NACHOS: Software-Driven Hardware-Assisted Memory Disambiguation for Accelerators

Naveen Vedula, Arrvindh Shriraman, Snehasish Kumar, and William N Sumner
School of Computing Sciences
Simon Fraser University, Burnaby, Canada
Email: {nvedula, ashriram, ska124, wsumner}@sfu.ca

***Abstract*—Hardware accelerators have relied on the compiler to extract instruction parallelism but may waste significant energy in enforcing memory ordering and discovering memory parallelism. Accelerators tend to either serialize memory operations [43] or reuse power hungry load-store queues (LSQs) [8], [27]. Recent works [11], [15] use the compiler for scheduling but continue to rely on LSQs for memory disambiguation.**

NACHOS is a hardware assisted software-driven approach to memory disambiguation for accelerators. In NACHOS, the compiler classifies pairs of memory operations as NO alias (i.e., independent memory operations), MUST alias (i.e., ordering required), or MAY alias (i.e., compiler uncertain). We developed a compiler-only approach called NACHOS-SW that serializes memory operations both when the compiler is certain (MUST alias) and uncertain (MAY alias). Our study analyzes multiple stages of alias analysis on 135 acceleration regions extracted from SPEC2K, SPEC2k6, and PARSEC. NACHOS-SW is energy efficient, but serialization limits performance; 18%–100% slowdown compared to an optimized LSQ. We then proposed NACHOS a low-overhead, scalable, hardware comparator assist that dynamically verifies MAY alias and executes independent memory operations in parallel. NACHOS is a pay-as-you-go approach where the compiler filters out memory operations to save dynamic energy, and the hardware dynamically checks to find MLP. NACHOS achieves performance comparable to an optimized LSQ; in fact, it improved performance in 6 benchmarks(6%—70%) by reducing load-to-use latency for cache hits. NACHOS imposes no energy overhead in 15 out of 27 benchmarks i.e., compiler accurately determines all memory dependencies; the average energy overhead is $\simeq$6% of total (accelerator and L1 cache); in comparison, an optimized LSQ consumes 27% of total energy. NACHOS is released as free and open source software. Github: https://github.com/sfu-arch/nachos

***Keywords*-Hardware accelerators; Dataflow; Memory disambiguation; Load-Store Queues; Alias Analysis;**

I. Introduction

Out-of-order(OOO) execution enables high performance by executing many instructions in parallel. Memory disambiguation is required to enforce program order between operations accessing the same memory location (for correctness) while executing those that access different locations in parallel (for performance). Accelerators execute instructions out-of-order as well (typically with a dataflow architecture) and need memory disambiguation support as well.

Often hardware accelerators target regular algorithms where memory dependencies are resolved early on at hardware design time, and the dataflow is carefully orchestrated [23], [44]. Spatial accelerators that tend to target a broad range of program behavior (e.g., CGRAs [29], Dyser [8]) rely on the compiler to find instruction level parallelism. Still they continue to rely on the OOO's Load-Store-Queue (LSQ) to enforce memory ordering and are integrated into OOO's pipeline; this limits the number of operations in the accelerated region. Some accelerators constrain the accelerated region (e.g., basic blocks [43] and compound function units [10]), to only have a single memory operation further limiting granularity. Recent work has also shown that significant energy efficiency can be gained from recognizing nearby store-load dependencies and localizing the communication [11], [12]. Memory disambiguation influences multiple parameters in accelerators including performance, energy, design effort and integration with the processor.

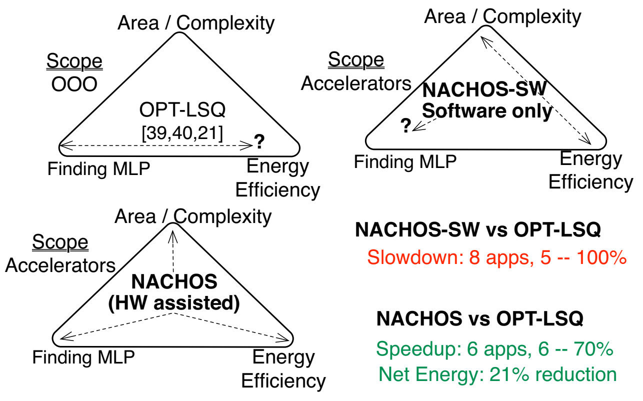


Figure 1: Optimized LSQ designs vs NACHOS. Arrows indicate the parameters that are targeted and improved.

The most prevalent approach amongst accelerators [8], [27], [28] that require memory disambiguation is to reuse the OOO's LSQ. While some rely on the compiler to optimize the memory access patterns [11], [15] they continue to use a separate energy inefficient LSQ. Figure 1 highlights the tradeoffs in LSQs in a hardware accelerator context. In a Coarse Grain Reconfigurable Array (CGRA) based accelerator, we find that the LSQ dominates overall energy consumption (since other overheads are minimal); even an optimized LSQ (OPT-LSQ) that is partitioned and filters power-hungry Content Addressable Memory (CAM) accesses [30], [32], [34]–[36], [41] accounts for 27% of total energy. It is also unclear if the LSQ (ports and size) can be scaled to match the memory parallelism available in the acceleration region without additional hardware [16], [33]. Section VIII-C includes details on the baseline OPT-LSQ we evaluate.

Our Approach: We propose NACHOS, software-driven hardware-assisted memory disambiguation for hardware accelerators. NACHOS uses an LLVM-based prototype compiler which determines the aliasing relationship between all pairs of memory operations in the accelerated region. The pairwise relations are labeled as *MAY, MUST or NO* alias to indicate whether a pair of memory operations may, will or never access the same memory location. NACHOS enforces memory ordering between operations pairwise, unlike an LSQ's centralized approach. NACHOS handles dependent (MUST alias) memory operations, similar to the data dependencies in a dataflow graph. Memory operations which do not alias (NO alias) proceed in parallel. However, often the compiler lacks enough information to classify a pairwise relation as MUST or NO alias. One conservative approach employed in prior work [25], [31] is to treat MAY aliases as MUST alias. Younger memory operations are stalled to ensure correctness as the compiler is unable to determine whether they access the same memory location. However, pathological scenarios may arise where a single ambiguous memory operation may serialize potential parallelism. NACHOS deploys hardware checks to dynamically compare the addresses of MAY alias memory operation pairs. Based on the check the hardware either enforces the ordering if required or allows the operations to proceed in parallel. We also discuss NACHOS-SW, a software-only approach (similar to Tartan [25]) that enforces all dependencies both when the compiler is certain (MUST aliases) and uncertain (MAY aliases); this helps study the limit of a software-only approach.

As far as we are aware, we are the first to comprehensively quantify the energy and performance impact of using alias analysis to enforce memory ordering for hardware accelerators. Seminal works in application specialization (e.g., [4], [25], [43]) mention the use of alias analysis. However, they neither discuss qualitatively nor evaluate quantitatively the impact of using compiler based alias analysis to enforce correct memory ordering. Other non-accelerator works have leveraged alias analysis for optimally scheduling memory operations (e.g., [1], [11], [42]) but continue to rely on LSQ-like structures for correctness. Our contributions:

- We will release NACHOS-SW, an LLVM-based compiler that leverages alias analysis for memory disambiguation in hardware accelerators. NACHOS-SW adds support for inter-procedural analysis, polyhedral analysis [9], and optimizations for removing redundant alias checks (see § V).
- We study 135 accelerated regions (across 27 benchmarks) with four stages of alias analysis and find that while a software-only approach (NACHOS-SW) improves energy efficiency over an LSQ, compiler ambiguity excessively serializes operations. Six applications; 18%—100% slowdown (see § VI).
- We propose NACHOS, a decentralized hardware assist that dynamically disambiguates accesses to extract memory parallelism (see § VII). NACHOS's performance is comparable to an optimized LSQ (OPT-LSQ). In six workloads performance improves (6%—70%) due to improved load-to-use latency for cache hits. NACHOS saves 21% energy compared to OPT-LSQ (see § VIII).

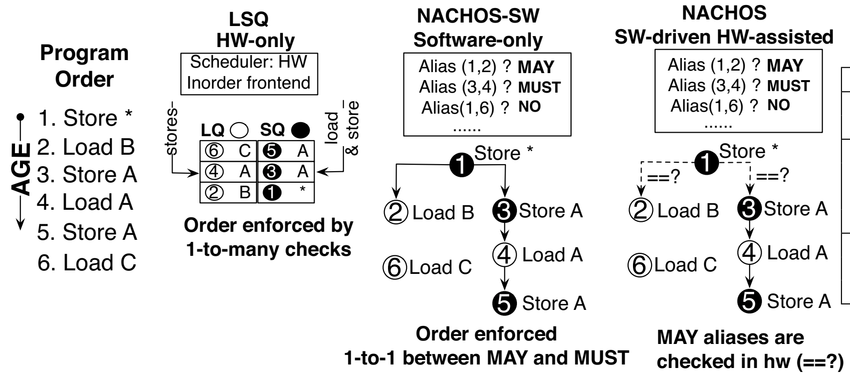
II. Scope and Related Work

Here we contrast the centralized approach adopted by conventional LSQs to a distributed approach proposed by NACHOS. We also discuss current accelerator designs and how they can benefit from NACHOS (§ II-B).

A. Memory ordering with LSQs and NACHOS

Memory disambiguation helps enforce the ordering between memory operations which access the same location from a single thread in a program. The following orderings are enforced (see Figure 2:example): i) *ST-ST* ordering: in-order retirement of stores to the same address to ensure the final value in a location (❸ → ❺) , ii) *ST-LD* ordering, to forward in-flight values from older stores to younger loads (❸ → ④), and iii) *LD-ST* ordering, to ensure that stores do not overwrite the values of older loads (④→❺); *LD-LD* ordering is only required in parallel racy programs.

