Leveraging Compiler Alias Analysis To Free Accelerators from Load-Store Queues

by

Naveen Vedula

B.Tech, National Institute of Technology Warangal, India 2010

Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of
Master of Science

in the
School of Computing Science
Faculty of Applied Sciences

SIMON FRASER UNIVERSITY
Fall 2016

Approval

Name:	**Naveen Vedula**
Degree:	**Master of Science (Computing Science)**
Title:	***Leveraging Compiler Alias Analysis To Free Accelerators from Load-Store Queues***
Examining Committee:	**Chair:** Ryan Shea Research Assosiate

Arrvindh Shriraman
Senior Supervisor
Associate Professor _____

William N. Sumner
Supervisor
Assistant Professor _____

Jiangchuan Liu
Internal Examiner
Professor
School of Computing Science
Simon Fraser University _____

Date Defended: December 6, 2016

Abstract

Hardware accelerators are an energy efficient alternative to general purpose processors for specific program regions. They have relied on the compiler to extract instruction level parallelism but may waste significant energy in memory disambiguation and discovering memory level parallelism (MLP). Currently, accelerators either i) Define the problem away, and rely on massively parallel programming models [1, 48] to extract MLP. ii) Reuse the Out of Order (OoO) processor [7, 28], and rely on power hungry load-store queues (LSQs) for memory disambiguation, or iii) Serialize – some accelerators [47] focus on program regions where MLP is not important and simply serialize memory operations.

We present *NACHOS*, a compiler assisted energy efficient approach to memory disambiguation, which completely eliminates the need for an LSQ. *NACHOS* classifies memory operations pairwise into those that *don't alias* (i.e., independent memory operations), *must alias* (i.e., ordering is required between memory operations), and *may alias* (i.e., compiler is unsure). To enforce program order between must alias memory operations, the compiler inserts ordering edges that are enforced as def-use data dependencies. When the compiler is unsure (i.e., *may alias*) about a pair of memory operations, the hardware checks if they are independent. We demonstrate that compiler alias analysis with additional refinement can achieve high accuracy for hardware accelerated regions.

In our workload suite comprising of SPEC2k, SPEC2k6, and PARSEC workloads; Across 15 applications *NACHOS* imposes no energy overhead over the function units (i.e., compiler resolves all dependencies), and in another 12 applications *NACHOS* consumes $\simeq$17% of function unit energy (max: 53% in povray). Overall *NACHOS* achieves performance similar to an optimized LSQ and adds an overhead equal to 2.3$\times$ of compute energy.

Keywords: Dataflow architectures; Alias analysis; Accelerators; Load Store queues;

Dedication

To mom and dad.

Acknowledgements

I would like to thank Dr. Arrvindh Shriraman for constantly pushing me and guiding me throughout my journey, Dr. William N. Sumner for being a source of motivation and an inspirational figure, Snehasish Kumar for all the help and advice in our projects together, and finally my friends and labmates for all the support.

Table of Contents

List of Tables

List of Figures

List of Acronyms

ALU Arithmetic Logic Unit

CGRA Coarse Grained Reconfigurable Architectures

CAM Content Addressable Memory

DFG Data Flow Graph

GPU Graphic Processing Unit

ILP Instruction Level Parallelism

KMH Knuth's Multiplicative Hash

LLC Last Level Cache

LD Load

LSQ Load Store Queue

MDE Memory Dependency Edge

MLP Memory Level Parallelism

OoO Out of Order

RAM Random Access Memory

ROB Reorder Buffer

ST Store

SRAM Static Random Access Memory

TLP Thread Level Parallelism

Chapter 1

Introduction

Memory disambiguation is required to detect and parallelize memory operations accessing different memory locations and enforce program order between memory operations to the same location. Existing accelerator studies have focused extensively on compute specialization and have left memory disambiguation unexplored. In many cases, hardware accelerators target conventional algorithms or specific loop patterns, and the hardware designer has manually disambiguated the memory locations [23, 49]. Interestingly, much of the energy efficiency gain in these accelerators comes from recognizing nearby store-load dependencies and localizing the communication [10]. Systems such as Altera's FPGA-OpenCL [1] use a massively parallel programming model and rely on thread level parallelism (TLP) to extract memory level parallelism (MLP); hence they serialize accesses from each thread (like a Graphic Processing Unit). Memory disambiguation is a critical challenge for hardware accelerators that target a broad range of program behavior (e.g., Coarse-Grained Reconfigurable Architectures (CGRAs) [31], Dyser [7], BSA [29], Compound function units [9], Big-Little [30]) and is required to ensure utility within existing applications.

Many hardware accelerators [7, 28, 30] repurpose the Out of Order processor's (OoO) load-store queue (LSQ) for memory disambiguation. Figure 1.1 highlights the trade-offs in LSQs. In a Dyser-like CGRA accelerator, we find that the OoO's LSQ dominates overall energy consumption (since other overheads are minimal); the LSQ energy is $\simeq$2.6$\times$ the function unit energy. It is also unclear if the LSQ ports and entries can be scaled to match the MLP available in hardware accelerators. Research on LSQs have focused on hierarchical designs to filter accesses from the LSQ and reduce energy consumption [2, 36–39, 44]. We find that even highly optimized LSQs (Tiered [36] and Partitioned [37]) impose overhead of 2.3$\times$. In Chapter 4.1.1, we analyze the integration of LSQs with the accelerator. Another challenge with LSQs is area; distributed processors (e.g., TRIPS [34], CoreFusion [16]) investigated partitioned LSQ; unfortunately in TRIPS [34] each had to be worst case sized and occupied area comparable to an 8KB L1 cache bank; a 48 entry LSQ would be equivalent to 50 integer ALUs.

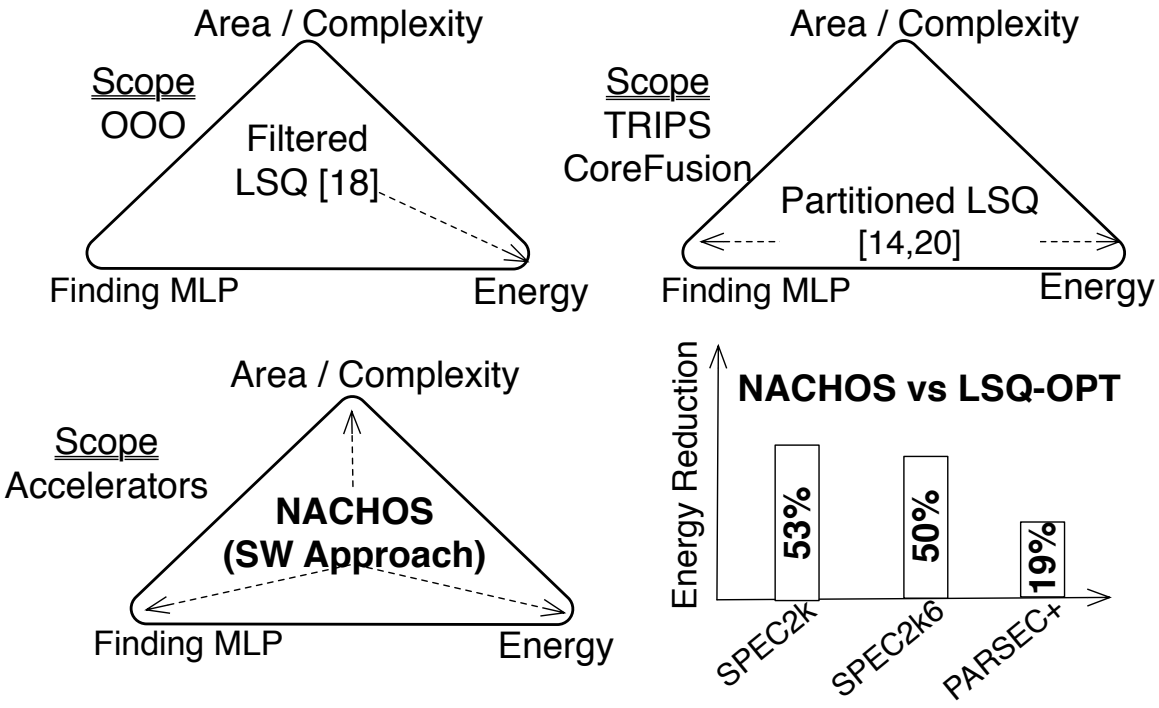


Figure 1.1: Optimized LSQ designs vs *NACHOS*. Arrows indicate the parameters that are targeted and improved.

Our Approach: We propose *NACHOS*, a compiler-assisted approach to memory disambiguation for hardware accelerators. We eliminate the LSQ by restricting the scope of alias analysis to the offloaded acceleration region and improving the overall quality of the alias analysis. *NACHOS* is an LLVM-based prototype compiler that analyzes memory operations in the accelerated region to find independent memory operations– to parallelize, and memory dependencies– to enforce program order.

NACHOS's compiler enforces ordering between aliasing memory operations with a def-use edge, a memory dependency edge (MDE) between them. Hardware accelerator enforces MDE ordering, similar to ordering instruction data dependencies. In cases when operations don't alias, the compiler inserts no dependency, and the memory operations can proceed in parallel. In cases when the compiler is unsure, it adds a *may alias* dependency edge between two memory operations. *NACHOS* compares the addresses in hardware (part of the dataflow fabric) to determine if the two operations can run in parallel. *NACHOS* enforces ordering between operations pairwise, instead of LSQ's centralized approach; Chapter 3 provides an overview and Chapter 5 elaborates on the overall design.

NACHOS is more scalable and efficient than an LSQ. The compiler has access to more contextual information related to objects, stack vs. heap, and memory allocation, enabling it to achieve good accuracy. We describe stages of alias analysis in Chapter 5.2. It incorporates standard, advanced alias analysis and our extensions and refinement techniques. Accuracy improves since *NACHOS*

only targets a fixed window of instructions offloaded to the accelerator as opposed to the entire program [18].

We have analyzed a total of 135 accelerator regions across 27 complete workloads from 4 benchmark suites (PARSEC, PERFECT, SPEC2000, SPEC2006) and studied the potential for *NACHOS* memory disambiguation. Our software approach accrues no energy overhead for memory disambiguation in 15 of the 27 workloads. In these workloads, the compiler was able to accurately identify all the memory orderings required and uncover the MLP available. Compared to an optimized LSQ, *NACHOS* saves 53% of energy for SPEC2K, 50% of energy for SPEC2K6, and 19% of energy for PARSEC and other workloads. We achieve performance comparable to the LSQ by not enforcing any more dependencies than an LSQ and show that runtime checks in hardware are necessary when the compiler is unsure of the aliasing relationship.

Chapter 2

Background

2.1 Architecture

Processors: Memory disambiguation is an important function of the LSQ. The LSQ typically provides the following four functions: 1) buffering store addresses and values for in-order retirement (ST-ST ordering), 2) forwarding in-flight store values to loads (ST-LD forwarding), 3) detection of load and store ordering violations (LD-ST ordering), and 4) detection of memory consistency violations. Most LSQ designs use a pair of age-ordered queues – content addressable memories (CAMs). One each for loads and stores that can be associatively searched by the memory address. Loads search the store queue while stores search both the queues. CAMS are expensive regarding energy, to reduce the energy overhead of searching through these CAMs, filtering techniques are used to filter independent memory addresses and store them in additional buffers (Random Access Memory - RAM). Traditional LSQs are age ordered and thus need to keep an entry in the LSQ from the decode stage, before the address is even known, until the commit of the instruction. This leads to unnecessary high occupancy time and a number of entries. Late Binding allows unordered execution on age and allocates entries during issue time. Filtering addresses which are independent is a widely used strategy for reducing the number of LSQ accesses. The main focus of these works is to reduce or eliminate the CAM searches in LSQs and by filtering out accesses that do not require checks. Note that filters only minimize accesses to the LSQs to save energy; an LSQ is still needed to handle misses in the filter. Filtering addresses however, requires extensive support structures for filtering, prediction, and data forwarding and add additional complexity and area. Sha et al. [38] have proposed to employ memory dependence prediction. They pairwise match up potentially aliasing loads and stores to eliminate the power-hungry searches. In this proposal, the associative CAMs of LSQs are replaced with large multi-ported RAM structures (10s of KB). Hardware accelerators have minimal complexity (a 64 function unit fabric is approximately the size of the L1 cache); it is unclear whether it would be feasible to incorporate an area-hungry LSQ. Fire-and-Forget [44] and NoSQ [39] both proposed methods for eliminating the store queue by forwarding values to the loads through the register file or load queue respectively. Both proposals use sophisticated dependence predictors (store

sets [4]) that require additional area. Pericas et al. proposed the use of two level LSQs [2] in which an L0 or L1 LSQ filters accesses. The challenge with the filtering approach is that it does not reduce the size of the original LSQ. All active loads and stores still need to have an entry in the LSQ to handle the cases when the filter misses. Prior work has proposed a variety of filters including bloom filters [37] and even two-level LSQs [2]. One approach to reduce LSQ energy is an address-based banking [37, 43]. This optimizes for latency and power, but not area since the total capacity across all the banks must match the capacity of a centralized LSQ. Furthermore, additional flow control is required to handle cases when a bank overflows. Perhaps the most closely related approach to our work is from Huang and Huang [14]. They use binary instrumentation to filter out loads from the LSQ that were guaranteed to be safe. They only save energy and continue to require the LSQ, since the binary instrumentation is best-effort. Furthermore, their technique seeks to estimate which loads are safe but does not provide a mechanism to enforce ordering amongst unsafe operations.

