaneously, with the same latency as fetching a single sector. This approach would fully populate the vector line on the first compulsory miss, without additional penalties.



BC: Bicameral Cache, W/O: without prefetching (red), PF: with prefetching (blue), IDL: with ideal prefetching (green line).

Figure 2: *Performance evaluation.*

Memory model. A simplified main memory model was used to simulate the behaviour of the BC as part of the memory hierarchy; a 4GB DRAM technology with 8 banks, each supporting a single open row at a time. The modelled memory controller enqueues requests into each bank's queue, scheduling on a FCFS basis.

Simulator. We extend Cavatools [2], an open-source RISC-V ISA simulator.

Vector benchmarks. We use `axpy`, `blackscholes`, `jacobi-2d`, `pathfinder` and `lavaMD` from [3], and in-house matrix-matrix (`mm`), matrix-vector (`mv`) and sparse matrix-vector (`spmv`) multiplications.

Vector architecture. We evaluate on a range of architectural vector lengths, from 128 to 4096 bits.

Results

The Bicameral Cache improves performance on stride-1 benchmarks with an average best-case speedup of 1.31x over the baseline, which grows up to 1.57x if enabling the prefetching (Figure 2). In addition, for non-stride-1 workloads, the performance remains mostly unchanged with the basic configuration, yet it improves on an 11% for the average best-case scenario with prefetching.

Overall, these results, driven by a substantial reduction in the average memory access time, represent a significant improvement, as they are achieved without additional hardware or an increase in the cache size; just by restructuring and leveraging the available resources to better exploit the data locality. Finally, prefetching proved to be a succesful optimization.

References

[1] J.B Rothman. "Sector cache design and performance". In: *Proceedings Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*. 2000.

[2] P. Hsu. *Cavatools*. GitHub. URL: `https://github.com/phaa-eu/cavatools`.

[3] C. Ramírez et al. "A RISC-V Simulator and Benchmark Suite for Designing and Evaluating Vector Architectures". In: *ACM Trans. Archit. Code Optim.* (2020).