Figure 2 illustrates the LSQ; a content-addressable queue each for load and store operations. LSQs logically order entries based on their age (either by physically ordering entries based on age or by explicitly encoding the age). Every



	LSQ	NACHOS-SW	NACHOS
Scope	OOO	Accelerators	
Sched.	Implicit	Explicit (dataflow)	
Ordering	Runtime	Compile time	Hybrid
Hardware	1-N CAM	None	(==)
FalseOrder	No	Yes (possible)	No
Perf.	✓	—	✓
Scalability	—	✓	✓
Energy	—	✓	✓

Figure 2: LSQ vs NACHOS-SW and NACHOS. Arrow between operations indicate memory dependency edges (MDEs).

in-flight memory operation has to perform an expensive 1-N search against every other memory operations to identify potential matches. Loads (◯ in the figure) check for older stores in the Store Queue (SQ) and matched stores forward their values. Stores check (●) for matches in both queues to detect potential ordering violations. A challenge unique to dataflow based architectures [34] is that they lack a front-end and the LSQs have to rely on a compiler to explicitly indicate the age. This can lead to over-provisioning of the LSQ [40] and increased occupancy [13]; here we observe a noticeable performance impact due to inorder issue of memory operations (see § VI) Another challenge with incorporating LSQ-based approaches is scaling up and down with the number of memory operations and memory level parallelism (MLP) in the accelerated region. In the accelerated region, memory operations can vary a lot, from 0–38% and MLP between 2—128 (see Table II). This impacts energy and area; Section VIII-C includes details on the baseline.

NACHOS-SW is a software-only approach (see Figure 2). The compiler analyzes pairwise relations between memory operations and assigns a label for every pair: NO alias, MUST alias (operations need to be ordered), or MAY alias (the compiler is uncertain). In the example shown, NACHOS-SW identified that i) ❸→④ alias with each other, ii) ② does not alias with [3,4,5], iii) ❶ may alias with [2,3,4,5], and iv) ⑥ does not alias at all. The compiler inserts no dependence edge into ⑥;, it can execute concurrently with all operations. To enforce ordering between the MUST alias operations (3,4,5) and MAY alias operation (1), the compiler inserts a dataflow ordering edge between the memory operations. These edges are enforced by the accelerator (similar to a data dependence). Since NACHOS-SW has no runtime information it conservatively enforces ordering from ❶ which increases the critical path. For instance, if ❶ misses in the cache other operations and their forward slices are stalled limiting overall performance. NACHOS-SW is orthogonal to pipecheck [22] which verifies whether a microarchitecture graph (specifying the CPU's hardware) preserves the program order for a particular litmus test.

NACHOS adds a hardware assist to check the memory operations when the compiler is uncertain (i.e., MAY alias). Unlike NACHOS-SW which serializes memory operations when the compiler is uncertain, NACHOS can accommodate varying degrees of MLP and number of memory operations. To handle compiler may alias operations (21% of all pairwise memory operation checks), NACHOS sets up a may alias enforcement edge (❶ $\overset{==?}{\dashrightarrow}$ to ②, ❸). An additional hardware comparator implemented within the dataflow accelerator dynamically checks the addresses whether they alias. If they do not alias then both ② and ❸ are permitted to proceed in parallel with ❶ increasing MLP. NACHOS implements the address comparison at the site of the younger memory operation i.e., ② and ❸, decentralized and in parallel. In NACHOS, the compiler based alias analysis is used to minimize the number of hardware checks needed. For instance, ❶ does not have any (==?) to ④,❺ since the compiler knows that 1, 4 and 5 already must alias. Overall, NACHOS is pay-as-you-go approach and employs dynamic checks only when the software is uncertain, and even then enforces only the least number of pair-wise checks required to find all the MLP.

B. How does NACHOS benefit accelerators?

Table I discusses the trade-offs in how different accelerator types handle memory accesses. Prior approaches can be broadly classified into four designs: i) compute-only accelerators, ii) compound function units with minimal memory operations (typically 1) iii) access accelerators that leverage regular kernel behavior and data parallelism and iv) whole program accelerators that target arbitrary regions.

Compute-only accelerators depend on the OOO core to perform memory accesses and thus require frequent interactions with the processor. Achieving high energy efficiency is challenging in workloads with more memory operations. Compound function unit (CFU) based accelerators combine frequently used operations but often terminate CFUs at a memory operation, limiting acceleration granularity. Memory

		Compute-Only Dyser [8]	**Compound Unit** CFU [4], [10],C-Core [43]	**Access Accelerator** MAD [15],DESC [11]	**Program Accelerator** Tartan [25],SEED [27]
Target	Scope	Hyperblocks $\cancel{③}$		Kernels	Program
	Granularity	10s of ops $\cancel{③}$	10s of ops	100s of ops	Nested loops
Design	Mem. Ops?	None;	1-per block $\cancel{①}$	Regular: multiple ops. Irregular: Depends. $\cancel{①}$	
	Mem. Ordering	OOO	Inorder	Large LSQ $\cancel{④}$	Serialize and LSQ [27]
	MLP	Limited		High	Depends
	Integration	Close to OOO		Uncore	L1-cache
	⇔OOO	High $\cancel{②}$	Medium (on block termination) $\cancel{②}$		N/A
	Compiler support	N/A	Targeted use. CFU design [10], Parallelizing NO alias loads [11], [25]		
Benefits of NACHOS for each accelerator type					
	Find MLP ①	✓	✓	✓Irregular program regions	
	Energy Efficiency ②	✓Decouple from OOO		✓Decouple from LSQ	
	Coarse Offload.③	✓Multiple mem-ops, Increase Accel. granularity		✓Irregular programs	
	Low HW overhead.④	✓. Accelerators only need ability to enforce dependencies explicitly.			

Table I: Comparison of how accelerators handle memory accesses. Also lists the specific benefits of NACHOS for each accelerator.

ordering demands that the compound operations execute in order which also limits performance. As observed by prior work [27], such designs continue to rely on an OOO for achieving performance. Access accelerators largely focus on optimizing the schedule and continue to use a conventional LSQ for memory ordering. These techniques focus on regular data loops and kernels, and high MLP requires large multi-ported LSQs (DESC [11, Section 3.3]: 128 entries, MAD [15, Section 5.4]: 4 ported LSQs). Perhaps the most closely related work to ours is accelerators that target whole programs and do not rely on the OOO core for memory accesses and memory ordering. Such accelerators have primarily been demonstrated on loop-dominated workloads [25] and in the presence of irregular memory accesses (e.g., linked lists) tend to serialize execution or require LSQs [27]. Prior work has neither qualitatively nor quantitatively studied the influence of alias analysis on performance and energy efficiency of hardware accelerators. VLIW processors [1], [6], [21] and few recent works [11], [42] have leveraged alias analysis for scheduling independent operations. If the operations are not independent (or compiler is unsure), they either serialize (performance suffers) or continue to rely on LSQ-like associative structures.

NACHOS is suited for architectures that implement the dataflow graph and enforces data dependencies explicitly, either with custom netlist [37], [43] or as a spatial fabric, e.g., CGRA [8]. The dataflow-based accelerators are a natural fit for the compiler-based memory ordering since they already enforce data dependencies explicitly between operations. See §VII for how we overload existing dataflow architecture mechanisms. NACHOS benefits other accelerators in the following manner i) Extracting MLP: NACHOS finds all the MLP available, either with the compiler analysis or by hardware runtime checks (where the compiler is uncertain), ii) Energy efficiency: NACHOS leverages the compiler to filter away many of the hardware checks and even when checking employs decentralized checks (in-lieu of the LSQ's centralized approach) iii) Decoupling from OOO: NACHOS permits greater flexibility by effectively handling memory ordering. Compute-only accelerators and CFU accelerators can include multiple memory operations and increase their granularity. Access accelerators and whole program accelerators can effectively deal with irregular programs since NACHOS can extract MLP with a combined hardware-software approach. iv) Low hardware overhead: NACHOS relies on existing logic in dataflow architectures.

III. Baseline Accelerator Framework

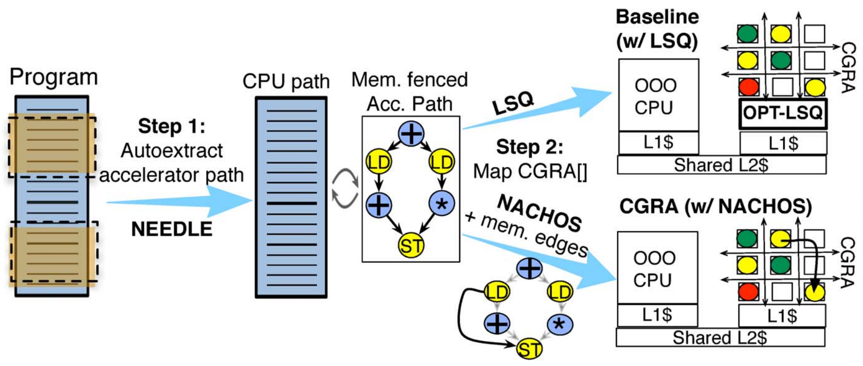
Here we provide a brief overview of the compiler and baseline architecture. Our focus is the memory disambiguation for accelerators and detailing the internals of the previously proposed spatial accelerators [8], [10], [27] is challenging within the page limits. We briefly summarize below.

Compiler Support: The accelerator compiler takes a program written in a high-level language like C/C++, partitions the application into two components, the x86 executable which executes on the CPU and a dataflow graph which maps onto the CGRA. First, we deploy NEEDLE [18], which is an accelerator independent compiler pass that auto-partitions the application, processes LLVM IR and constructs the offload path. More details in the appendix. We would like to emphasize the compiler builds on prior work; NACHOS is only an intermediate step in the compilation process for handling memory dependencies. Figure 3 provides an overview of the framework.