Accelerators: Accelerator architecture research has focused extensively on compute specialization. In most cases, they rely on the LSQ of the host processor to enforce the appropriate semantics and interface with memory. Dataflow accelerators like DySER [7], leverage the Host core to take care of memory operations for accelerators. The host core fills the scratchpad of accelerators with data, and then execution starts. Whenever the accelerator hits a memory operation, it has to stall and notify the host core. There is a high overhead of context switch, and the accelerator cannot proceed without the data being present in the scratchpad, which limits the acceleratable program regions to a class of applications with access and execute patterns. In the presence of irregular memory operations or limited compute, architectures like DySER is not feasible.

Dynaspam [21] is also tightly integrated with OoO processor and uses reorder buffer (ROB) entries of the OoO processor to determine the age of memory operations. It aggressively does memory dependence prediction (like store sets) and does rollback in case of memory order violations for re-execution. The memory order violations are detected by OoO pipeline at retirement stage from ROB. It deals with similar problems as DySER. Architectures like DySER and Dynaspam have high area overhead, since they can only afford one accelerator per OoO core. Given a fixed area budget and data parallel applications, it is desirable to have more accelerators per chip, but that is not possible with high area overhead architecture designs.

Architectures like SEED [28] are loosely coupled with OoO core and provide a low-overhead interface to switch between the host core and the accelerator. SEED can calculate addresses for memory operations but depends on the store buffer of the host core for memory disambiguation. Thus, the overhead for switching between OoO and accelerator is low, and SEED does not need to depend on OoO to fetch data. However, Store buffers are designed based on the number of memory operations in an instruction window of an OoO core. On the other hand, dataflow accelerators can have a wide range of MLP, from as low as a few operations to greater than the instruction window size of an OoO core, and it depends on the program behavior. If OoO and the accelerator are running at the same time, adding memory addresses from the accelerator region to the OoO store buffers will

increase contention. This situation gets worse from the fact that addresses need to keep an entry in store buffers, at the decode stage of the processor until the commit of the corresponding instruction. Power gating either the OoO processor or the SEED is the chosen approach to tackle the above problem. However, various energy hungry structures of OoO still needs to be active. Note that there are LSQ designs like Late Binding [37] which do not create an entry in the LSQ until the issue stage. However, even in the case of Late Binding the LSQ size has to be designed for the worst case i.e. equal to all memory operations in the dataflow graph like the TRIPS architecture [34].

SGMF [48] maps a compute kernel represented as a dataflow graph on to a Coarse-grained Reconfigurable Fabric (CGRA). SGMF is composed of a grid of interconnected functional units and allow streaming data of multiple threads through these units. It runs several LSQ units, which use a CAM structure to select the thread running on the LSQ unit. It can issue only one memory operation per thread per LSQ unit, and also due to the CAM structure it cannot run many threads in parallel like Graphic Processing Units (GPUs). SGMF exploits thread level parallelism (TLP) but does not exploit available MLP per thread.

In Wavescalar [45], all memory instructions are statically identified by two Ids: 1) the sequence number of the instruction within the wave (trace) and, 2) a wave number indicating the wave (trace) invocation. All issued memory instructions from the fabric are reassembled in the memory system and executed in total load store order. Similarly, Conservation Cores [47] issue memory operations sequentially. These architectures trade-off low energy with performance. Again, architectures like wave scalar exploit the available TLP, but both Wavescalar and Conservation Cores fail to exploit the available MLP in data parallel programs.

To exploit MLP in workloads with defined memory access regions and execute regions, a specialized data fetch engine like MAD [13] is used in lock step with accelerators to prefetch data to the accelerator data storage unit. MAD prefetches data to the storage region of an accelerator, and signals to the accelerator to start execution. MAD aids architectures like DySER, to improve performance, but it still has to deal with the overhead of context switches.

There are classes of applications, which have frequently executed program regions, with irregular memory accesses and sometimes lots of available MLP. It is challenging to realize such applications in accelerators. This brings us to think about how the decades of research in memory interfaces for an OoO core is useful in the context of accelerators. A good start is to understand the challenges in designing memory interfaces for accelerators, and how they are different from OoO memory interfaces. Then, use this information to augment existing designs to fit the context of accelerators.

2.2 Alias Analysis in Compilers

The relationships between pointers in a program have been extensively examined in compiler and software engineering research. Static *alias analysis* determines whether two different pointers in a program may point to the same object when a program executes. In contrast, *pointer analysis* or *points-to* analysis determines the set of objects to which a pointer may point. By helping to

determine which operations on pointers may affect each other, these analysis provide a foundation for many other optimizations and program analysis [3, 41]. Computing perfectly precise alias analysis is impossible/undecidable in general, so substantial work has explored the trade-offs between efficiency and precision in alias analysis to derive pragmatic solutions [12].

Identifying pointers that *must not alias* is key to pruning potential conflicts between memory operations using those pointers. This same notion of pruning conflicting accesses via must-not-alias information has also been used for precise data race detection by Naik and Aiken [27]. While prior work uses must-not-alias analysis to detect bugs in software, we instead use the absence of conflicting accesses to improve efficiency when accelerating a target region of code. Must-not-alias analysis has also been used to improve the general efficiency of CFL-based alias analysis [50].

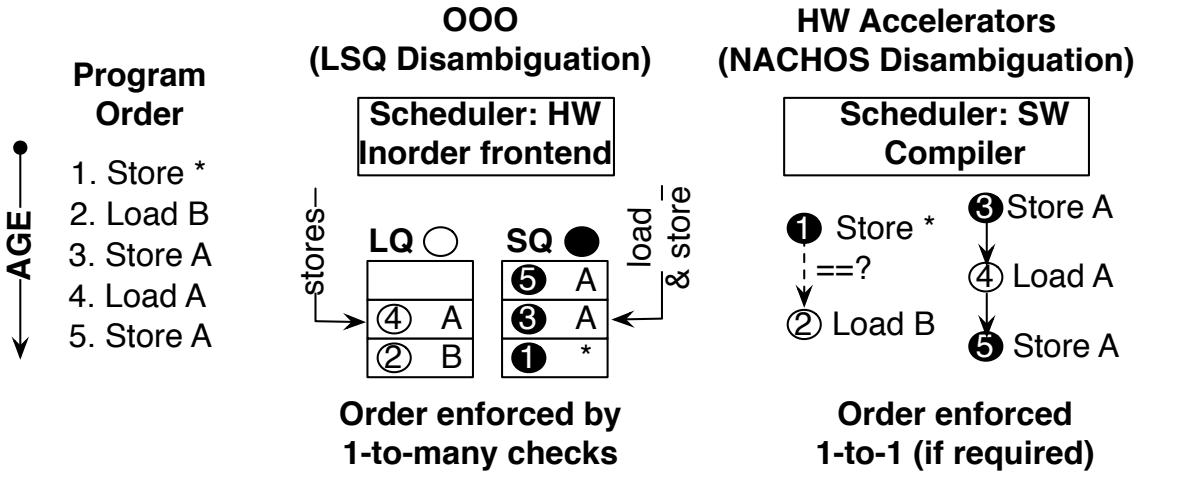
Some analysis instead exploit the dynamic aliasing relationships present in the executing software, in contrast to the static relationships that alias analysis can derive. Mock et al. determined that most dynamic points-to sets were small in practice (of size 1, 98% of the time) and that optimizing based on the dynamic alias relationships can improve performance [25]. The predictability of aliasing behavior from profiles has led to other works that speculatively exploit likely aliasing relationships, with rollback when speculation fails [5, 20].

Chapter 3

Scope of the Work

NACHOS is independent of the micro-architecture of the hardware accelerator. It is suited for hardware accelerators that implement the dataflow graph of the offloaded region and enforce data dependencies either with custom net-list [40, 47] or as a spatial fabric [7]. Please see Chapter 4.1 for our compiling and acceleration infrastructure; simulation infrastructure is described in the Chapter 5.7.

Memory disambiguation requires the following i) *ST-ST* ordering: in-order retirement of stores to the same address to ensure the correctness of final value in a location, ii) *ST-LD* ordering, to forward in-flight values from older stores to younger loads, and iii) *LD-ST* ordering, to ensure that stores do not corrupt the values of older loads. *LD-LD* ordering to conflicting addresses is only required in parallel programs that share memory regions across threads and can have data races (we address this in Chapter 5.6).



NACHOS enforces the memory dependencies explicitly as def-use dependencies. * (1): indicates an address unknown until execution which conflicts with all memory operations in an LSQ until the address is known. *NACHOS* can use alias analysis to restrict potential conflicts for *. When the compiler is unsure it inserts a == hardware check e.g., between ❶ and ②

Figure 3.1: Memory disambiguation. LSQ vs *NACHOS*.

Figure 3.1 illustrates an LSQ interface; A CAM queue each for load and store operations. Loads (◯ in the figure) check for matches in the store queue, and stores (●) check for matches in both queues. Each check is an energy hungry 1-to-many CAM entry check. A key challenge with incorporating LSQ based approaches is scaling up and down with the number of memory operations and MLP in the accelerated region. As we show, programs can contain between 10—50% memory operations in the accelerated region and MLP between 2—32 operations (see Table 4.1). Both of these impact energy and area. Many hardware accelerators assume a dataflow based execution model, and the accelerator cannot reconstruct program order due to the lack of a front end. The age is required to be encoded in operation and mapped a priori to LSQ entries, making LSQs over-provisioned [16, 34] and have increased occupancy [11].

Figure 3.1 also demonstrates how *NACHOS* handles memory disambiguation and can accommodate varying degrees of MLP and number of memory operations. *NACHOS* leverages compiler alias analysis and uses dataflow dependencies to specify the ordering between memory operations. For instance, here it determines that ①,❷ do not alias with any of the ❸,④,❺, in which case in the dataflow graph there are no dependency edges inserted between these operations (permitting them to proceed in parallel). In the case of ST-LD dependencies, the compiler uses instruction dependencies to forward values (❸—❺), while in the case of ST-ST and LD-ST dependencies the dataflow edge ensures ordering. When the compiler is unsure it sets up a may alias edge (❶ $\overset{==?}{\dashrightarrow}$ ②), and additional hardware in the dataflow fabric compares the addresses to check if they alias. We describe our stage wise analysis and refinement approach in Chapter 5.2.

Chapter 4

Motivation

For 27 chosen benchmarks, we select the function that consumes the largest amount of time (profiled via gprof). We profile all dynamic memory operations within the function (see Table 4.1, C2). The memory operations were segregated on whether they refer to locations on the *stack* or the *heap*. When a program allocates memory dynamically (using `malloc` or `new`), it is reserved on the heap. Calling glibc's `alloca()` allocates memory from the stack. However, the primary use of stack memory is to provide programmer transparent temporary storage. The compiler uses the stack to store local variables with limited scope, pass function arguments and supplement the lack of registers (register spilling).

In 7 of 28 applications, the number of memory operations which access stack locations account for 75% of all memory operations. To determine the primary cause of stack operations, we analyze each workload starting at the selected function. Our search is inter-procedural and recursively analyses all functions called from the selected function. We find that only 9 of 28 workloads have function-local data allocated on the stack. Of these, the average is seven variables. The highest number of variables is 69 for 401.bzip2, due to a lot of temporary integer buffers used in the compression algorithm. The dynamic behavior is profiled on a 64-bit system. Unlike 32-bit architectures, 64-bit architectures like the Microsoft X64, System V calling conventions, and Linux allow passing of parameters in registers. In the workloads we study, this covers all call sites of interest. Thus stack memory is primarily referenced due to register spills. This problem is particularly acute in x86 systems which have eight and sixteen registers in 32-bit and 64-bit flavors. Dataflow oriented hardware accelerators, unlike Von-Neumann architectures, are not constrained by a fixed number of registers. Thus all stack operations can be converted to local references which could be issued to the memory system.

4.1 Baseline Hardware Accelerators

How are acceleratable regions identified? The baseline we evaluate; models accelerators which explicitly embed the operations of a dataflow graph in a grid of function units and closely integrated with the OoO processor [7, 9, 21, 28]. Such accelerators statically map the instructions to function

units using a compiler. The accelerator we model is a grid of reconfigurable function units (256 units). See Chapter 5.7 for our simulation infrastructure. The accelerator can autonomously issue memory operations independent of the Out of Order (OoO). Note that the accelerator hardware by itself cannot determine program order for memory operations since it lacks instruction fetch. The compiler thus exposes total memory order through the form of explicit IDs [42]. The explicit ID (8 bits; max of 256 memory operations in offload) directly maps the memory operations to LSQ entries [11]. We use NEEDLE to identify hot paths [17, 18] to be offloaded to the accelerator. We select the hottest path (i.e., highest % of dynamic instructions) with the largest number of memory operations for this study. Similar to prior work [9, 21], the profiled paths are converted into superblocks [15]. Superblock formation is a static compiler analysis that groups together program basic blocks with a high likelihood of executing one after another. This gives the compiler opportunity to perform optimizations on a larger window of instructions.

The accelerator paths we study were recently released at IISWC. [18][1]

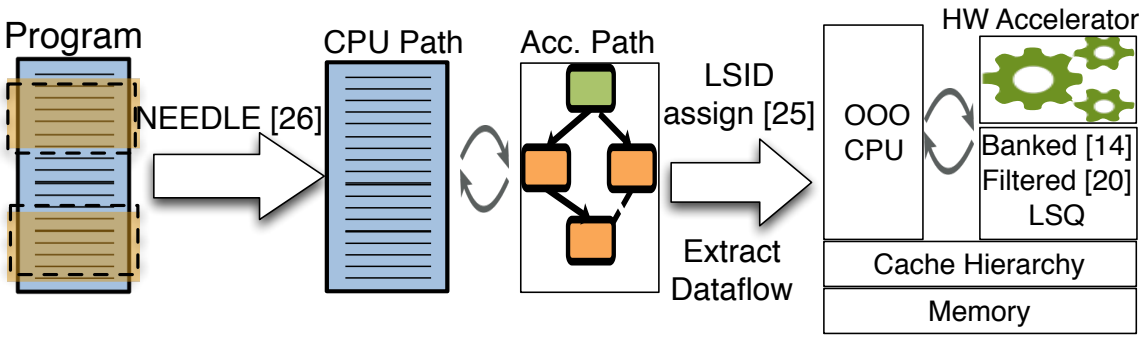


Figure 4.1: Overview of *NACHOS* framework. We used NEEDLE [17] for extracting hot paths to run on the accelerator. LSID Assign [42]: The compiler indicates total memory order to the dataflow graph.

Workload Characteristics: Table 4.1 describes the features of the accelerated program paths. We profile all dynamic memory operations within the function (see Table 4.1, C2). The memory operations were segregated on whether they refer to locations on the *stack* or the *heap*. When a program allocates memory dynamically (using `malloc` or `new`) it is reserved on the heap. The compiler uses the stack to store local variables with limited scope, pass function arguments and supplement the lack of registers (register spilling).

In many applications, the maximum amount of memory dependency checks is caused by the stack due to register refills. Hardware accelerators typically use a fixed-size local SRAM to replace the stack. In 12 of 28 applications, the number of memory operations which access stack locations account for 20%+ of all memory operations (50%+ in 2 applications). Note that in a conventional OoO the LSQs and caches do not distinguish stack from heap operations. Hardware accelerators do not rely on the stack; they use local custom storage and transform stack memory accesses into

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

local accesses [7, 17, 40, 47], which helps restrict the scope of compiler alias analysis to only heap operations in hardware accelerators. To determine the primary cause of stack operations, we analyzed the function and found that principal cause of stack related memory traffic is register spills. We find that only 9 of 28 workloads have function-local data that allocated on the stack. Of these, the average is seven variables (max:69 in bzip2). Note that such memory traffic will be eliminated entirely from the LSQ if a custom number of registers or local RAM is used, like in a hardware accelerator (GPUs make similar tradeoffs[2]).

Table 4.1: Workload Characteristics

		C1	C2	C3	C4	C5	C6	C7	C8	C9
							Heap Depend.			Function
	App	Function	Cov%	#OP	#M	MLP	S-S	S-L	L-S	%STACK
6***SPEC2k**	gzip	longest...	59	64	4	4	.	.	.	21
	art	match ...	11	100	36	4	6	6	10	0
	181mcf	price_...	10	29	2	2	.	.	.	5
	equake	smvp ...	53	559	215	16	.	.	12	2
	crafty	Evaluat...	3	72	7	8	.	.	.	40
	parser	table_...	51	81	12	4	.	.	2	34
10***SPEC2k6**	bzip2	BZ2_co...	5	501	110	128	3	.	3	27
	gcc	bitmap:.	67	47	2	2	.	.	.	26
	429mcf	price_...	10	30	3	4	.	.	.	24
	namd	Compute...	44	527	100	16	6	6	30	41
	soplex	CLUFact...	12	140	32	4	.	.	8	19
	povray	All_Sp...	1	223	74	32	4	21	24	95
	sjeng	gen ...	5	99	11	8	.	.	.	33
	h264ref	dct_lu...	19	224	42	8	.	.	5	27
	lbm	LBM_pe...	96	427	57	32	.	.	.	12
	sphinx3	vector:.	40	133	20	32	.	.	.	0
10***PARSEC+Others**	blacks.	BlkSchl...	8	297	0	0	.	.	.	4
	bodytr.	ImageMe...	2	285	42	4	30	30	42	10
	dwt53 .	dwt53_...	37	106	16	16	.	.	.	11
	ferret.	image_...	3	185	0	2	.	.	.	29
	fft-2d.	fft ...	22	314	80	4	.	.	48	18
	fluida.	Compute...	4	229	28	8	.	.	.	14
	freqmi.	FPArray...n	3	109	32	4	.	.	8	17
	sar-back	sar_ba...	1	151	7	8	.	.	.	64
	sar-pfa.	sar_in...	7	500	32	16	12	20	12	19
	stream.	pgain ...	41	210	32	16	.	.	.	0.5
	histog.	rgb2hsl...	70	522	48	16	.	.	.	0

C2:Cov%: Fraction of dynamic instructions accelerated. C4:#M: Number of mem ops. C5:MLP: Number of parallel ops the memory system had to issue for optimal performance; we varied from 2—128. e.g. 16 means MLP is between 8 to 16

There is little aliasing behavior on the heap, which implies that the majority of the LSQ checks are empirically for operations that may execute in parallel with each other. Our hypothesis is that if a compiler can identify much of this parallelism, then hardware cost would be less to enforce the dependencies between actual aliasing operations. Table 4.1 (C6, C7, C8) shows the breakdown of aliasing behavior in accelerator friendly regions of the workload. The memory dependencies were collected from a dynamic run of the program. Column C6 illustrates the number of aliasing

[2]http://docs.nvidia.com/cuda/cuda-c-programming-guide/

addresses between store operations. C7 shows stores which alias with existing loads and C8 shows loads which alias with existing stores. These counts represent the number of aliasing operations for each execution of the path.

4.1.1 Hardware Accelerator with LSQ

The baseline LSQ design we assume is address partitioned [35]. These designs interleave LSQ banks based on cache-line addresses, and deal with the resultant overflow challenges by prioritizing older instructions and occasionally rejecting younger instructions using flow-control techniques. This LSQ design creates opportunities for scaling. The energy-efficiency of the small address interleaved LSQ banks can be improved further by the addition of simple Bloom filters [36]. See Chapter 5.7 for simulator details.

Challenge 1: Area of LSQ

A key concern when incorporating LSQs in accelerators is the area. Unlike an OoO core, hardware accelerators strip out all extraneous structures. Thus provisioning an LSQ for an accelerator constitutes a significant fraction. For instance a 48 entry, 2-ported LSQ is equivalent in area to 50 integer ALUs. LSQ structures for TRIPs [34] occupy more than 50% of the data tile; Banking and reorganization [37] will improve scalability but will not impact area.

The number of entries in the LSQ predetermines how many independent memory operations can be issued (effectively regulating MLP [37]), and the number of ports in the LSQ determines overall instruction throughput.

Accelerators tend to have irregular memory access behavior with variations both in the number of memory operations and amount of MLP (see Table 4.1). Accelerators also seek to achieve high peak instruction throughput since they dedicate most of the hardware for function units. These design issues make it more challenging to design LSQ size and ports. Overall, we find that 11 workloads have up to 20 memory operations but six workloads have 50+ memory operations. The MLP can also vary significantly (Table 4.1:C5); 16 apps $<$8ops and 4 apps $>$32 ops.

Challenge 2: Energy Efficiency

We find that even for an optimized LSQ design (i.e., Partitioned + Bloom filter) LSQs impose a geomean of 2.3$\times$ energy overhead relative to the compute. For the LSQ design without the Bloom filter, LSQs impose a geomean of 2.6$\times$ energy overhead relative to compute.

We show the dynamic energy overhead of the LSQ relative to the compute energy (i.e., function unit energy) of the accelerator [7]. The LSQ is partitioned (48 entries/bank), and the number of banks activated for a particular application depends on the proportion of memory operations in the accelerated region. Our energy cost model is adopted from McPAT [19]; Table 5.4 lists the parameters and simulation infrastructure (in Chapter 5.7). The breakdown of energy consumed by compute operations (INT/FP) and memory operations (LSQ energy) is shown in Figure 4.2. The number

above each stack represents the fraction of memory operations in the accelerator. Overall, the LSQ constitutes the dominant overhead: geomean 64% of total energy. We find that for two workloads (blackscholes and ferret) the accelerated region does not include memory operations; a compiler-based approach would recognize the opportunity to get rid of memory disambiguation completely. Only in floating point, heavy workloads does compute dominate as expected, e.g. 470.lbm.

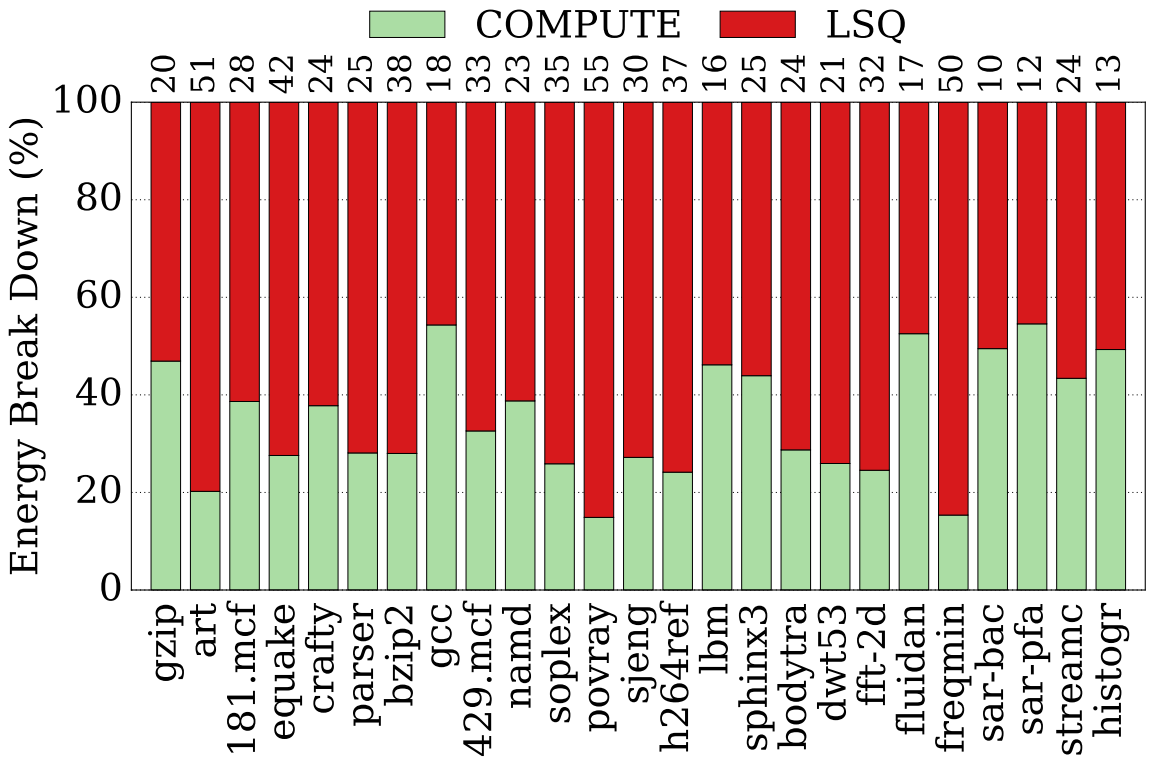


Figure 4.2: Energy Breakdown (INT/FP, LSQ)

LSQ-Opt (Partitioned and Filtered [36]) Bloom filters have been shown by Sethumadhavan et al. [36] to reduce the number of LSQ checks and save energy. We study the potential design space of a Bloom filter to reduce the overall LSQ access energy.

The Bloom filter summarizes entries in the LSQ and acts as a filter. All memory operations check the Bloom filter. When a memory operation accesses a Bloom filter, it either gets a response of "yes – the memory operation is present in the LSQ", or it gets a response of "no – the memory operation is not present in the LSQ". A Bloom filter has zero false negatives, i.e. if the Bloom filter says no, it is definitely true. However, it can have false positives, i.e Bloom filter hits (cases when Bloom filter says yes) can be false. In case of Bloom filter hits, the memory operation also needs to check the LSQ, consuming LSQ access energy. Thus, it is required that Bloom filter has low false positive (FP) rate (i.e. $\frac{FP}{FP+TN}$).

Figure 4.3 plots the FP rate while varying the size of the filter by a factor of two from 32 entries to 512 entries. Each entry corresponds to a 6-bit saturating counter (empirically determined to not overflow for our workloads). Two different types of hashing functions are studied a) LSB

hashing [36] (○) and b) Knuth's Multiplicative Hash (KMH)(▲). Each workload is represented as two scatter points, one for each type of hash.

Increasing the size of the filter often implies a commensurate decrease in FP rate. For the 27 workloads we study, we found that KMH had 5 workloads with non-zero FP rate and LSB had 2 non-zero FP rate, when the size of the filter was increased to 512 entries. Thus we select Bloom filter with 512 entries (375 bytes) and LSB Hashing.