In Step 2, we inject NACHOS's compiler passes that transforms the offloaded dataflow graph. NACHOS analyzes the dataflow graph of the offload region and inserts the requisite memory dependency edges. The offload path itself is memory fenced to ensure ordering with respect to the OOO's path execution. The OOO and CGRA accelerator are loosely coupled, and all data to and from the CGRA are passed through the caches and memory. In the baseline system with an LSQ in Step 2: since the CGRA spatial dataflow architecture does not have an embedded ordering unlike instructions in an OOO, the compiler uses explicit IDs (8 bits; max of 256 memory operations) to specify the memory ordering (like in TRIPS [40]). Finally, we use a CGRA mapping pass [5], [7] that schedules the LLVM IR operations onto the grid of function units and configures the static network. We generate a hybrid executable which executes on a CPU and a configuration for the CGRA.

The compiler aggressively localizes data, reducing the need for memory disambiguation. Where possible the compiler allocates local scratchpad memory for data used only within the accelerated program region. Examples of data directly mapped to the scratchpad are global data accessed only by the accelerated region and local stack variables. Both OPT-LSQ and NACHOS only disambiguate non-local data.

Accelerator architecture and Simulation: The accelerator we deploy is a loosely coupled CGRA with 32×32 homogeneous functional units similar in design to Dyser [8]. The accelerator includes its own private cache and is cache coherent with the host CPU through the shared L2 cache. Page limit prevents us from providing background on how CGRAs work (the appendix includes a detailed example). Briefly, each functional unit in the 32×32 grid maps a single instruction from the dataflow graph of the offload path. The data dependencies between the operations are explicitly routed over a static mesh operand network. NACHOS explicitly adds dependencies between memory operations, and the CGRA enforces these dependencies using the same operand network. The operand network also routes values from the cache at the edge of the grid to the function units in the grid. As noted in Step 2(Figure 3) we rely on previously released software for mapping the LLVM IR of the offload path to the CGRA and configuring the operand network. We use a cycle-by-cycle timing model based on recently released CGRA simulator [38]; the OOO core is modeled in detail using macsim [17]. The memory hierarchy and the associated cache coherence is modeled in detail using Ruby [24]. To



LLVM	build flags: -O3, -unroll-count=8
Host Core Cache	2 GHz, 4-way OOO, 96 entry ROB. 32 entry LSQ. L1: 64K 4-way, 3 cycles. LLC : 4M shared 16 way, 25 cycles. MESI Directory. Memory 200 cycles.
Accelerator	
CGRA **Energy** [20]	32× 32 homogeneous units [5], [7] Network: 600 fJ/link. ALU : 500 fJ/INT, 1500 fJ/FP.MDE. May: 500 fJ/edge Must: 250 fJ /edge
LSQ (§6)	2 port 48 entries/bank. # banks 2—8. Loads:2500 fJ, Stores: 3500fJ

Figure 3: Accelerator framework. Step 1: NEEDLE [18] extracts accelerators. Step 2: Scheduler [7] maps dataflow graph of accelerator path to CGRA. Target 1: LSQ enforces order like TRIPS [40]). Target 2: CGRA (w/ NACHOS): Compiler adds memory dependencies.

model the power consumption, we adopt an event-based power model similar to Aladdin [37]. We implement an aggressive non-blocking interface to memory that buffers and routes values from the cache to the function units (like earlier work [27]). We evaluate three systems, OPT-LSQ – a baseline CGRA using a hardware LSQ (§VIII-C), NACHOS-SW — a CGRA in which the compiler conservatively enforces all memory dependencies (§ V), and NACHOS — which builds on NACHOS-SW and includes logic within the CGRA's function unit for runtime checking(§ VII). Both NACHOS and NACHOS-SW do not require LSQ.

IV. Accelerator Workload Characteristics

We select the hottest path (i.e., high % of dynamic instructions) with the largest number of memory operations for this study. The accelerator paths we study here were recently released at IISWC [19][1]. Table II describes the characteristics of the accelerated program paths. We use optimal LLVM flags and aggressively unroll (`-O3 -mem2reg -unroll-count=8`).

Observation 1: *In the accelerated code, the compiler can allocate a notable fraction of data (e.g., stack variables) to a local scratchpad and perfectly disambiguate such data.* Memory accesses in the acceleration path can be classified into local or non-local (heap and global) memory based on the compiler. Local accesses refer to memory locations which can be allocated statically (e.g., local variables in the path). The compiler [18], [43] can perfectly disambiguate local accesses and promotes these accesses to a scratchpad or local registers. Table II: C5 refers to the % of operations which got promoted to local registers. In 12 out of 28 applications more than 20% of operations got promoted to local registers. NACHOS's restrict scope only to non-local operations Table II: C4. For fairness, our baseline OPT-LSQ in § VIII-C, also elides local accesses.

Observation 2: *There is little conflict behavior on heap and global memory accesses (see Table II: C4)* .

The memory dependencies listed are for a specific dynamic run; we found this relation to be the same across several runs. Only 5 out of 27 workloads have Store-Load dependencies. In many workloads, a large % of LSQ checks are for independent operations. If a compiler can identify these operations, the memory operations that actually alias would require minimal hardware to enforce memory ordering; NACHOS exploits this observation.

Observation 3: *MLP and number of memory operations vary significantly across workloads.* Columns C1–C3 summarize the compute and memory characteristics. A few observations that highlight the challenge of designing a scalable memory disambiguation unit (1) accelerator regions can be compute heavy with few memory operations and do not require any memory disambiguation (e.g., blackscholes), (2) accelerator regions can be memory dominated (38% of operations are memory), but not much aliasing; its imperative memory disambiguation finds the MLP to ensure performance. (e.g., equake) (3) accelerator regions can be memory dominated and have a lot of aliasing which puts memory disambiguation on the critical path of loads (e.g., fft) and (4) There can be high variance in MLP (C3) and the number of memory operations across workloads, which requires the memory disambiguation to scale both up and down (9 apps: MLP>16, 7 apps: MLP <4)

A. Why is alias analysis suited to accelerators?

VLIW compilers have extensively used alias analysis for identifying independent memory operations [1], [6] and have noted the limitations of using alias analysis for a whole program. We find that alias analysis is effective in the offload accelerated path.

We changed the scope of alias analysis from offloaded accelerated paths to the parent function before path extraction. Then we observed the extra % of MAY Alias relations which were added due to the set of memory operations which were previously part of the offloaded region. These % MAY alias relationships are new dependencies added between memory operations which are selected for accelerated path extraction, and memory operations which are part of the parent function and not selected for accelerated paths.

[1]https://github.com/sfu-arch/pdws

Table II: Acceleration Region Characteristics

		C1	C2	C3	C4			C5
		Static			Mem. Aliases			%LOC
	App	#OPs	#Mem	MLP	St-St	St-Ld	Ld-St	Access
SPEC2k	gzip	64	4	4	.	.	.	21
	art	100	36	4	6	6	10	0
	181mcf	29	2	2	.	.	.	5
	equake[2]	559	215	16	.	.	12	2
	crafty	72	7	8	.	.	.	40
	parser	81	12	4	.	.	2	34
SPEC2k6	bzip2	501	110	128	3	.	3	27
	gcc	47	2	2	.	.	.	26
	429mcf	30	3	4	.	.	.	24
	namd	527	100	16	6	6	30	41
	soplex	140	32	4	.	.	8	19
	povray	223	74	32	4	21	24	95
	sjeng	99	11	8	.	.	.	33
	h264ref	224	42	8	.	.	5	27
	lbm	427	57	32	.	.	.	12
	sphinx3	133	20	32	.	.	.	0
PARSEC+Others	blacks.[1]	297	0	0	.	.	.	4
	bodytr.	285	42	4	30	30	42	10
	dwt53 .	106	16	16	.	.	.	11
	ferret.	185	0	2	.	.	.	29
	fft-2d.[3]	314	80	4	.	.	48	18
	fluida.	229	28	8	.	.	.	14
	freqmi.	109	32	4	.	.	8	17
	sar-back	151	7	8	.	.	.	64
	sar-pfa.	500	32	16	12	20	12	19
	stream.	210	32	16	.	.	.	0.5
	histog.	522	48	16	.	.	.	0

#MEM: Global memory operations that need disambiguation. #OP and #MEM: Static counts in the dataflow graph. **C4:** # of memory dependencies (Ld-St: Load-Store). **C5:** % of memory operations to scratchpad (disambiguate perfectly), not part of # MEM.

For 12 out of 27 benchmarks, the % of MAY aliases increased; 5 benchmarks had more than 10× increase in MAY aliases. Bzip2, soplex and povray had maximum increase; 380×, 85×, and 100×.

NACHOS makes the key observation that hardware accelerators restrict the window of execution anyway since they statically schedule the instructions in the dataflow graph. Hence we only need to apply alias analysis to path offloaded to the accelerator (the DFG in Figure 3); the program paths running on the OOO CPU continue to rely on an LSQ. Memory fences in the offload path ensure ordering between the CPU and accelerated paths; since the accelerated paths are coarse enough (see #OPs in Table II) The fences constitute minimal overhead. Coarse grain offload paths are created by NEEDLE [18] (Step 1 in Figure 3 by forming superblock (or trace) based on program profile. Offload path has no control flow which helps eliminate complex control flow and layers of indirection that often confound simple alias analysis techniques. The effectiveness of alias analysis techniques also depends on the specific algorithms and nature of the workloads. We discuss the details of the impact of various alias analysis in § V.

V. NACHOS-SW: Softare-only Memory Disambiguation

We describe NACHOS-SW and analyze when hardware assistance is needed. The input to NACHOS-SW is the dataflow graph of the offload path and output is an augmented dataflow graph which includes *memory dependency edges* (MDEs) that need to be enforced. The output dataflow graph, similar to prior work [8] is mapped to the function units of the CGRA accelerator that enforces ordering explicitly, similar to data dependencies. Figure 4 illustrates NACHOS-SW.