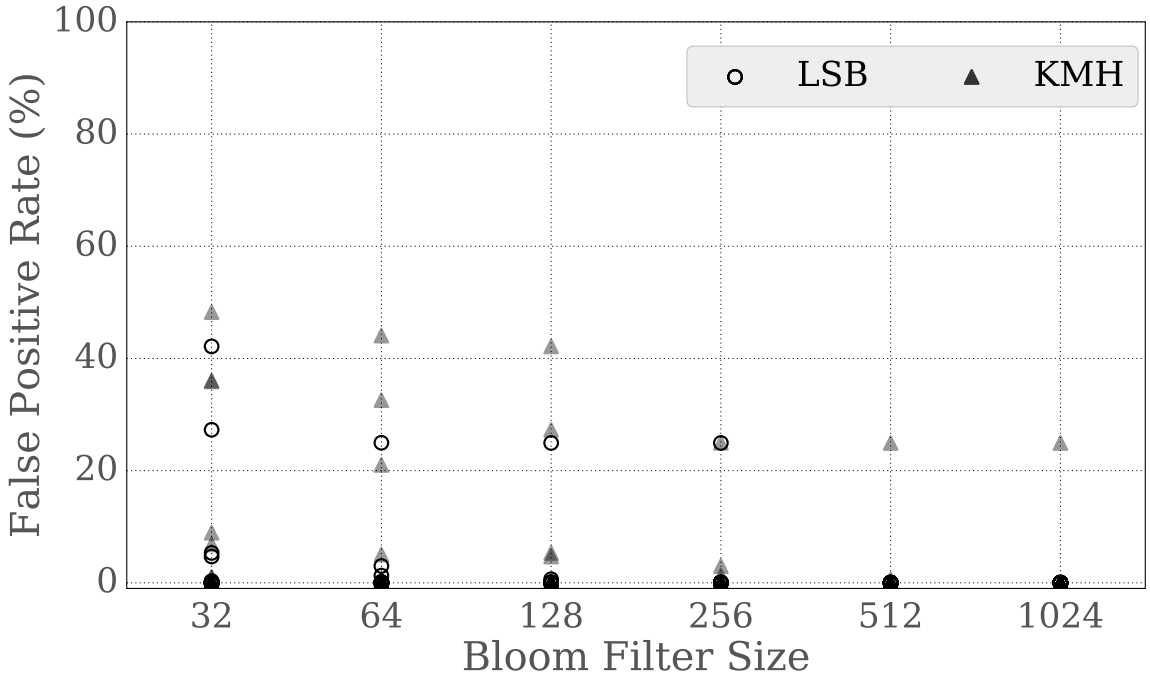
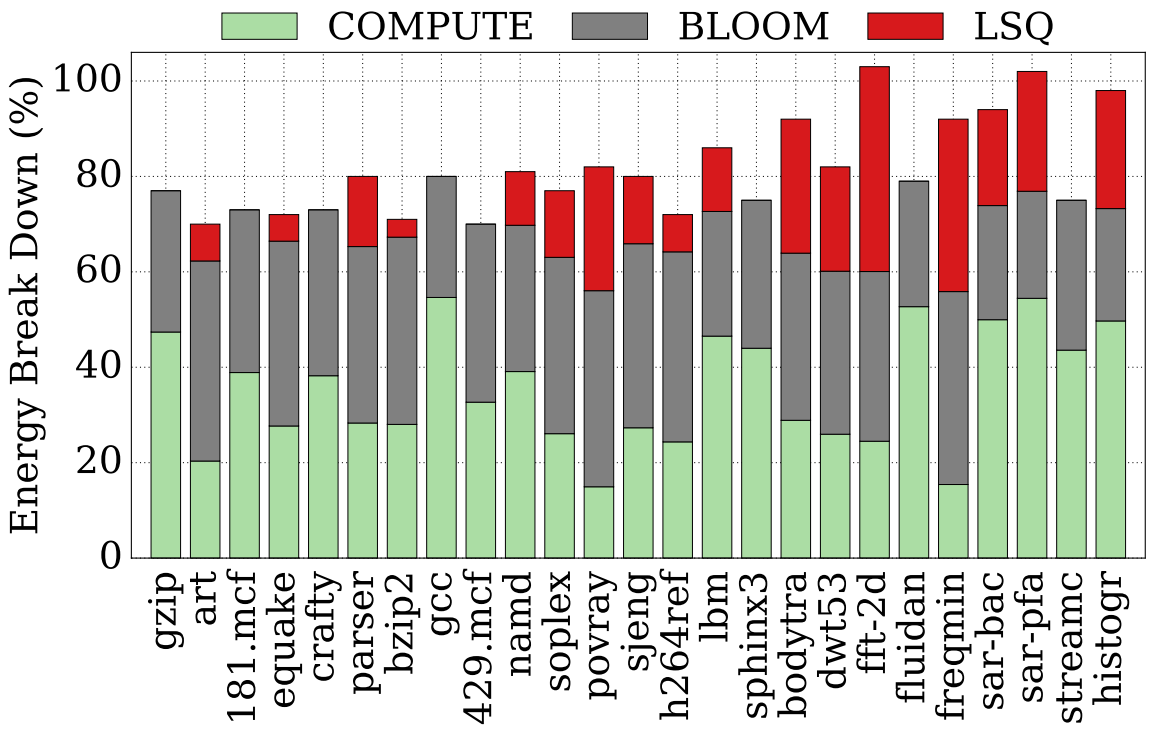


Figure 4.3: Bloom Hash Function

The primary energy cost of a Bloom filter is that spent in the memory core [22, 33]. Our energy model indicates that probing the selected Bloom filter configuration is $30-50\%$ of the cost of probing the LSQ.

We extend our baseline partitioned LSQ with a Bloom filter (512 entries; 6-bit counters). Figure 4.4 shows the energy breakdown for compute, LSQ and Bloom-512 (normalized to Figure 4.2). We observe that the geomean energy reduction of Bloom+LSQ compared to the LSQ-only is 18.5%.

Ten benchmarks have perfect Bloom filter behavior, i.e., 0 hits and no LSQ checks. Note that the Bloom filter is strictly an energy optimization; a full capacity LSQ is still needed to handle all the in-flight memory operations in case the Bloom filter hits. For these workloads, we see a geomean 18.5% (max 30.4%, 429.mcf) reduction in energy consumption compared to an LSQ baseline. In fft-2d (103%) and sar-pfa-interp1 (102%), the Bloom filter+LSQ combination is more energy hungry than only an LSQ. The reason being high Bloom filter hit rate. The hit rate of fft-2d (63.4%) and sar-pfa-interp1 (52.9%) is top two amongst the workloads we study. In case of Bloom filter hits, the LSQ-Opt design incurs the cost of Bloom Filter checks as well as LSQ checks whereas, the



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 4.4: LSQ-Opt Energy Breakdown (INT/FP, LSQ, Bloom-512) - Normalized to Figure 4.2

LSQ baseline only incurs the cost of LSQ checks. The geomean hit rate for non zero (seventeen) benchmarks is 23.3%.

Challenge 3: Scaling LSQs with #Memory operations and MLP

LSQ size must scale with the increase in number of memory operations and also has to have fixed Low overheads for handling compute-intensive accelerators. For instance, TRIPS adopted LSQ designs from traditional OoO cores. However, the structure was scaled up in size to accommodate the potentially larger number of in-flight memory operations. For the regions that we focus on, we find that the number of memory operations can vary a lot. Table 4.1 shows the number of static memory operations. While the median is 32, it ranges from zero (ferret) to 215 (183.equake). Accounting for the worst case size for an LSQ may lead to overall increased static power, due to large CAM structures.

Chapter 5

NACHOS: Compiler Assisted Memory Disambiguation

An age-ordered LSQ answers the following question: *For a memory operation X, what are the in-flight memory operations that overlap with X and what is the program order between X and the overlapping memory operations ?*. The LSQ answers this question by maintaining all in-flight addresses of memory operation, and when a memory operation arrives it checks against all in-flight memory addresses. *NACHOS* answers the question: *Given two memory operations X and Y do they alias or overlap and what is the program order between them.* The key difference is that *NACHOS* performs most of the alias checks statically ahead-of-time and saves dynamic energy. *NACHOS* does use a hardware assistant to check aliasing at runtime to discover parallel memory operations, but only when the compiler is unsure. Compiler alias analysis can emulate LSQ-based memory disambiguation by considering all memory operations pairwise in the program region. While considering all pairs of operations may seem daunting, it is tractable for accelerators, which only focus on a fixed window of instructions(see C1:Table 4.1). Additionally, the compiler uses program characteristics such as type information to eliminate alias candidate pairs quickly. Overall, we find performing pairwise alias analysis checks takes less than one second for each acceleration region.

5.1 Dataflow Accelerator with Memory Dependencies

NACHOS can be adapted to work with any dataflow-based accelerator. *NACHOS* only relies on the accelerator to enforce ordering between memory operations similar to data dependencies and is independent of the particular alias analysis. The input to *NACHOS* is the dataflow graph of the program region to be offloaded and the program order of all operations. *NACHOS*'s output is an augmented dataflow graph which includes *memory dependence edges* (MDEs) to ensure correctness in the absence of an LSQ. Following this, the back-end of the compiler can then generate a static schedule [7,29,47] to map the operations to dataflow functional units and enforce MDEs in a fashion similar to instruction dependencies. *NACHOS* (see Figure 5.1) processes the dataflow graph of the

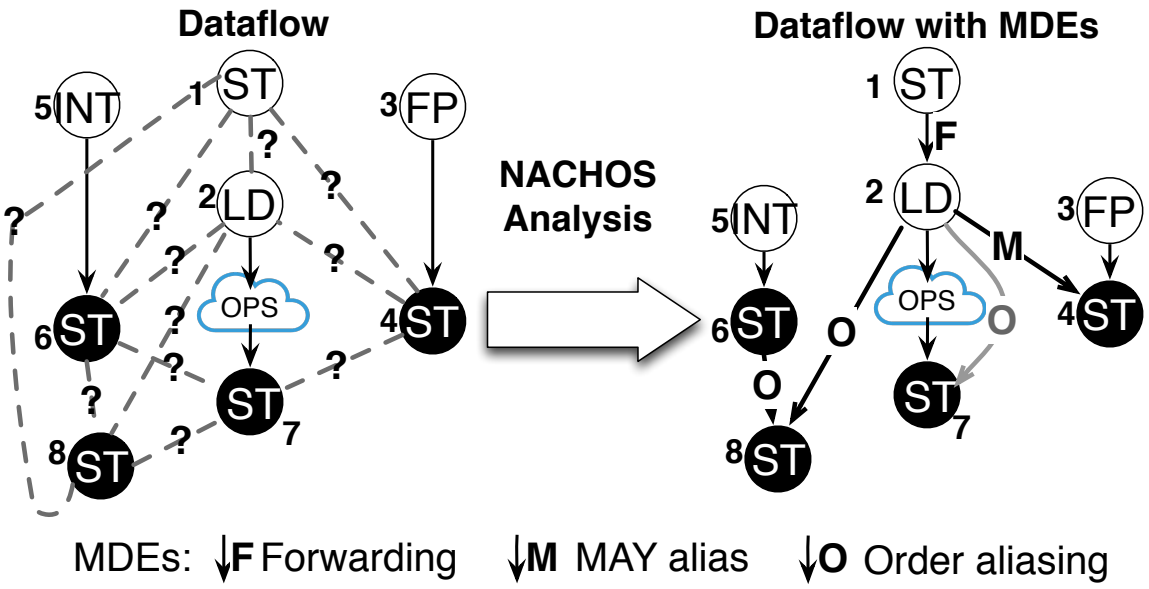


Figure 5.1: Memory disambiguation using *NACHOS*. Forwarding, May and Order edges are introduced to eliminate LSQs. LD: Load operation, ST: Store operation, INT: Integer operation, and FP: Floating point operation

accelerated region and performs pairwise alias checks for all memory operations (indicated by the ? in the figure). The compiler can provide three types of responses for each alias check – NO, MUST and MAY. For NO aliasing memory operation pairs, the compiler does not add any dependency edges allowing them to execute in parallel. For MUST aliasing memory operation pairs, *NACHOS* ensures the correct order of memory operations by introducing two types of MDEs in the dataflow graph – *ORDER*(O) and *FORWARD*(F). Finally for MAY aliasing memory operation pairs (i.e., when the compiler is unsure whether two operations alias), *NACHOS* inserts a special *MAY*(M) MDE between the memory operations which leverage hardware assistance for checking. Note the dataflow graph is a directed acyclic graph. The source and sink memory operations between the edges are decided based on the LSIDs passed to *NACHOS* by the NEEDLE [17] framework (see figure 4.1). LSIDs are derived based on the program order, thus the source memory operation is the older (or lower LSID) of the two memory operations and the the sink memory operation is the younger (higher LSID) of the two memory operations.

- **ORDER (O) Edges** (1 bit valid): The oreder dataflow edges are inserted between a load operation and a store operation (LD-ST), and between a store operation and a store operation (ST-ST) (e.g., (2) – (8)) that *must* alias with each other. *ORDER* edges do not carry values between the two memory operations. They ensure that the operations to the same memory location are executed in program order. *ORDER* edges are enforced similar to instruction dependencies. Additionally, note the edge (2) – (7). This edge is interesting from the perspective that an existing instruction dependence through non-memory operations already

ensures ordering between (2) and (7). An MDE is redundant, and we exploit this information in Section 5.2.3.

- **FORWARD (F) Edges** (72 bits – 64 bit value, #Bitmap): The order dataflow edges are inserted between a store operation and a load operation (ST-LD) that the compiler deems *must* alias with each other. *FORWARD* edges pass values between the older store and a younger load. The bitmap indicates which bytes were written so that partial forwarding can be supported. When the number of store operations (STs) which forward partial values to a younger load operations (LD) exceed two, we convert the ST-LD pairs to *ORDER* edges. Thus the LD proceeds only after the STs complete. Most popular hardware LSQ do not support partial forwarding. Unlike a hardware LSQ, the compiler can make the decision on when to *FORWARD* and when to *ORDER* individually for each memory operation pair.
- **MAY (M)** Edges (64 bit address, 1 bit valid): *MAY* edges have no equivalence in LSQ parlance; they are an artifact of compiler alias analysis. For a pair of memory operations, the compiler may be unable to ascertain a strict must or must-not alias relationship. In such cases, *NACHOS* inserts a *MAY* edge between the pair of memory operations. At runtime, the memory address is passed from the source memory operation to the sink memory operation. The sink memory operation uses a comparator (==?) to determine if two addresses overlap i.e, if the memory regions conflict which is determined by assuming a fixed eight byte memory region. For instance, consider the two operations (2) and (8). At runtime, the hardware function unit assigned to sink will compare (8)'s address with the address passed from (2) and determine if they overlap (i.e., they alias). If they alias, then (8) stalls until (2) completes execution i.e., MAY converted into an *ORDER* edge. If the addresses do not overlap, then the younger operation is allowed to proceed, i.e., MAY turned to an NO case.

The key challenge to *NACHOS* are the compiler-introduced *MAY* edges. *MAY* edges require additional hardware checks, thus increasing energy consumption. Note that these checks are pairwise between memory operations as opposed to centralized checks. Hence a memory operation that may alias with many operations could potentially be an energy hog. We elaborate on the alias analysis strategies and refinement to reduce the number of *MAY* alias cases in Section 5.2.

Ground Truth: Do memory operations conflict or alias? Two memory operations require ordering when they access the same memory location and at least one of the operations is a store [1]. Figure 5.2 shows the breakdown of aliasing memory operations at runtime observed in our workload suite. We instrumented and collected the addresses for every *heap* memory access within frequently executed, accelerator friendly regions. An accelerator does not include any stack operations since the stack in the original function is replaced with custom local scratchpad into which map all stack and temporary data [40, 47].

[1] LD-LD ordering is needed for enforcing consistency; discussed in Chapter 5.6

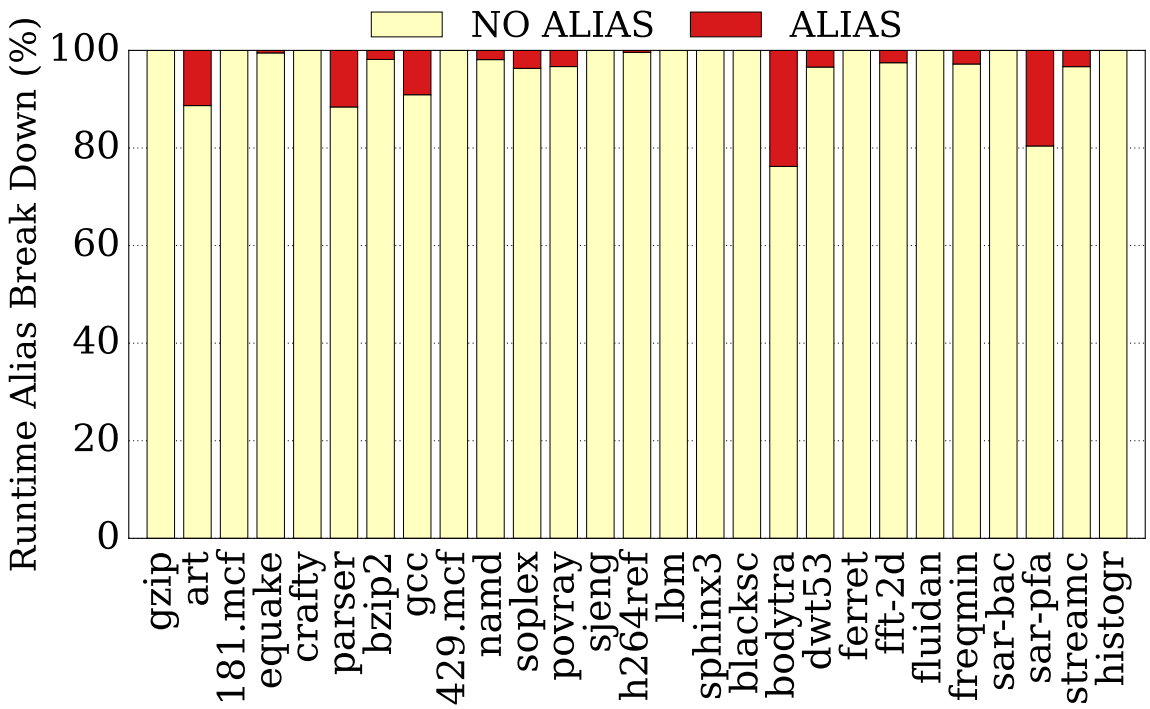


Figure 5.2: Pairwise alias checks (top five accelerated paths).

Most pairwise heap address checks in the frequently executed regions of the workload do not alias. We found that 10 out of 27 workloads do not have memory accesses which alias. The bodytrack has the most (292) pairs of memory operations which alias at runtime in the top five frequently executed regions. An average of 27 pairs of memory operations alias, amongst benchmark with such behavior.

Our profile also revealed that pairs of memory operations seem to exhibit stable aliasing behavior, which means that if two memory operations alias in the first dynamic instance, then they alias for every dynamic instance of those operations. It indicates that the hardware LSQ often performs redundant checks for each subsequent dynamic instance of the operations. In this work, we endeavor to determine the exact aliasing relationship between memory alias pairs at compile time itself.

Alias Analysis $\neq$ LSQ address checks: Compilers can reason abstractly about the possible relationships between pointers. The structure of the program provides static guarantees about the program behavior even without knowing any of the memory addresses involved and in general uses a rich array of contextual program information (both control and dataflow). Consider the operations in Figure 5.3. In this case, compiler alias analysis would identify that pointers `p` and `q` must hold different addresses, so the stores cannot alias. Two different pointers are computed, `p` and `q`, at two dynamically computed offsets into two different arrays (A and B) at offsets `x` and `y` respectively. If we consider only the relationship between two stores to memory performed by `*p=...` and `*q=...`,

the compiler may observe that both p and q points to entirely different arrays; and no aliasing is possible. Note if the arrays are not in bounds then that is undefined behaviour.

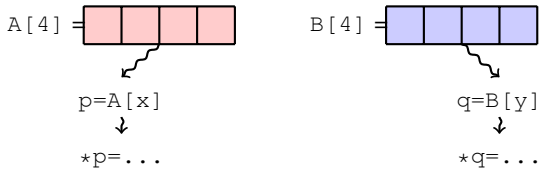


Figure 5.3: Analysis can prove p and q do not alias if x and y can be proven to be within bounds of allocated memory.

5.2 *NACHOS* analysis

The techniques described in this section allow us to improve accuracy and prune the number of memory orderings to enforce for correctness. Figure 5.4 summarizes the three stages of *NACHOS* analysis. The first stage employs standard alias analysis passes in the compiler to label each pair of operations as *MUST*, *MAY* or *NO* Alias relations. The second stage uses inter-procedural information to further resolve *MAY* alias relations to *NO* alias relations if possible. Finally, the third stage leverages existing data dependencies to trim the number of MUST and MAY relationships that need memory dependence edges (MDEs).

Summary

- Stage 1 adds no MDE for seven workloads; of the remaining 20, a geomean of 10% of alias relations can be determined to be *MUST* or *NO*.
- Stage 2 further converts a geomean of 11% of *MAY*→*NO* alias relations for 10 workloads. Of these, it is particularly effective for five workloads where 22%–80% of *MAY* are converted to *NO*.
- Stage 3 removes 40%–84% of alias relations which do not need to be enforced due to transitive data dependencies within the dataflow graph.
- Across all our workloads, Out of a total of nC2 pairs of memory operations, we require MDEs for a geomean $\simeq$ 25% of memory operation pairs. Overall added MDEs incur 30%–70% lower energy (potentially due to link energy, bytes transfer, and comparator checks) compared to an LSQ (which checks all in-flight memory addresses). The details of which can be found in Chapter 5.3

5.2.1 Stage 1: Off-the-shelf Alias Analysis. Assigning MAY, MUST and NO labels

NACHOS analyzes memory operations pairwise and assigns a label to each pair of memory operations which indicates their aliasing relationship. In the first stage, *NACHOS* uses advanced compiler alias analysis passes in a production compiler (LLVM 3.8). Table 5.1 enumerates the list of analysis.

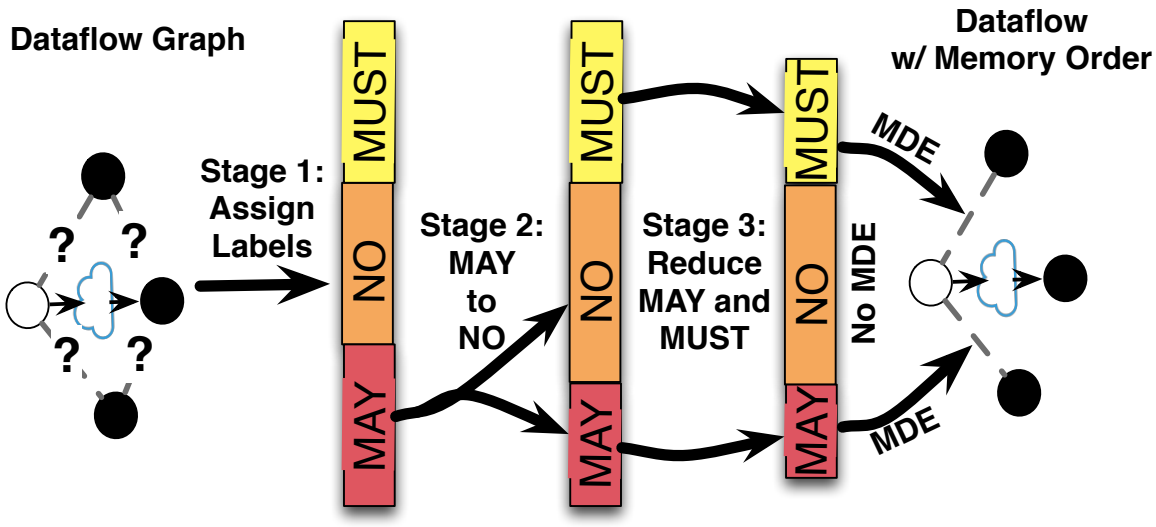


Figure 5.4: Stage-wise pruning and refinement of alias relations into MDEs to be enforced. Refer to Figure 5.1 for MAY Edge ($\xrightarrow{M}$) and MUST Edge ($\xrightarrow{O}$ or $\xrightarrow{F}$)

For each pair, three types of alias labels are possible *MUST*, *MAY*, and *NO*. The *MUST* result from memory operations that provably identify to the same location. Similarly, *NO* result from memory operations that identify to independent memory locations according to alias analysis. 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 be executed in parallel. However, because alias analysis is undecidable [32], it can also give up and say that two accesses may or may not alias (i.e., *MAY* alias relation). Stage 1 alias analysis efficacy is limited in workloads where the accelerator regions are composed of complex program paths (i.e., not just simple loops or array accesses). In most workloads, 19 of 27, the dominant form of relationship is the *MAY* alias.

Table 5.1: Stage 1: Alias Analysis Passes

Name	Description
Basic	Stateless checks, eg. Base pointers, constant pointers
TypeBased	Uses object types
Globals	Tracks global variables and function purity
SCEV	Limited pointer arithmetic to handle loop accesses
ScopedNoAlias	Uses variable scope information
CFL	Targets data structures ([52,53])

Figure 5.5 summarizes the impact of applying standard alias analysis (see Table 5.1) on all pairs of memory operations in the top five frequently executed accelerator friendly regions. Some benchmarks (see (1) in Figure 5.5) have no stores among the memory operations in these regions, i.e. gzip, mcf, crafty, and blackscholes. The benchmark sjeng has store and load operations (2), however alias relationships between all pairs are perfectly identified by this stage. Overall, 7 of 27 workloads

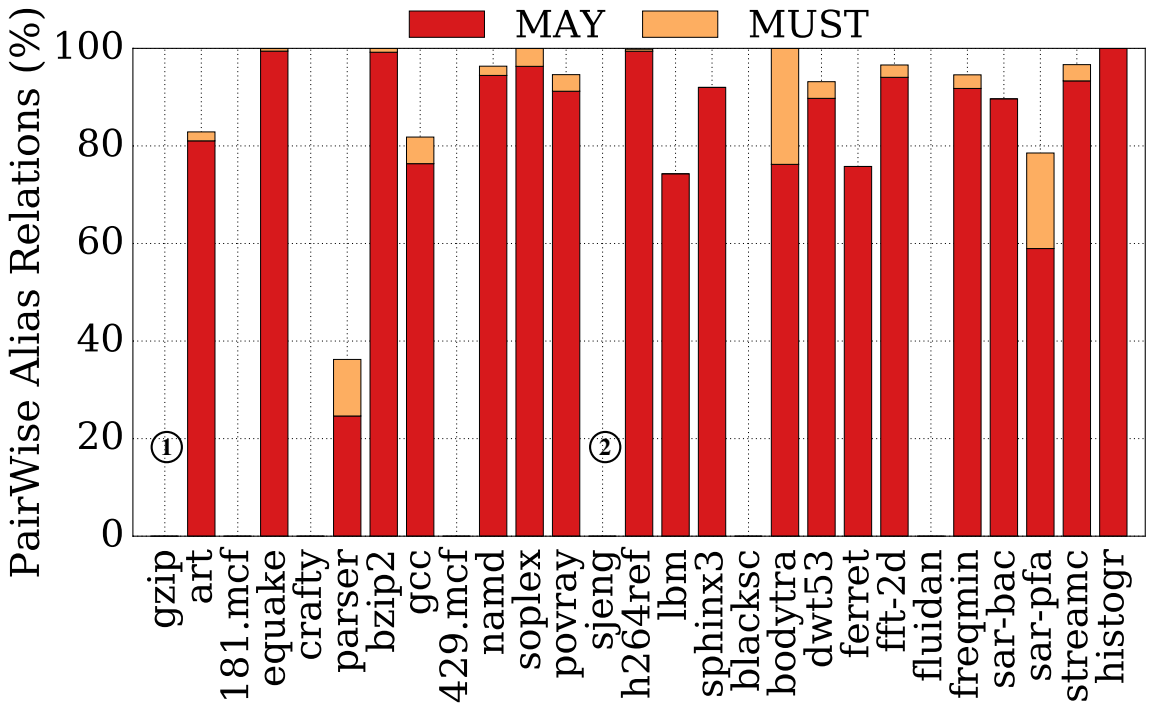


Figure 5.5: MAY and MUST alias relationships between memory operation pairs identified by Stage 1. MAY and MUST require MDEs. NOs do not and not shown in plot. Top 5 paths.

need no further analysis. Of the remaining 20 workloads, the stage 1 can classify on average 3% of pairwise checks as *MUST* alias and 7% as *NO* alias relations per workload.