An age-ordered LSQ answers the following question: *For a memory operation X, which in-flight memory operations overlap with X? and what is the program order with respect to X.* NACHOS-SW answers the question: *Given two memory operations X and Y do they alias (overlap) and what is the program order amongst them.* NACHOS-SW performs the alias checks statically ahead-of-time and saves dynamic energy. Compiler analysis help to determine which operations on pointers may affect each other, these analyses provide a foundation [2], [14], [39]. Here we leverage alias analysis to enforce correct ordering for overlapping memory operations. We study in detail the accuracy of the compiler alias analysis and its impact on the performance and energy consumption of the hardware accelerator. Compiler alias analysis can emulate LSQ-based memory disambiguation by considering all memory operations pairwise in the program region. While this may seem daunting, it is performed on the static dataflow graph and furthermore is restricted to the accelerated region only; a memory fence orders the accelerator's memory operations with the CPU.

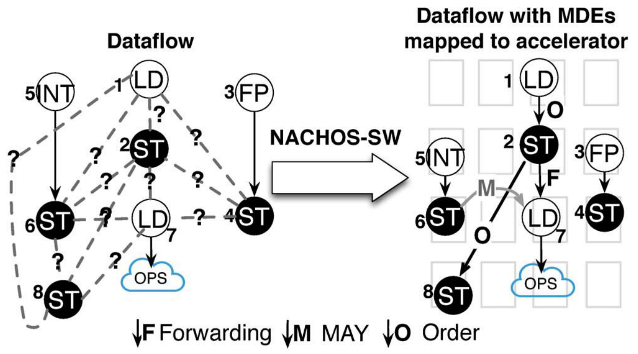


Figure 4: NACHOS-SW Memory Disambiguation. NACHOS-SW labels memory operations pairs as NO, MAY, and MUST. MUST labels require Forwarding or Order edges; NACHOS-SW treats MAY labels as MUST.

For every pair of memory operations in the acceleration region, the compiler assigns three types of labels: **NO**, **MUST** and **MAY** alias. For the **NO** label, the compiler does not add any MDE, allowing the memory operations to issue in parallel (e.g., 4 ST operation is allowed to proceed in parallel). For **MUST**, NACHOS may introduce one of two types of MDEs in the dataflow graph, *ORDER*(O) and *FORWARD*(F). The order dataflow edges **(O edges: 1bit)** are inserted between LD-ST and ST-ST memory operations (e.g., 2 → 8) that *must* alias. O edges are 1-bit ready signals which ensure that operations to the same memory location are executed in program order, i.e., the younger operation waits for the older operation to complete. The forward dataflow **(F edges: 64bit value)** is inserted between ST-LD memory (e.g., ❷→⑦) operations that must alias and forward values; essentially the memory dependency has been transformed into a data dependency. For each LD operation, we support forwarding from at most one ST. Since ST-LD dependencies are uncommon (see Table II) we handle the uncommon cases (e.g., partial overlap, LD depending on multiple STs) by enforcing the memory dependence as an ordering edge and stall the LD until the STs complete. Finally for **MAY** (i.e., compiler unsure if operations alias), NACHOS-SW conservatively treats it as **MUST** and enforces ordering.

A. Stage-wise refinement in NACHOS-SW

NACHOS-SW employs a stagewise approach to improve accuracy and prune the **MUST** and **MAY** labels and consequently the number of MDEs required for correctness compared to baseline compiler. Figure 5 summarizes the four stages of NACHOS analyses. The first stage modifies LLVM's alias analysis for memory disambiguation and assigns *MUST*, *MAY* or *NO* alias label to each pair of memory operations in the dataflow graph. The second stage uses inter-procedural information to further resolve *MAY* labels to *NO* labels. The third stage leverages existing data dependencies to trim redundant MUST and MAY labels. Finally, stage four employs polyhedral loop analysis [9] to detect independence in multidimensional array accesses. Overall results:

- Stage 1 proves no requirement for disambiguation for seven workloads; of the remaining 20, on average 18% of alias relations can be proved to be *MUST* or *NO*.
- Stage 2 further converts 11% of *MAY*→*NO* alias relations for 10 workloads. Of these, it is particularly effective for five workloads; 22%–80% converted.
- Stage 3 removes 40%–84% of redundant alias relations which do not need to be enforced.
- Stage 4 Polly is a loop and data-locality optimizer framework that primarily targets multidimensional array accesses. It resolved all the memory accesses accurately in the acceleration region of 5 benchmarks.
- Across all workloads, NACHOS-SW introduces memory dependencies between ≃ 25% of memory operation pairs.

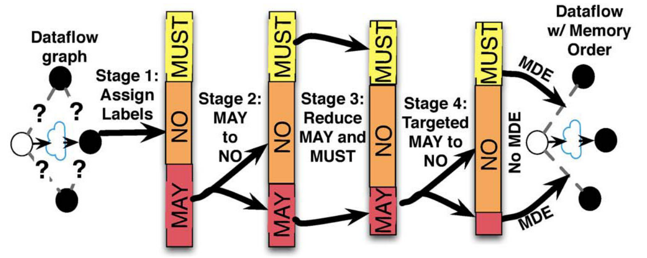


Figure 5: Stage wise pruning and refinement of alias relations into Memory Dependency Edges (MDEs). Refer Figure 4 for MAY Edge ($\xrightarrow{M}$) and MUST Edge ($\xrightarrow{O}$ or $\xrightarrow{F}$)

B. Stage 1: LLVM Alias Analysis. Assigning MAY, MUST and NO labels

NACHOS-SW analyzes memory operations pairwise and assigns a label to each pair of memory operations. Even Stage-1 in NACHOS-SW leverages all the advanced alias analysis in LLVM, Basic (Stateless checks, Base/Const Pointers), Types , Global variables, SCEV (pointer arithmetic and loops), ScopedNoAlias (variable scope), and CFL (data structures [45], [46]).

For each pair, three types of alias labels are possible *MUST*, *MAY*, and *NO*. The *MUST* and *NO*, which result from alias analysis provably identifies memory operations that are either in the same location or not respectively. The *MUST* label results in either an *ORDER* edge between ST-ST and LD-ST pairs or a *FORWARD* edge between ST-LD operations. Memory operation pairs with a *NO* alias relationship can execute in parallel. However, because alias analysis is undecidable, it can also give up and insert a *MAY* alias relation (i.e., compiler unsure). In most workloads, 19 of 27, the dominant alias type is *MAY*. Stage 1 alias analyses efficacy is limited in workloads where the accelerator regions are composed of complex program paths; this is evidence for the wide range of acceleration regions we study compared to prior work that targeted loops [25].

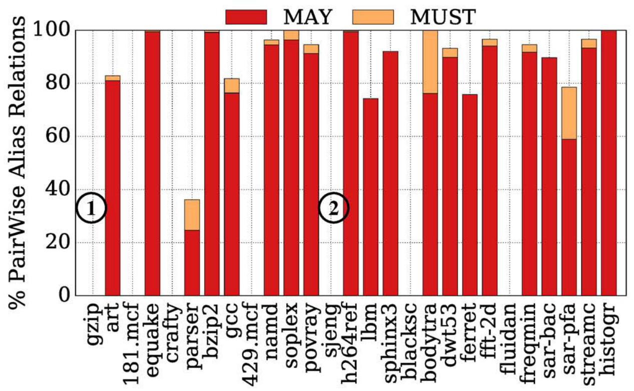


Figure 6: Stage 1: MAY and MUST alias relationships between memory operation pairs. Top 5 accelerated paths.

Figure 6 summarizes the results of Stage-1 for analyzing the top five top five frequently executed accelerator regions. Overall, 7 of 27 workloads need no further analysis. Some benchmarks (see ① in Figure 6) have no stores among the memory operations in these regions, i.e. gzip, mcf, crafty and blackscholes. sjeng has store and load operations ②, however alias relationships between all pairs are perfectly identified. On the remaining workloads, the stage 1 can classify on average 3% of pair wise checks as a *MUST* alias and 7% as *NO* alias relations.

C. Stage 2: MAY →NO with inter-procedural analyses

Note that the standard alias analyses presently in LLVM 3.8 cannot reason across function boundaries. While investigating the source code of our benchmarks, we observed that some of the pointers corresponding to the *MAY* label were derived from global or local variables whose addresses passed across function boundaries. These could be resolved to *NO* label by tracing the provenance of the pointers back across function boundary to the source global or local variable. We perform limited context sensitive analysis and the accelerator tend to be invoked from a single call site. Our workloads do not have function pointer invocation and it is tractable to trace the provenance of pointers across the core-accelerator boundary. This analysis takes as input the *MAY* alias relations from stage-1 and attempts to trace the data-dependence of the pointer back into the calling function to a source object. When two memory operations trace back to different objects, they are classified as *NO*.

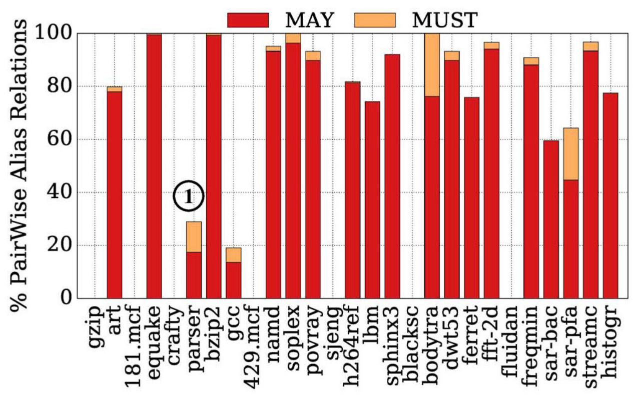


Figure 7: Stage 2 : Refinement of MAY alias from Stage 1 to NO. Top 5 paths.

Figure 7 presents the results of stage 2 applied to the *MAY* labels identified by stage 1. 10 workloads with *MAY* labels were refined by stage 2 of NACHOS-SW analyses. Where the inter-procedural analysis was effective, it converted 11% of *MAY* alias relations to *NO* alias relations. In parser ①, we find that stage 1 introduces *MAY* alias relations as it cannot reason about the equivalence of local pointers with a global pointer variable – `Table_connector ** table`. Stage 2 is able to convert 29% of *MAY* labels to *NO* in parser. Similarly, inter-procedural checks are particularly effective in gcc, sar-pfa-interp1, sar-backprojection and histogram. In these workloads, 20%–80% of *MAY* labels are converted to *NO*.

D. Stage 3: Removing redundant MAY and MUST

NACHOS-SW makes the key observation that not all of the alias relations identified by stage 1 and 2 need to be enforced in the dataflow graph. A memory dependency is redundant if a data dependencies already exist between the operations. Often we find that there already exists an implicit transitive data dependence between a pair of memory operations which enforces the required ordering. Removing redundant orderings is critical to NACHOS-SW as enforcing *ORDER*, *FORWARD* or *MAY* MDEs incur energy overhead. In figure 8 memory operations ①–❺ and ②–❻ are identified as aliasing. However the existing dataflow constraints (via ③) implicitly

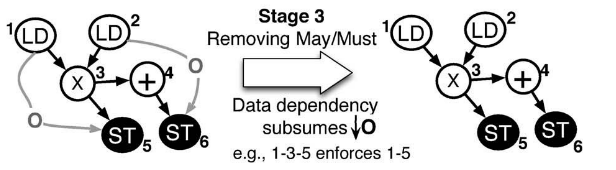


Figure 8: Implicit data dependencies eliminate the need to explicitly enforce ordering.

ensure that ① must complete before ⑤ begins, there is no reason for explicit memory ordering between them. Similarly, ② must complete before ❻ can execute due to ④. Stage-3 reduces the number of redundant ordering edges between memory operations, thus reducing overheads while program correctness is maintained.

In order to remove these redundant aliasing relations, stage-3 performs a reachability analysis between two memory operations in the dataflow graph. Since the accelerator dataflow graphs are directed and acyclic this is tractable. The offload region in the program is traversed in reverse topological order (post order traversal of the dataflow graph). For each alias relation check if the younger operation is reachable from the older memory operation in the dataflow graph. If it is reachable, then discard the alias relation as there exists an implicit data dependence. We do not eliminate St-Ld aliases even if they are redundant to ensure forwarding. If it is not, retain the MDE identified by Stage-1 or Stage-2. Additionally, all *MUST* alias relations are enforced prior to *MAY* alias relations.

Figure 9 shows the fraction of alias relations retained after simplification in stage 3 with respect to *all* alias relations determined in stage 1. Overall, we find that stage 3 is able to remove the need to enforce 68% of alias relations (*MUST* and *MAY*). Each bar is divided into the *MUST* alias and *MAY* alias relations which need to be enforced. The largest amount of redundant relations was in fft-2d, 84%.

E. Stage 4: Polyhedral Analyses and Multidimensional loops (MAY to NO)

Using standard alias analyses for 5 of the 27 workloads failed to provide meaningful alias information to minimize the addition of MDEs. Polly provided perfect information on MAY aliases within the stencil pattern loops in 5 applications and managed to successfully detect all the MAYs to be NO alias. We manually inspected the source of the accelerator region with the highest coverage to understand the reason for poor aliasing information. We find that for each of these workloads the standard alias analyses is confounded by multi-dimensional indexing into arrays. We list the code locations and the respective files: (equake, equake.c:1212), (lbm, lbm.c:175), (namd, ComputeNonbo.h:14), (bodytrack, ImageMeasure:108), (dwt53, dwt.c:179). The specific code example in equake would be: `w[col][0] += A[Anext][0][0]*v[i][0] + A[Anext][1][0]*v[i][1]....`

We leverage Polly, an LLVM project which uses a mathematical representation based on integer polyhedra to analyze and optimize memory access patterns [9]. The model is suitable for analysing the stencil based inner - loop patterns observed in the workloads where standard alias analyses fail. Polly disambiguated all memory operations in these workloads.

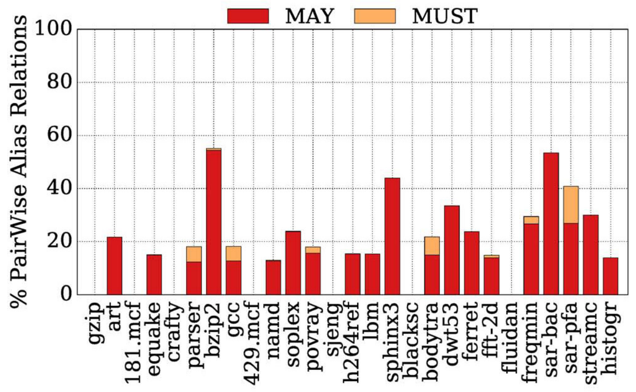


Figure 9: Stage 3 : Impact of simplification on alias dependencies for top five accelerated paths. Top 5 paths.

VI. NACHOS-SW vs OPT-LSQ Performance

Observation 1: *NACHOS-SW's performance depends on a number of independent memory operations and whether the compiler is able to find them. In 6 applications we see slowdown between 18%—100% ; runtime checks are needed*

Observation 2: *In some workloads, NACHOS-SW can perform better than OPT-LSQ since it reduces the load-to-use latency by issuing accesses out-of-order to the cache; 6 workloads 8%—62% improvement.*

OPT-LSQ disambiguate all memory operations at runtime through hardware checks and find as much MLP as available in the execution window, whereas NACHOS-SW serializes both MUST and MAY alias operations. However, with OPT-LSQ, while memory operations can execute and complete out-of-order they have to issue and allocate LSQ entries in program order. The LSQs are on the critical path and affects load-to-use latency (especially for cache hits). With NACHOS-SW when the compiler is able to find independent memory operations statically, they can issue in parallel.

Figure 10 shows the relationship between % of memory operations and % of MAY relations. We observe that the performance of NACHOS-SW depends on the ability of the compiler to find large % of non-conflicting memory operations (%NO and %MUST alias operations), and high % of memory operations in the offload path. Workloads which observe speedup or slowdown compared to OPT-LSQ (Figure 11), have high % of memory operations.

Figure 11 plots the % slowdown of NACHOS-SW normalized to OPT-LSQ. The Y axis is centered at 0; negative on Y axis means NACHOS-SW was faster; we explain below why. We found that for 21 out of 27 workloads,NACHOS-SW had under 4% performance overhead compared to OPT-LSQ.NACHOS-SW performed better than OPT-LSQ in 7 benchmarks. We illustrate with h264ref ①; 17% of the

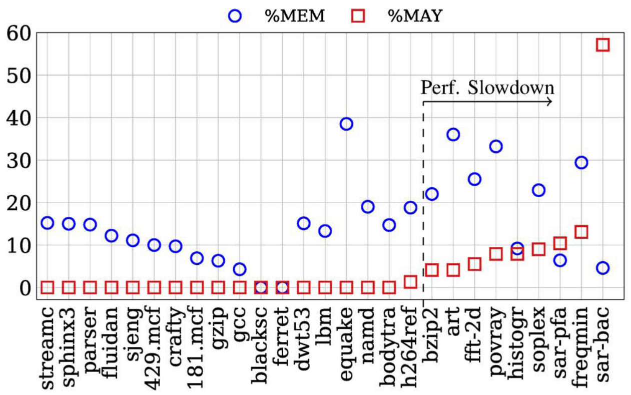


Figure 10: %MEM: % of memory operations. %MAY: Memory Ops with MAY label. X axis benchmark order based on %MAY (different from other plots in paper).

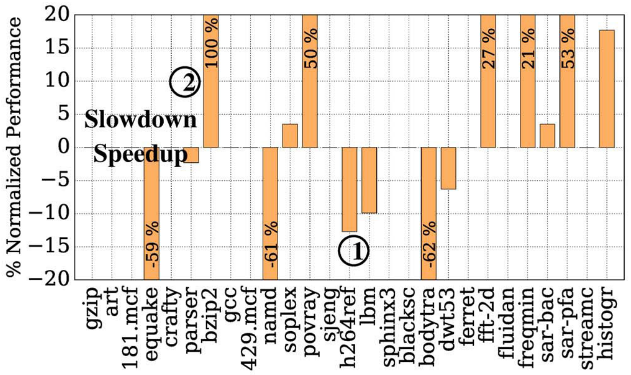


Figure 11: Performance of NACHOS-SW vs OPT-LSQ. Positive Y: % Slowdown. Negative Y: % Speedup

total operations are independent memory operations (MLP:8, 42 memory operations, 264 operations) (see Table II) many of which hit in the cache. Since the memory operations are issued in program order, LSQ is on the critical load path and increases load-to-use latency by two cycles; in NACHOS-SW there is no extra penalty; similar effects can be observed in equake (MLP:16, 215 memory operations, 559 operations) and namd. bodytrack and dwt53 have 0% of MAY alias relations, and have many MUST alias dependencies that the LSQ dynamically tracks and imposes a penalty to track them, while the NACHOS-SW statically identifies the dependencies (serializes them) and does not expend any cycles in tracking dependencies at runtime.

② For 6 benchmarks NACHOS-SW was slower compared to OPT-LSQ and varied from 18% to 100%. This is primarily caused by the prevalence of MAY alias cases and the number of memory operations in the critical path. In all these workloads (bzip2, art, fft, povray, histogram, soplex, sar-bac,sar-pfa and freqmine), the MAY edges serialize the memory operations which may be independent, and the performance is further affected by the presence of compute operations or floating point operations in the critical path of the dataflow accelerator. The compute operations dependent on these memory operations could also get serialized due to their parent memory operations being serialized. Povray's dataflow graph had a critical path of 95 operations, 42% of which were floating point many which were serialized due to 30 may aliases.

OPT-LSQ vs BASELINE COMPILER (Stage1 + Stage3): Observation 3: *The baseline compiler without NACHOS-SW improvements slows down significantly compared to OPT-LSQ due to serialization of MAY alias relationships; 10 applications slowdown more than 10% (max: 4×).* To understand the challenges for existing accelerator work, we studied the performance of the baseline LLVM compiler with only stage 3 optimizations and removed the alias analysis passes for Stage 2 and Stage 4 (Figure 12). In the absence of Stage 2 (inter procedural analysis), slowdown increased for three benchmarks namely, h264ref, sar-pfa-interp1, and histogram. Similarly, in the absence of Polyhedral Analyses, all five benchmarks (equake, namd,lbm, bodytrack, and dwt53) performed poorly. lbm (400 % slowdown) particularly perform poorly because a lot of MDEs are added to the dataflow graph which increases the critical path by 7.5×. When a memory operation in the critical path misses in the cache, the performance degrades accordingly (e.g., Bodytrack, 464.h264ref, and histogram).