5.2.2 Stage 2: MAY → NO using inter-procedural analysis

Note that the standard alias analysis presently in LLVM 3.8 (see Table 5.1) cannot reason across function boundaries to determine aliasing relationships between pointers. While investigating the sources of the *MAY* alias relationships in the benchmarks, we observed that some of the pointers were derived from global or local variables whose addresses were taken and passed as function arguments to the accelerated region. Furthermore, we observed these could be resolved to *NO* alias relationships by tracing the provenance of the pointers back across one function call boundary to the source global or local variable. This simple analysis takes as input the *MAY* alias relations from the previous stage and attempts to trace the data-dependence of the pointer back into the function call to a source object. When two memory operations have their pointers traced back to different source objects, those pointers can be safely classified as *NO* alias.

Figure 5.6 presents the results of applying the inter-procedural analysis to the previous *MAY* alias relations obtained in stage 1. Ten workloads with *MAY* alias relations were refined by stage 2 of *NACHOS* analysis. Where the inter-procedural analysis was useful, it converted a geomean of 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

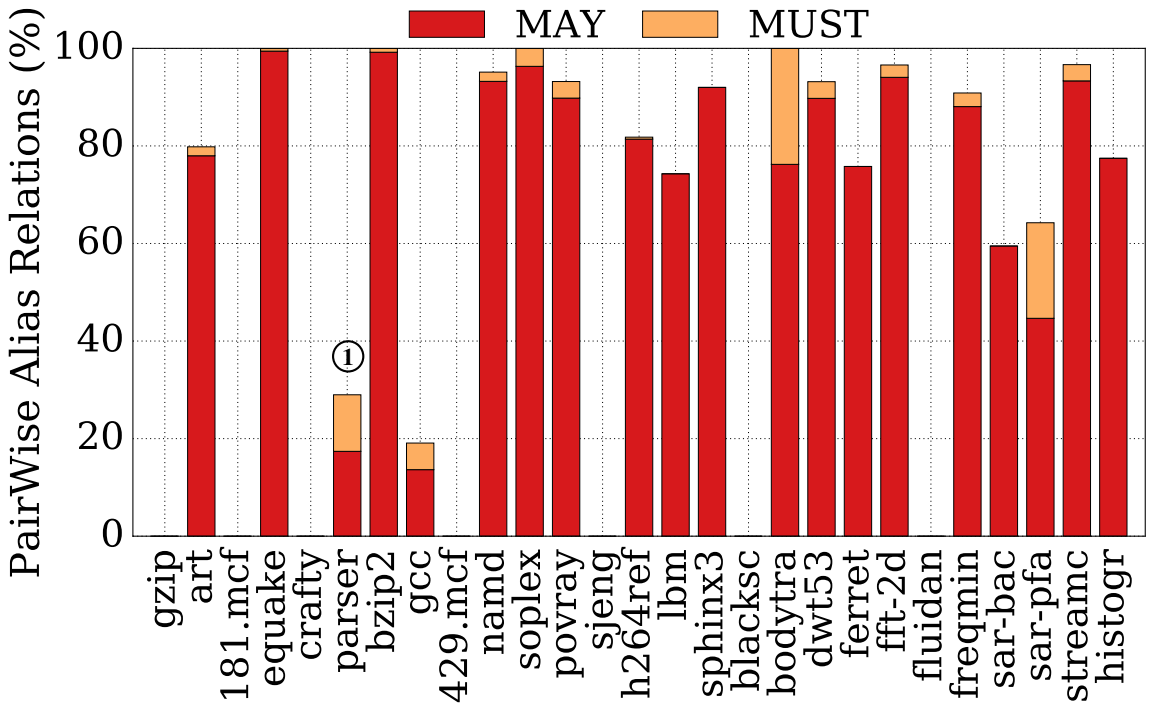


Figure 5.6: Stage 2 : Refinement of MAY alias from Stage 1 using inter-procedural object equivalence. % of MAY + MUST in Stage 2 shown as a fraction of all alias relationships. MAY converted to NO only in this stage. Top 5 paths.

variable – `Table_connector ** table`. Stage 2 is able to convert 29% of *MAY* alias relations to *NO* alias relations in parser. Similarly, inter-procedural checks are particularly effective in gcc, sar-pfa-interp1, sar-backprojection and histogram. In all these workloads, upto 82% of *MAY* alias relations are converted to *NO* alias relations.

5.2.3 Stage 3: Removing redundant MAY and MUST

The *MUST* alias results from stage 1 combined with the refined *MAY* alias results from stage 2 identify the memory operations whose execution order must be constrained in order to ensure correct program execution. However, not all of the alias relations identified by stage 1 and 2 need to be enforced in the dataflow graph. We often find that there already exists a transitive data dependence relation between a pair of memory operations in the dataflow graph which impose an ordering.

Consider Figure 5.7. The pairs ① – ❺ and ② – ❻ are identified as aliasing memory accesses. However the existing dataflow constraints (via ③) ensure that ① must finish before ⑤ begins, so there is no need to enforce an explicit ordering between them. Similarly, ② must complete before ❻ can execute – due to transitive constraints by ③ and ④. Simplification is critical because it reduces the number of links that must be added to memory operations and can thus reduce overhead while maintaining program correctness. The output of stage 3 is an augmented dataflow graph which

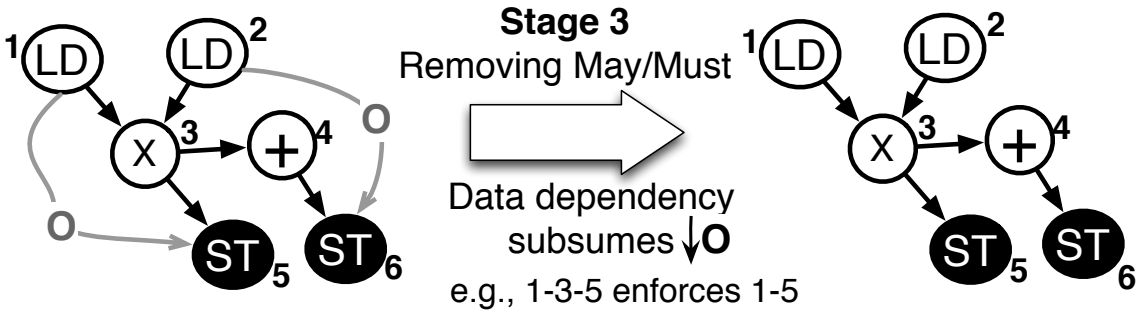


Figure 5.7: Implicit data dependencies eliminate the need to explicitly enforce ordering.

includes MDEs in addition to original data dependence edges. The MDEs serve to ensure correct ordering of memory operations without the need for an LSQ.

To remove these redundant aliasing relations, stage 3 performs a simplification pass. Note that the dataflow graphs of the regions we consider are directed and acyclic in nature. Checking for reachability between two vertices (memory operations) is sufficient to determine the need to enforce ordering. The offload region in the program is traversed in reverse topological order (postorder traversal of the dataflow graph). For each alias relation where the current node is the source, we check if the destination is reachable. If it is reachable, then we discard the alias relation as there exists an implicit data dependence. If it is not reachable, then we add an MDE to the dataflow graph. The nature of the MDE depends on the memory operation pair (see Section 5.1 for details). Additionally, all *MUST* alias relations are enforced prior to *MAY* alias relations. Once all alias relations are processed for a node, we proceed to the next node in reverse topological order.

Figure 5.8 shows the fraction of alias relations retained after simplification in stage 3 with respect to *all* alias relations determined in stage 1. Each bar is divided into the *MUST* alias and *MAY* alias relations which need to be enforced. Overall, we find that stage 3 can remove the need to enforce 68% of alias relations (*MUST* and *MAY*). The least amount of redundant relations was 40% in sar-backprojection and the largest in fft-2d, 84%. Removing redundant alias relations is critical to *NACHOS* as enforcing *ORDER*, *FORWARD* or *MAY* MDEs incurs energy overhead.

5.2.4 Polyhedral analysis: Multidimensional loops (MAY to NO)

Using standard alias analysis for 5 of the 27 workloads failed to provide meaningful alias information to minimize the addition of MDEs. We leverage Polly, an LLVM project which uses a mathematical representation based on integer polyhedra to analyze and optimize memory access patterns [8]. Stencil codes are a class of iterative kernels which update array elements according to some fixed pattern called stencil. The Polly project is suitable for analyzing the stencil based inner-loop patterns observed in the workloads where standard alias analysis fail. We find that applying polyhedral alias analysis locally to the specialized region was successful.

Polly provided comprehensive information on MAY aliases within the stencil pattern loops in 5 applications and managed to detect all the MAYs to be NO alias successfully. We manually

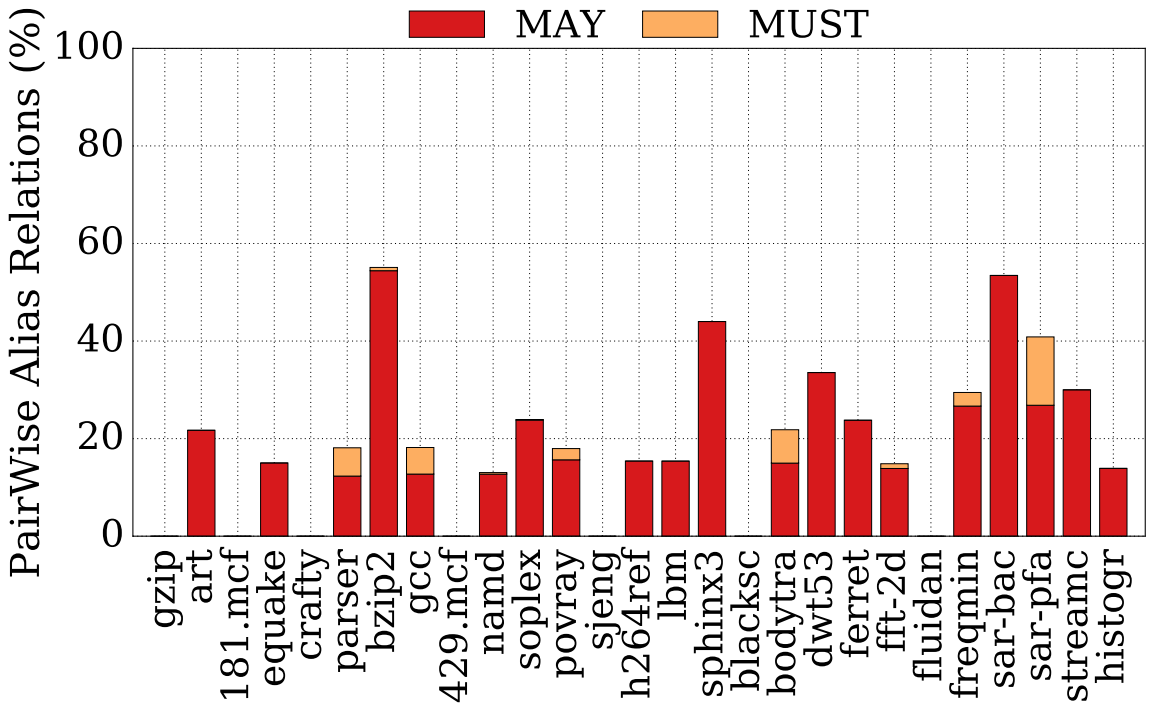


Figure 5.8: Stage 3 : Impact of simplification on alias dependencies for top five accelerated paths. Top 5 paths.

inspected the source of the accelerator region with the highest coverage to understand the reason for poor aliasing information. We found that for those workloads the standard alias analysis 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] [2], [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]...`

5.3 *NACHOS*: Energy Efficient Memory Disambiguation

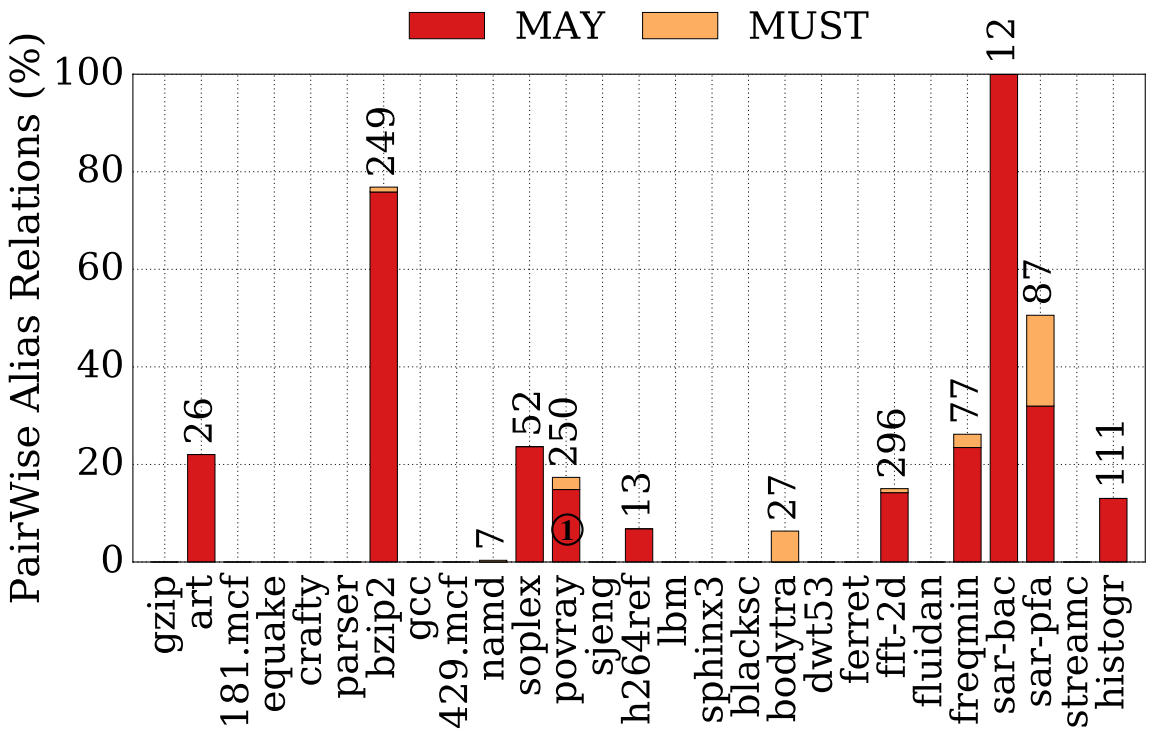
In this section we apply *NACHOS* to the accelerated region (path) with highest dynamic coverage and evaluate the energy benefits relative to an optimized LSQ (LSQ-OPT in Figure 4.4). Table 4.1 lists the features of the accelerated path.