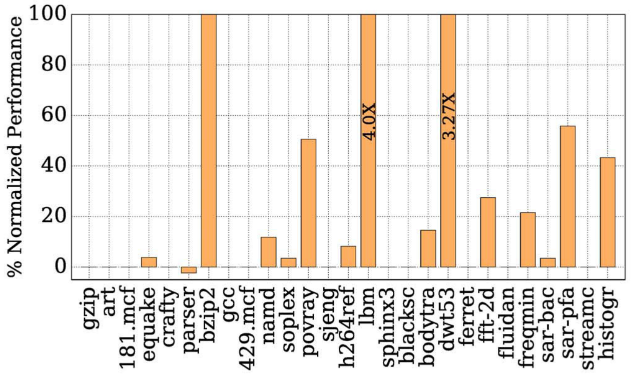


Figure 12: %Slowdown of Baseline Compiler (Stage1 + Stage3) normalized to OPT-LSQ (Lower is better)

VII. NACHOS: Hardware-assisted Runtime Disambiguation of MAY Aliases

NACHOS is hardware-assist for NACHOS-SW. The performance of NACHOS-SW depends on the % of MAY labels assigned by the compiler. Consider the dataflow graph in Figure 13, the Store operation ST ❷ conflicts with Load operation LD ⑤, and ST ❹ does not conflict with LD ⑤. However, the compiler is unsure of these relations (MAY alias) and adds a MAY edge for these relations. NACHOS-SW considers these MAY edges as MUST edge and serializes them. The LD ⑤ stalls for ❷ and ❹ which causes the cloud of compute operations dependent on the LD ⑤ to stall as well, which could have started execution as soon as ST ❹ finished execution.

Unlike NACHOS-SW which simply treats a MAY edge as a MUST edge, in NACHOS MAY edges disambiguate at runtime by the hardware to find independent memory operations. The operations ST ❷ and ST ❹ are checked in a round-robin manner against the LD ⑤ address, and the corresponding bit is set in the *Result* register if there is no conflict. In case there is a conflict, for example when ST ❷ and LD ⑤ addresses are compared, the corresponding bit is not set in the result. Once, the ST ❷ has finished execution it sets the corresponding bit in the Result to 1. Once, all bits are set in the Result, the LD ⑤ sends a request to the cache. When the response is available at LD ⑤, the state register is set to zero, LD ⑤ is marked as finished, and the cloud of operations can start execution.

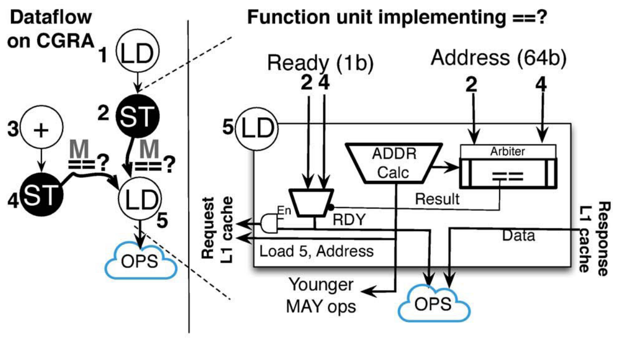


Figure 13: NACHOS hardware assist for runtime checking MAY aliases.

Why decentralized checking ?: We look at the trade-offs of centralized (LSQ) OPT-LSQ approach vs decentralized (comparator ==?) NACHOS approach. LSQs implement a 1-N CAM checks, where a single memory operation is compared against all other addresses in the CAM. Here the MAY alias relationships are pairwise, and it is possible to perform 1-1 check between two memory operations. For instance, consider LD ⑤ and ST ❹ in the Figure 13.

NACHOS comparator 1-1 checks could be slower than an LSQ if multiple parent operations are ready in the same cycle (i.e address already calculated). For example, if ST ❹ and ST ❷ are ready in the same cycle then the arbiter will make sure that only one comparator (==?) check is performed every cycle. Based on this delay at a level in the dataflow graph, the forward-slice of operations dependent on the memory operation will have to stall. This leads to domino effect where delay at a few memory operations will lead to delays accumulating at each level of the dataflow graph. Note, in the previous example if ST ❷ and ST ❹ got ready in different cycles then there would be no delay. Our simulator does model the ==? arbitration latency in detail for NACHOS and we optimistically assume a single cycle latency for OPT-LSQ. While LSQs impose a fixed energy penalty per memory operation, in NACHOS the energy overhead depends on the number of MAY aliases. Theoretically, if the number of MAY aliases are very high; the energy penalty of the ==? increases (Figure 13) compared to an OPT-LSQ; we do not observe this phenomenon in any of our workloads. See Appendix for equations

Fan-In of MAY aliases: Figure 14 presents a breaks down of *fan-ins* i.e the distribution of the number of parent memory operations that have a MAY alias relation with a memory operation. Overall, 9 workloads (e.g., gzip) have only independent memory operations (i.e., no older parents with a MAY alias relation); in 11 workloads (e.g., art), 50% of the memory operations have less than 1 older memory operation on which they may depend. A few workloads (e.g., bzip2, sar-pfa,fft-2d, soplex, povray, fft-2d) have memory operations with high fan-ins. In both bzip2 and sar-pfa this high fan-in increases the cycles spent in ==? (which includes only one comparator) and affects performance (more details in the next section).

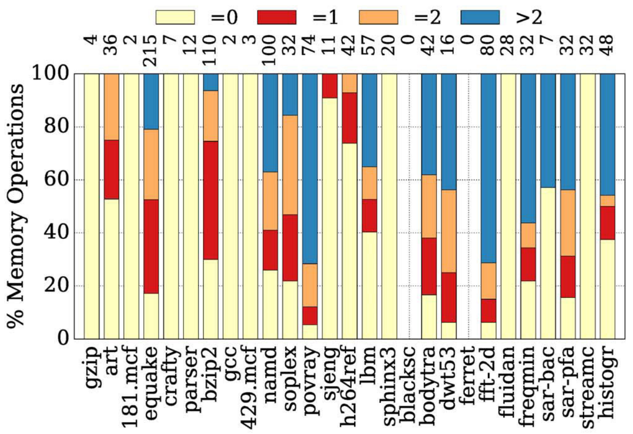


Figure 14: Breakdown of Older memory dependencies that MAY alias with a memory operation.

VIII. NACHOS vs OPT-LSQ

A. Performance

Observation 1: *For 19 benchmarks NACHOS was within 2.5% of OPT-LSQ. For six benchmarks performance improved between 6%—70% due to better load-to-use latency.*

NACHOS improves performance by verifying and parallelizing MAY alias memory operations that NACHOS-SW would have serialized. Figure 15 normalizes NACHOS's performance against OPT-LSQ. (1) NACHOS performed better than NACHOS-SW in 8 applications (464.h264ref, freqmine, histogram, bodytrack, fft-2d, sar-pfa-interp1, 453.povray, 401.bzip2). In h264ref and bodytrack, NACHOS-SW outperformed OPT-LSQ by minimizing the load-to-use penalty of cache hits; NACHOS outperformed NACHOS-SW further by 12%. (2) In workloads with a large number of MAY aliases (fft-2d, sar-pfa, povray and bzip2), NACHOS significantly outperformed NACHOS-SW (21—46% performance improvement); in both fft-2d and povray NACHOS achieves performance similar to OPT-LSQ.

NACHOS cannot mine more MLP than an OPT-LSQ and it primarily targets energy. For nineteen benchmarks it performs within 2.5% of OPT-LSQ, and improves performance up to 70% for six benchmarks. (3) In bzip2 and sar-pfa NACHOS experienced an 8% slowdown relative to OPT-LSQ. The slowdown is a result of the contention caused by fan-ins (of MAY alias) from many older parents at a few memory operations (see Section VII: Why decentralized checking?). In particular, bzip2 had three memory operations that potentially aliased with 50 older parents and with sar-pfa 43% of memory operations had >2 MAY alias parents (see Figure 14). These workloads also have high MLP (bzip2:128 ops sar-pfa: 16) and many memory operations fires simultaneously increasing contention at the ==? site of the younger memory operation. For 15 benchmarks the compiler is certain about all the dependencies (i.e., no MAY aliases); both NACHOS and NACHOS-SW achieves the same performance as OPT-LSQ (Figure 10 e.g., gzip). Finally, NACHOS improve over NACHOS-SW by detecting many more opportunities for ST-LD forwarding (bodytrack).

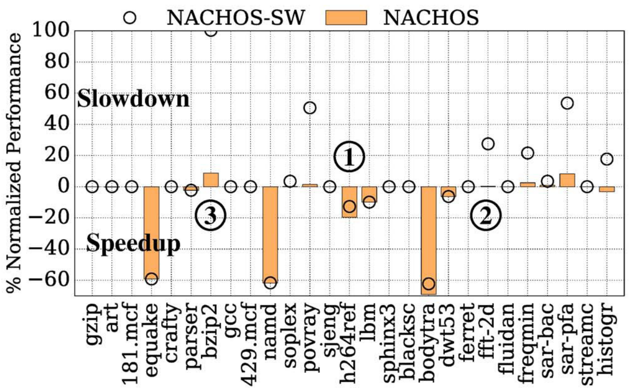


Figure 15: Performance: NACHOS vs OPT-LSQ. Marker indicates performance of NACHOS-SW.

B. Energy Efficiency

Observation 1: *NACHOS requires only 6% of total energy for memory ordering; for 15 workloads it imposes no overhead. OPT-LSQ requires 27% of total energy (includes the L1 cache).* leftmargin=*

- eliminating LSQ checks when *MUST* relations should be enforced and enforcing them using a pairwise ordering edge. NACHOS uses an *ORDER* edge (single bit) instead of hardware disambiguation.
- eliminating LSQ checks when *NO* aliasing exists and compiler proves memory operations can be run in parallel.
- finally, decomposing runtime disambiguation of *MAY* alias memory operations into pairwise checks instead of 1-to-many LSQ CAM checks.

Figure 16 the shows fraction of MDEs require by NACHOS's compared to the baseline compiler (see § VI :Baseline) in the selected acceleration region. The number on each bar represents absolute number of MDEs in work which correlates to the fraction of energy expended in NACHOS Figure 17.

In the workloads where MDEs were introduced, between 7–296 additional edges were added. This adds to MDE

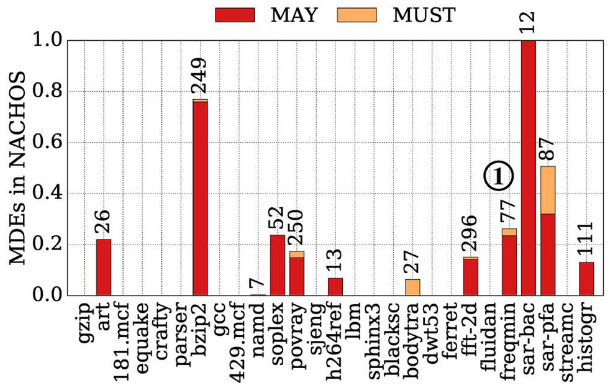


Figure 16: NACHOS vs Baseline compiler (Stage 1 and 3 only). Relative *times* of MDEs enforced (i.e., MAY + MUST); lower the better. Number on bar: # MDEs in NACHOS.

energy cost (Figure 17) i.e, comparator checks for MAY alias relations and serializing cost for MUST alias relations. Three workloads, povray, bzip2 and fft-2d required more than 250 MDEs which have the maximum % MDE energy cost in NACHOS. However, for fft-2d and povray, this represents enforcing less than 20% of MDEs by the baseline compiler. Overall, on an average 54 MDEs were added to workloads.

Figure 17 shows the energy breakdown of the NACHOS architecture. Cache energy is same for NACHOS and OPT-LSQ (see Figure 18). With NACHOS, the accelerator is 21% (12–40%) more energy efficient than OPT-LSQ. We elaborate:

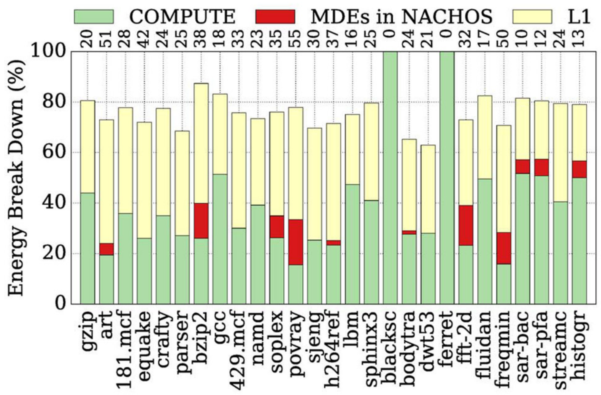


Figure 17: Reduction in NACHOS Energy vs OPT-LSQ. Number on bar: % memory operations.

sjeng (Efficacy of Stage 1). Related : gzip, mcf, crafty, mcf, and fft-2d. sjeng has 99 operations (11 memory operations) in the accelerated region. Of the 11 memory operations, only a single operation is a store. The stage 1 of NACHOS analyses is able to reason about the memory location of the store operation and deduce no alias relationships for all pairs of memory operations. NACHOS reduces energy consumption by 54.5% by enforcing exactly the dependencies which need to be enforced. Figure 6 shows the same trend true for not just the most frequently executed region, but also for the top five most frequently executed regions.

fluidanimate (Efficacy of Stage 2). Related: gcc, parser, h264ref, sar-backprojection, sar-pfa-interp1, and freqmine. We find a 26% reduction in energy as no MDEs are added to the dataflow graph for the most frequently executed region; 28 of 229 operations are memory operations. Stage 2 of NACHOS is able to reason about the objects in the parent context of the specialized region using inter-procedural alias analysis checks. An examination of the source `serial.cpp:40` shows the usage of global variables which are involved in pointer checks.

histogram (Efficacy of Stage 3) : Related : vpr and povray. In Stage 3 of NACHOS-SW, MAY edges need not be enforced due to the existing data dependence relationships in the dataflow graph, which reduces the number of dynamic checks required. This simplification pass removes 1293 of 1404 (93%) potential MDEs in the accelerated region across all workloads and adds an extra 22% MDEs to the original dataflow graph.

equake (Efficacy of Stage 4) : Related : lbm, namd, bodytrack and dwt53. Stage 4 provides perfect alias information for five benchmarks and thus does not incur any MDE energy cost.

C. OPT-LSQ: Baseline LSQ for accelerators

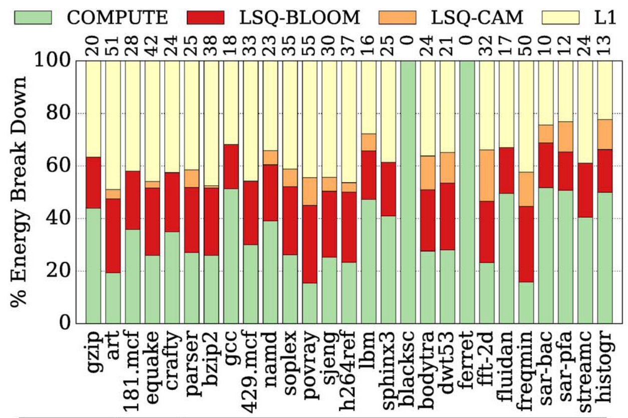
The OPT-LSQ is the baseline for all the performance and energy plots. The OPT-LSQ is an address partitioned [34] LSQ that minimizes energy penalty per LSQ check; it also includes a bloom filter [32]. Other optimizations were primarily designed for OOOs (e.g., [3], [26], [32], [35], [36], [41] and require complex prediction structures. See Figure 3 for OPT-LSQ parameters and access costs.

Challenge 1: Energy of OPT-LSQ

For an optimized LSQ design (i.e., Partitioned + Bloom filter), LSQs consumes 27% of total energy (including cache).

Figure 18 presents the energy breakdown of a CGRA accelerator with an LSQ. All accesses check the bloom filter; hits in bloom filter have to check the LSQ while misses do not. Nine benchmarks have perfect bloom filter behavior i.e., 0 hits and no CAM checks. Note that the bloom filter is strictly a best-effort energy optimization; a large LSQ is needed to handle all the in-flight memory operations. For 5 benchmarks (e.g., fft-2d), OPT-LSQ suffered in energy due to high % of bloom hits (20%+, see table in Figure 18) . In case of high bloom hits, the memory operations will have a higher penalty for OPT-LSQ compared to an LSQ without any bloom filter. For these benchmarks, the % of store operations are also high (25% to 50%) and they cause many bloom filter hits. The OPT-LSQ also expends high energy when forwarding values from stores to loads (e.g., bodytrack).

Challenge 2: Scaling with MLP and # Memory operations: *LSQs occupy a large amount of area relative to function units on the accelerator; a 48-entry LSQ occupies approximately the same area as 128 integer ALUs.* Provisioning



Bloom Hits(%)	Mean	Workloads
0	0	gzip, mcf, crafty, gcc, mcf, lbm, sphinx3, fluidanimate, streamcluster
0 – 10	2.5	art, bzip2, soplex, sjeng, h264ref
10 – 20	12.6	equake, parser, namd, povray, dwt53, sar-backprojection
20+	28	bodytrack, fft-2d, freqmine, sar-pfa-interp1, histogram

Figure 18: OPT-LSQ Dynamic Energy. Number on bar: % memory operations.

an LSQ for an accelerator takes away from valuable function unit resources. Determining size and ports is challenging since acceleration regions across our workloads tend to have varied memory behavior, both in the number of memory operations and MLP (see Table II). The number of entries in the LSQ predetermines MLP and dataflow partitioning [34], and the number of CAM ports determines overall instruction throughput. Overall, we find that 11 workloads have up to 20 memory operations but 6 workloads have 50+ memory operations. The MLP can also vary significantly (Table II:C3); 16 apps MLP<8 and 4 apps MLP>32 operations). Table II shows the number of memory operations in the acceleration region; can range from zero (ferret) to 215 (183.equake). This makes it challenging to find an optimal LSQ configuration for all workloads.

IX. Conclusion

We comprehensively investigate compiler-driven memory disambiguation for hardware accelerators. We will be releasing NACHOS-SW an LLVM based compiler that incorporates recent improvements in alias analysis, inter-procedural analysis and our own optimizations (redundancy remover) for minimizing compiler uncertainty. We propose NACHOS as a pay-as-you-go approach where the compiler filters out memory operations that need no checking, and the hardware dynamically checks (when necessary) to find MLP. NACHOS achieves performance comparable to the optimized LSQ and saves 21% energy compared to an optimized LSQ.