NACHOS imposes 17% overhead on compute energy for 12 workloads. For 15 workloads it imposes no overhead on the compute energy. Overall, NACHOS achieves 50% energy efficiency over an optimized LSQ implementation. *NACHOS* achieves energy efficiency by

- eliminating LSQ checks for alias relations when ordering *MUST* be enforced. It uses an *ORDER* edge (single bit) instead of hardware disambiguation.

[2] 444.namd required minor modifications to the source.

- eliminating checks entirely when the compiler proves memory operations can be run in parallel
- eliminating checks entirely when data dependencies ensure the operations cannot be run in parallel,
- decomposing memory disambiguation into pairwise checks that can be achieved in a distributed fashion.



$\frac{Mem.Dep}{DataDep}$	Applications
≃0%	gzip, 181.mcf, equake, crafty, parser, gcc, mcf, sjeng, lbm, sphinx3, blackscholes, dwt53, ferret, fluidanimate, streamcluster, sar-backprojection
1-25%	444.namd, 464.h264ref, freqmine, 450.soplex, bodytrack, histogram, fft-2d
25-87%	art, bzip2, fft-2d, freqmine, histogram, povray, sar-pfa-, soplex

Figure 5.9: Number of alias relations which need to be enforced to maintain memory ordering. The percentage is relative to all pairwise alias relationships. Every MAY relation enforced as MDE expends energy at runtime; hardware comparator checks if memory addresses overlap; every MUST edge incurs link energy for ordering the two operations.

Figure 5.9 shows the pairwise alias relations which need to be enforced as a fraction of all pairwise alias relations. Each bar is segmented into *MUST* and *MAY* relations. The final aliasing relations obtained employ the three stage *NACHOS* analysis as well as the polyhedral alias analysis for workloads with stencil memory access patterns. In the workloads where MDEs were introduced, between 7–296 new edges were added. Three workloads, povray, bzip2 and fft-2d – (1) – required

more than 250 MDEs. For fft-2d and povray, this represents enforcing less than 20% of all pairwise alias relationships. Overall a average of 54 MDEs were added to workloads where they were required.

We also summarize the introduction of MDEs as a fraction of existing data dependencies for the workloads we study. We find that in most workloads, 16 of 27, the number of edges introduced to enforce memory dependencies represents less than 1% percent of existing data dependence edges. In 7 workloads the percentage of MDEs added vary from 1–25%. For the remaining four workloads, we augment the dataflow graph with 25%–87% extra edges to enforce memory ordering.

***NACHOS* vs LSQ-OPT:** Figure 5.10 shows the overall energy breakdown of the *NACHOS* architecture. Each workload bar is normalized to the energy consumption of the LSQ-OPT hardware memory disambiguation approach (see Figure 4.4). We describe our overall simulation infrastructure in Chapter 5.7.

Across workloads, we find a geomean reduction in energy of 50.4%. The largest improvement in energy occurs for freqmine – (1) – with a reduction of 70%. For freqmine, *NACHOS* can determine *NO* alias relations for more than 70% of all memory operations, adding only 77 MDEs for 32 memory operations. The MDE's impose a 44% overhead on existing compute operations. Ferret and blackscholes – (2) – have no memory operations in the selected path. While they have the same dynamic energy consumption of the LSQ-OPT approach, *NACHOS* incurs no extra area or static power overheads. Overall, *NACHOS* is 31–70% more energy efficient than the LSQ-OPT approach.

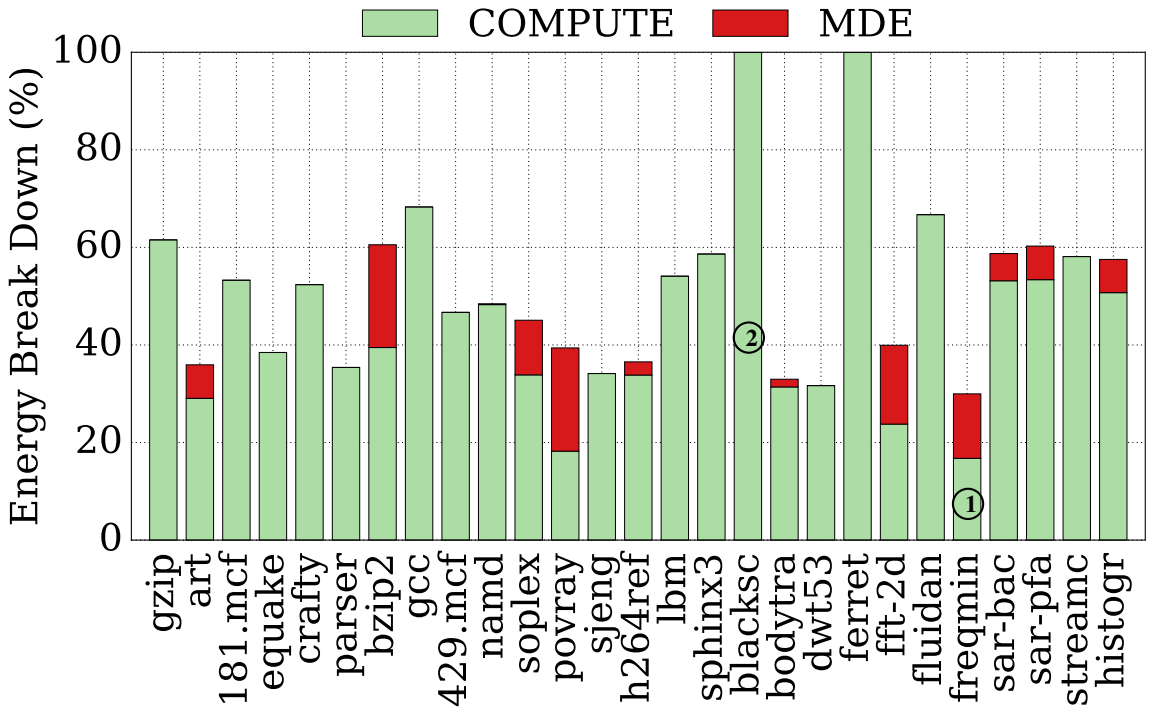


Figure 5.10: *NACHOS* Energy Breakdown (COMPUTE, MDE). Normalized to total energy of LSQ-OPT (Figure 4.4)

We choose three workloads to illustrate how the different stages of alias analysis and simplification lead to the minimal addition of MDEs. We also indicate the related workloads, where a similar effect is observed.

sjeng (Efficacy of Stage 1) : sjeng has 99 ops (11 memory) in the specialized region. Of the 11 memory operations, only a single operation is a store. The stage 1 of *NACHOS* analysis can 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 66%, by enforcing exactly the dependencies which need to be enforced. Figure 5.5 shows the same trend true for not just the most frequently executed region (11% of dynamic executed operations), but also for the top five most frequently executed regions. In all they account for 10% of the dynamic executed operations. ***Related :*** gzip, mcf, crafty, mcf, fft-2d

fluidanimate (Efficacy of Stage 2) :

We find a 33% 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* can 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 aliasing checks. ***Related :*** gcc, parser, h264ref, sar-*, histogram, freqmine

Histogram (Efficacy of Stage 3) :

Stage 1 pairwise alias analysis potentially introduces a significant number of *MAY* edges. Many of these edges need not be enforced due to the existing dependence relationships which exist in the dataflow graph. We analyze the transitive relationships between edges. Figure 5.9 includes the absolute number of pairwise edges which need to be enforced. The simplification pass removes 1293 of 1404 (93%) potential MDEs for the most frequently executed region (represents 70% of the dynamically executed code); This amounts to adding an extra 43% additional edges with respect to the original dataflow graph. ***Related :*** art, bzip2, soplex, povray, h264ref, sphinx3, ferret, fft-2d, freqmine, sar-backprojection, sar-pfa-interp1, streamcluster

Polly analysis :

For 5 of 27 workloads, we use polyhedral alias analysis from LLVM 3.8. This is useful when the standard alias analysis are confounded by multidimensional array accesses. In this section, the workloads that require polyhedral analysis are equake, lbm, namd, bodytrack and dwt53.

5.4 *NACHOS*-Conservative: Is compiler-only disambiguation sufficient?

Compiler only alias analysis is insufficient in many workloads. If we simply enforced all compiler specified orderings (i.e., treat MAYs as MUST) 9 workloads experienced a slowdown $> 2\times$. Hardware-based checks of MAY alias edges is essential to improve MLP. In *NACHOS* all *MAY* MDEs perform runtime memory disambiguation using pairwise comparator checks which require energy but ensure performance is competitive with an LSQ-based approach. In some workloads, such pairwise checks are energy intensive compared to compute energy (even if overall *NACHOS* is more efficient than an LSQ) (e.g., bzip2: 30% of compute energy; povray: $2\times$; equake: 50%).

We perform an experiment in which we convert all *MAY* edges to *ORDER* edges and enforce ordering assuming the operations alias; This will eliminate the runtime checks and *NACHOS* will impose no energy overhead for memory disambiguation. However, the extra ordering edges may increase the critical path of the accelerator leading to a performance degradation. We compare against the performance of the original dataflow graph with only the minimum memory orderings enforced. Table 5.2 summarizes the results. For 18 out of 25 workloads, we do not increase the latency of the critical path. We find that for these workloads, the critical path does not contain any MDEs. Though some MDEs may have been added to the workload's dataflow graph (e.g., sar-backprojection 3% extra edges) they did not feature in the critical path. In these applications, there is little energy gain as well, since the compiler has managed to determine both non-aliasing and aliasing operations.

Table 5.2: Performance. *NACHOS*-conservative (vs Ideal DFG)

Perf. Drop (Cycles)	App	# MDEs /Path	Crit. Path #ops ×	% MDE
0	gzip, 181.mcf, crafty, parser, gcc, 429.mcf, sjeng, sphinx3, blacksch, ferr, fluidani, sar-back, streamclu, namd, lbm, bodytr., dwt53, equake	≃0	0	0
5×	sar-pfa, bzip2, h264ref, art, soplex, freqmine, histogr,	7%	4×	12%
19×	fft, povray	22	11×	35%
% MDEs: Fraction of memory dependencies in overall dataflow graph which includes memory and instruction dependencies. # MDEs/Path: Number of memory dependencies in the critical path				

For seven workloads (e.g., art, bzip2), the increase in critical path latency due to enforcing all the MDEs as *ORDER* edges was severe; critical path length increased by $4\times$. Overall performance (cycle time) reduced by a factor of $5\times$. For these applications, the MDEs represent 12% of the dataflow edges (which includes memory and instruction dependencies). At least one or more MDE is part of the critical path and enforcing these orderings leads to a multiplicative effort that the overall length of critical path increases; latency increase is worse since the memory operations latency will delay the non-memory operations. Even enforcing a few MDEs can dramatically increase the length of the critical path. In art and h264ref, only three occur on the critical path. However, this led to a critical path increase of $3.1\times$ and $3.9\times$ respectively. For freqmine, the length of the critical path is 29 operations out of which enforcing unnecessary dependencies meant 19 operations were memory operations.

In 2 workloads (fft-2d, povray) with many memory operations and MAY edges, 296 and 250 respectively (Figure 5.9), the critical path increased by 11× and overall performance dropped by 20×. In such workloads memory dependencies account for over ≃35% of total dependencies. The severe degradation is caused due to repeated memory indirect addressing. For example, in povray (sphere.cpp:297) it is not feasible for the compiler to reason about aliasing statically. Additionally, the operations contained within the MDE chains formed by *NACHOS* were long latency floating point operations or other memory operations. For povray, the length of the critical path was 92 operations where the number of MDEs in the path was 30. On average, there were two floating point operations and one memory operation between each MDE. Similarly, for fft-2d, there were four floating point operations and one memory operation between memory operations.