Appendix: Limits of decentralized checking

To understand the trade-offs between LSQ checks and NACHOS checks, we use a simple mathematical model. Consider, there are N memory operations and E_{lsq} is the joules per LSQ check. The total energy required for LSQ : $TOT_{lsq} = N \times E_{lsq}$. NACHOS enforces checks pairwise between memory operations; the total number of possible $\binom{N}{2}$ pairwise memory aliases can be classified into NO, MUST, and MAY.

$$\begin{aligned} TOT_{nachos} &= Pairs_{NO} \times E_{NO} + Pairs_{MUST} \times E_{MUST} \\ &\quad + Pairs_{MAY} \times E_{MAY} \\ \text{where } & Pairs_{NO} + Pairs_{MUST} + Pairs_{MAY} = \binom{N}{2} \end{aligned} \tag{1}$$

E_{NO} is $\simeq 0$ since no runtime operations are required for handling independent memory operations. Furthermore, $E_{MUST} << E_{MAY}$ since MUST edges require a single bit for ordering; while MAY edges need to pass entire address (and possibly value). Also workloads rarely have true memory dependencies; hence $Pairs_{MUST} << N$. The first two terms thus $\simeq 0$. The expression simplifies to $TOT_{nachos} = Pairs_{MAY} \times E_{MAY}$.

$$\frac{TOT_{nachos}}{TOT_{lsq}} = \frac{Pairs_{MAY} \times E_{MAY}}{N \times E_{lsq}} = \frac{Pairs_{MAY}}{N} \times \frac{E_{MAY}}{E_{lsq}} \tag{2}$$

Overall, $f_{en-savings} = \frac{E_{MAY}}{E_{lsq}} = \frac{500fJ}{3000fJ}$ is ratio of energy savings for a single MAY alias in NACHOS compared to a 1-to-N CAM check in lsq; we have conservatively assumed a single MAY alias comparator to be 500fJ and the optimized LSQ to be 3000fJ (only a 6× difference). Decentralized checking is profitable if the avg. number of may aliases per memory operation ($\frac{Pairs_{MAY}}{N}$) is less than $\frac{1}{f_{en-savings}}$(6 here). Only in seven benchmarks is the ($\frac{Pairs_{MAY}}{N}$) ¿ 1 (i.e., bzip2, soplex, povray, fft, freqmine, sar, and histogram). The energy for MDEs in NACHOS as a fraction of total energy is higher in these workloads (see Figure 17); but certainly less than the OPT-LSQ since $\frac{Pairs_{MAY}}{N} < 6$.

References

[1] D. I. August, D. A. Connors, S. A. Mahlke, J. W. Sias, K. M. Crozier, B.-C. Cheng, P. R. Eaton, Q. B. Olaniran, and W.-m. W. Hwu, "Integrated predicated and speculative execution in the IMPACT EPIC architecture," in *ISCA*, 1998.

[2] D. R. Chase, M. Wegman, and F. K. Zadeck, "Analysis of pointers and structures," in *PLDI*, 1990.

[3] G. Z. Chrysos and J. S. Emer, "Memory dependence prediction using store sets," in *ISCA*, 1998.

[4] N. Clark, M. Kudlur, H. Park, S. Mahlke, and K. Flautner, "Application-specific processing on a general-purpose core via transparent instruction set customization," in *MICRO*, 2004.

[5] A. S. U. Compiler Microarchitecture Lab, "Gem5 for software managed reconfigurable accelerator (smra)." [Online]. Available: https://github.com/cmlasu/cml-cgra

[6] D. M. Gallagher, W. Y. Chen, S. A. Mahlke, J. C. Gyllenhaal, and W. mei W. Hwu, "Dynamic memory disambiguation using the memory conflict buffer," in *ASPLOS*, 1994.

[7] V. Govindaraju, "Dyser: Dynamically specialized datapaths for energy efficient computing." [Online]. Available: http://research.cs.wisc.edu/vertical/dyser-compiler/

[8] V. Govindaraju, C.-H. Ho, and K. Sankaralingam, "Dynamically specialized datapaths for energy efficient computing," in

HPCA, 2011.
[9] T. Grosser, A. Groesslinger, and C. Lengauer, “Polly - performing polyhedral optimizations on a low-level intermediate representation.”
[10] S. Gupta, S. Feng, A. Ansari, S. A. Mahlke, and D. I. August, “Bundled execution of recurring traces for energy-efficient general purpose processing,” in *MICRO*, 2011.
[11] T. J. Ham, J. L. Aragn, and M. Martonosi, “DeSC: decoupled supply-compute communication management for heterogeneous architectures.” in *MICRO*, 2015.
[12] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, “EIE: Efficient Inference Engine on Compressed Deep Neural Network.” in *ISCA*, 2016.
[13] M. Hayenga, V. R. K. Naresh, and M. H. Lipasti, “Revolver: Processor architecture for power efficient loop execution,” in *HPCA*, 2014.
[14] M. Hind, “Pointer analysis: haven’t we solved this problem yet?” in *PASTE*, 2001.
[15] C.-H. Ho, S. J. Kim, and K. Sankaralingam, “Efficient execution of memory access phases using dataflow specialization.” in *ISCA*, 2015.
[16] E. Ipek, M. Kirman, N. Kirman, and J. F. Martinez, “Core fusion: accommodating software diversity in chip multiprocessors,” in *ISCA*, 2007.
[17] H. Kim, J. Lee, N. B. Lakshminarayana, J. Sim, J. Lim, and T. Pho, “Macsim: A cpu-gpu heterogeneous simulation framework user guide.”
[18] S. Kumar, N. Sumner, S. Magrem, V. Srinivasam, , and A. Shriraman, “Needle : Leveraging program analysis to analyze and extract accelerators from whole programs,” in *HPCA*, 2017.
[19] S. Kumar, N. Sumner, and A. Shriraman, “Spec-ax and parsec-ax: Extracting accelerator benchmarks from microprocessor benchmarks,” in *IISWC*, 2016.
[20] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, “McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures,” in *MICRO*, 2009.
[21] J. Q. Lin, T. Chen, W.-C. Hsu, and P.-C. Yew, “Speculative register promotion using advanced load address table (alat),” in *CGO*, 2003.
[22] D. Lustig, M. Pellauer, and M. Martonosi, “Pipe check: Specifying and verifying microarchitectural enforcement of memory consistency models,” in *MICRO*, 2014.
[23] D. Mahajan, J. Park, E. Amaro, H. Sharma, A. Yazdanbakhsh, J. K. Kim, and H. Esmaeilzadeh, “TABLA: A unified template-based framework for accelerating statistical machine learning.” in *HPCA*, 2016.
[24] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, “Multifacet’s general execution-driven multiprocessor simulator (gems) toolset,” *SIGARCH Computer Architecture News*, 2005.
[25] M. Mishra, T. J. Callahan, T. Chelcea, G. Venkataramani, S. C. Goldstein, and M. Budiu, “Tartan: evaluating spatial computation for whole program execution,” in *ASPLOS*, 2006.
[26] A. Moshovos and G. S. Sohi, “Speculative Memory Cloaking and Bypassing.” *International Journal of Parallel Programming*, 1999.
[27] T. Nowatzki, V. Gangadhar, and K. Sankaralingam, “Exploring the potential of heterogeneous von neumann/dataflow execution models,” *ISCA*, 2015.
[28] S. Padmanabha, A. Lukefahr, R. Das, and S. A. Mahlke, “Trace based phase prediction for tightly-coupled heterogeneous cores.” in *MICRO*, 2013.
[29] H. Park, Y. Park, and S. Mahlke, “Polymorphic pipeline array: a flexible multicore accelerator with virtualized execution for mobile multimedia applications,” in *MICRO*, 2009.
[30] M. A. Pericàs, A. Cristal, F. J. Cazorla, R. González, A. V. Veidenbaum, D. A. Jiménez, and M. Valero, “A two-level load/store queue based on execution locality,” *ISCA*, 2008.
[31] M. Seth and S. C. Goldstein, “Optimizing memory accesses for spatial computation,” in *CGO*, 2003.
[32] S. Sethumadhavan, R. Desikan, D. Burger, C. R. Moore, and S. W. Keckler, “Scalable Hardware Memory Disambiguation for High ILP Processors,” in *MICRO*, 2003.
[33] S. Sethumadhavan, R. G. McDonald, R. Desikan, D. Burger, and S. W. Keckler, “Design and implementation of the trips primary memory system,” *2006 International Conference on Computer Design*, 2006.
[34] S. Sethumadhavan, F. Roesner, J. S. Emer, D. Burger, and S. W. Keckler, “Late-binding: enabling unordered load-store queues,” in *ISCA*, 2007.
[35] T. Sha, M. M. K. Martin, and A. Roth, “Scalable Store-Load Forwarding via Store Queue Index Prediction,” in *MICRO*, 2005.
[36] T. Sha, M. M. Martin, and A. Roth, “Nosq: Store-load communication without a store queue,” in *MICRO*, 2006.
[37] Y. S. Shao, B. Reagen, G.-Y. Wei, and D. M. Brooks, “Aladdin: A pre-RTL, power-performance accelerator simulator enabling large design space exploration of customized architectures.” in *ISCA*, 2014.
[38] A. Sharifian, S. Kumar, A. Guha, and A. Shriraman, “Chainsaw: Von-neumann accelerators to leverage fused instruction chains,” in *MICRO*, 2016.
[39] Y. Smaragdakis and G. Balatsouras, “Pointer analysis,” *Found. Trends Program. Lang.*, 2015.
[40] A. Smith, J. Gibson, B. A. Maher, N. Nethercote, B. Yoder, D. Burger, K. S. McKinley, and J. H. Burrill, “Compiling for EDGE Architectures.” *CGO*, 2006.
[41] S. Subramaniam and G. H. Loh, “Fire-and-Forget: Load/Store Scheduling with No Store Queue at All,” in *MICRO*, 2006.
[42] K. Tran, T. E. Carlson, K. Koukos, M. Själander, V. Spiliopoulos, S. Kaxiras, and A. Jimborean, “Clairvoyance: look-ahead compile-time scheduling,” in *CGO*, 2017.
[43] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor, “Conservation cores: reducing the energy of mature computations,” in *ASPLOS*, 2010.
[44] L. Wu, R. J. Barker, M. A. Kim, and K. A. Ross, “Navigating big data with high-throughput, energy-efficient data partitioning,” in *ISCA*, 2013.
[45] Q. Zhang, M. R. Lyu, H. Yuan, and Z. Su, “Fast algorithms for dyck-cfl-reachability with applications to alias analysis,” in *PLDI*, 2013.
[46] X. Zheng and R. Rugina, “Demand-driven alias analysis for c,” in *POPL*, 2008.