5.5 *NACHOS* and *NACHOS*-Conservative: Number of Fan-ins to a memory node

Since, *NACHOS* compiler alias analysis performs a pairwise check between each memory operation, there can be cases where there are a lot of MDEs incident to a memory operation in the worst case scenario. If these MDEs are true dependencies, then the hardware can do a sequential check to each MDE to reduce energy cost. However, this can be an issue if most of the dependencies issued by MDEs are false dependencies. Introducing false dependencies can degrade performance as well as increase energy costs. To get an idea of the number of MDEs incident to a memory operation, we create a histogram of MDEs and bin them based on the logarithmic value of 2 of the number of MDEs (i.e., log2(#MDE)). In Table 5.3, for example, bin-2 is the bin which contains all the nodes which had less than 4 MDEs and greater than 2 MDEs incident. Across 27 benchmarks of varying memory operations in an offloaded function, most of the memory operations are incident to less than 2 MDEs. In the worst case, out of 110 memory operations of bzip only one memory operation has more than 64 incident edges. Similarly, out of 215 memory operations in equake only one memory operation is incident to a maximum of 32 MDEs.

5.6 Memory consistency in *NACHOS*

Interestingly, LSQs in conventional processors have also been the sites for enforcing memory consistency. For strong consistency models such as TSO (total store order), *NACHOS* would simply enforce MDEs between the sequence of stores in the program order ensuring that stores are all ordered. Note that in such cases the loads can still slip past the stores with which they don't alias. To handle fences in TSO and weaker consistency models, *NACHOS* treats the fence like a dummy memory operation and introduces MDEs to implement the fence. For instance, for a mfence it will simply introduce a memory dependency edge between all pre-fence memory operations in the program and the fence operation and from the fence to all post-fence operations; enforcing the execution schedule Pre-fence memory operations → Fence (dummy op) → Post-fence memory

Table 5.3: MDE Fan-ins incident to a node in *NACHOS* generated dataflow graph

app	memop	bin-0	bin-1	bin-2	bin-3	bin-4	bin-5	bin-6
gzip	4	-	-	-	-	-	-	-
art	36	17	9	-	-	-	-	-
181.mcf	2	-	-	-	-	-	-	-
equake	215	178	2	1	1	1		
crafty	7	-	-	-	-	-	-	-
parser	12	-	-	-	-	-	-	-
bzip2	110	77	4	1	1	1	1	1
gcc	2	-	-	-	-	-	-	-
429.mcf	3	-	-	-	-	-	-	-
namd	100	74	1	3	2	-	-	-
soplex	32	25	3	1	-	-	-	-
povray	74	70	1	4	1	-	-	-
sjeng	11	1	-	-	-	-	-	-
h264ref	42	11	3	-	-	-	-	-
lbm	57	34	3	3	1	-	-	-
sphinx3	20	-	-	-	-	-	-	-
blackscholes	0	-	-	-	-	-	-	-
bodytrack	42	35	1	1	-	-	-	-
dwt53	16	15	4	4	-	-	-	-
ferret	0	-	-	-	-	-	-	-
fft-2d	80	75	1	1	2	-	-	-
fluidanimate	28	-	-	-	-	-	-	-
freqmine	32	25	1	2	1	-	-	-
sar-backprojection	7	3	1	3	-	-	-	-
sar-pfa-interp1	32	27	1	1	2	-	-	-
streamcluster	32	-	-	-	-	-	-	-
histogram	48	30	2	1	1	-	-	-

operations. Finally, our acceleration regions are enclosed in synchronization boundaries, i.e., our compiler does not permit offloaded regions to cross-synchronization boundaries (e.g., pthread lock and unlock calls) ensuring overall program correctness.

5.7 Simulation Infrastructure

We have developed a detailed cycle-accurate simulator that models the host core, the *NACHOS* accelerator, and spatial accelerator. Our compiler generates a binary with two components: the x86 executable for the cold paths and the dataflow graph for the offload accelerator. We model a spatial homogeneous fabric accelerator similar to [7, 31]. To model the accelerator we traverse the activity of the program dataflow graph cycle-by-cycle, generating any requisite memory operations in a cycle and stalling the appropriate operations as necessary. The host OoO core pipeline is modeled using MacSim [46]. We assume that *NACHOS* is an accelerator that communicates with the OoO core via the shared L2 cache. The memory hierarchy is modeled using Ruby [24]. We assume an aggressive non-blocking interface to memory. To model the power consumption, we adopt an event-based power model similar to Aladdin [40]. Table 5.4 shows the characteristics of the architectures that we model.

Table 5.4: System parameters

Host Core	2 GHz, 4-way OoO, 96 entry ROB, 4 INT, 4 FPU, INT RF (64 entries), FP RF (64 entries) 32 entry load queue, 32 entry store queue
L1	64K 4-way D-Cache, 3 cycles
LLC Memory	4M shared 16 way, 8 tile NUCA, ring, avg. 25 cycles. Directory MESI coherence. 200 cycles.
Accelerator	
CGRA16	16× 16 function units or
Energy Parameters (Static and Dynamic)	
OoO CGRA LSQ	Mcpat [19]; ARM A9 2Ghz template. CGRA Network (600 fJ/link), Function units (500 fJ/INT,1500 pJ/FP) Memory dependency edge. May: 500 fJ/edge Must: 250 fJ /edge 48 entries/bank 2 ports. 1—4 banks. Loads: 5000 fJ Stores: 7500 fJ Bloom Filter: 2500fJ. 512 entries counting.

5.8 Related Work

5.8.1 Architecture

The focus of much research in LSQ has been to either reduce the content-addressable LSQ checks(e.g., [36]) or eliminate it entirely (e.g., [39]). Research that focused on filtering accesses [4, 26, 36] may require additional hardware structures for filtering, predict when to filter, and RAM structures for data forwarding with additional complexity and area. Sha et al. have proposed to employ prediction to pairwise match up potentially aliasing loads and stores to eliminate the power-hungry searches [38]. Fire-and-Forget [44] and NoSQ [39] both proposed methods for removing the store queue by forwarding values to the loads. Both proposals use sophisticated dependence predictors [4] and may require multi-ported RAM structures. A promising scheme is the use of two level load-store queues [2] or bloom filters [36] to filter load store queue filters accesses. A popular approach is banking [37, 43] which optimizes for latency and power. As we demonstrate even a banked LSQ with a two-level filter introduces significant energy overhead in a hardware accelerator. Huang and Huang [14] used best-effort binary instrumentation to filter out loads from the LSQ that are guaranteed to be safe; the LSQ is required to enforce ordering among all other memory operations.

NACHOS demonstrates that for hardware accelerators, a compiler can completely drive the memory disambiguation and eliminate the LSQs as opposed to being best effort strategies (e.g., [4, 14]). We leverage the compiler to find MLP accurately as opposed to speculatively finding it [44]. Similar to memory cloaking and bypassing [26] we directly forward data between ST-LD pairs using dataflow dependencies but we leverage the compiler to form ST-LD pairs accurately. Finally, we use pairwise hardware checks to handle memory dependencies when the compiler is unsure.

5.8.2 Alias Analysis in Compilers

NACHOS is the first work to directly leverage alias analysis for driving memory disambiguation and builds on extensive research within the compiler community. Static *alias analysis* determines whether two different pointers in a program may point to the same object when a program executes. In contrast, *pointer analysis* or *points-to* analysis identifies the set of objects to which a pointer may point. By helping to determine which operations on pointers may affect each other, these studies provide a foundation for many optimizations [3, 12, 41]. *NACHOS* leverages these approaches for memory disambiguation. Others have leveraged static analysis for a seemingly related but different problem, program parallelization. Many research papers have exploited static analysis to extract threads [6, 51] from sequential programs. These works have largely been hampered by may alias relationships.

Identifying pointers that provably *must not alias* is key to finding MLP at compile-time itself. This same notion of pruning conflicting accesses via must-not-alias information has also been used for precise data race detection by Naik and Aiken [27]. Indeed, the formulation that we use for aliasing accesses matches the traditional definition of a data race in program analysis. Must-not-alias

analysis has also been used to improve the general efficiency of CFL-based alias analysis [50]. Some analysis instead exploit the dynamic aliasing relationships present in the running software. We find static analysis to be sufficient for memory disambiguation in hardware accelerators. Mock et al. determined that most dynamic points-to sets were small in practice (of size 1, 98% of the time), and optimizations based on dynamic alias analysis improves over static analysis [25]. The predictability of aliasing behavior from profiles has led to other works that speculatively exploit possible aliasing relationships [5, 20].

Chapter 6

Conclusion

We present *NACHOS*; a compiler-assisted approach to memory disambiguation for hardware accelerators. *NACHOS* leverages compiler alias analysis and exploits the limited execution window of hardware accelerators to perform compile time memory disambiguation and extraction of memory level parallelism (MLP). *NACHOS* frees accelerator designers from having to identify memory aliasing operations manually or use hardware memory disambiguation. It is a generalized approach that enables hardware accelerator designs to be more broadly employed in programs. Finally, it eliminates the need for a Load Store Queue (LSQ), while consuming 50% less energy at comparable performance.

6.1 Future Work

In Stage 2 of Alias analysis, we observed that we could improve alias analysis by tracing the provenance of the pointers back across one function call boundary. We could find out – how many levels of function calls we can trace back to find meaningful alias information. There is a trade-off in terms of the amount of work required for alias analysis in profiling stage vs. meaningful alias information that adds to energy reduction. Currently, transitivity property in Stage 3 helps to reduce the number of edges significantly. It will be interesting to look at benchmarks where this is not the case i.e. benchmarks with wide dataflow graphs, and explore how alias analysis can be used to improve energy and performance in such cases.

We only looked at the hottest path for our energy evaluation. However, we provide alias information for the top five hot paths. We can evaluate if it is feasible to build hardware for the top five hot paths. In case of multiple Store Load dependencies we convert Forwarding Edges to Ordering Edges, we can research other avenues to improve performance. We also need to explore efficient routing and placement of dependent memory operations to reduce the link energy.

Bibliography

[1] Altera opencl. https://www.altera.com/products/designsoftware/embedded-software-developers/opencl/.

[2] Miquel Peric as, Adrian Cristal, Francisco J Cazorla, Ruben Gonz alez, Alex Veidenbaum, Daniel A Jim 'enez, and Mateo Valero. A Two-Level Load/Store Queue Based on Execution Locality. In *PROC of the 35th ISCA*, 2008.

[3] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, PLDI '90, pages 296–310, New York, NY, USA, 1990. ACM.

[4] George Z Chrysos and Joel S Emer. Memory dependence prediction using store sets. In *ACM SIGARCH Computer Architecture News*, volume 26, pages 142–153. IEEE Computer Society, 1998.

[5] Jeff Da Silva and J. Gregory Steffan. A probabilistic pointer analysis for speculative optimizations. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 416–425, New York, NY, USA, 2006. ACM.

[6] Gina Goff, Ken Kennedy, and Chau-Wen Tseng. Practical dependence testing. In *PROC of the 1991 PLDI*, 1991.

[7] Venkatraman Govindaraju, Chen-Han Ho, and Karthikeyan Sankaralingam. Dynamically specialized datapaths for energy efficient computing. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, pages 503–514. IEEE, 2011.

[8] Tobias Grosser, Armin Größlinger, and Christian Lengauer. Polly - performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 22(4), 2012.

[9] Shantanu Gupta, Shuguang Feng, Amin Ansari, Scott Mahlke, and David August. Bundled execution of recurring traces for energy-efficient general purpose processing. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 12–23. ACM, 2011.

[10] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. EIE: Efficient Inference Engine on Compressed Deep Neural Network. In *Proc. of the 43rd ISCA*, pages 243–254, 2016.

[11] Mitchell Hayenga, Vignyan Reddy Kothinti Naresh, and Mikko H Lipasti. Revolver: Processor architecture for power efficient loop execution. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 591–602. IEEE, 2014.

[12] Michael Hind. Pointer analysis: Haven't we solved this problem yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '01, pages 54–61, New York, NY, USA, 2001. ACM.

[13] Chen-Han Ho, Sung Jin Kim, and Karthikeyan Sankaralingam. Efficient execution of memory access phases using dataflow specialization. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 118–130. ACM, 2015.

[14] R Huang, A Garg, and M Huang. Software-hardware cooperative memory disambiguation. In *PROC of the 12th HPCA*, 2006.

[15] Wen-Mei W Hwu, Scott A Mahlke, William Y Chen, Pohua P Chang, Nancy J Warter, Roger A Bringmann, Roland G Ouellette, Richard E Hank, Tokuzo Kiyohara, Grant E Haab, et al. The superblock: an effective technique for vliw and superscalar compilation. *the Journal of Supercomputing*, 7(1-2):229–248, 1993.

[16] Engin Ipek, Meyrem Kirman, Nevin Kirman, and Jose F Martinez. Core fusion: accommodating software diversity in chip multiprocessors. In *PROC of the 34th ISCA*, 2007.

[17] Snehasish Kumar, Nick Sumner, Steven Magrem, Viji Srinivasam, , and Arrvindh Shriraman. Needle : Leveraging program analysis to analyze and extract accelerators from whole programs. In *Proc. of the 18th Intl. Symp. on High Performance Computer Architecture*, HPCA, 2017.

[18] Snehasish Kumar, Nick Sumner, and Arrvindh Shriraman. Spec-ax and parsec-ax: Extracting accelerator benchmarks from microprocessor benchmarks. In *Proc. of the Intl. Symp. on Workload Characterization*, IISWC, 2016.

[19] Sheng Li, Jung Ho Ahn, Richard D Strong, Jay B Brockman, Dean M Tullsen, and Norman P Jouppi. McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *PROC of the 42nd MICRO*, 2009.

[20] Jin Lin, Tong Chen, Wei-Chung Hsu, Pen-Chung Yew, Roy Dz-Ching Ju, Tin-Fook Ngai, and Sun Chan. A compiler framework for speculative optimizations. *ACM Trans. Archit. Code Optim.*, 1(3):247–271, September 2004.

[21] Feng Liu, Heejin Ahn, Stephen R Beard, Taewook Oh, and David I August. Dynaspam: dynamic spatial architecture mapping using out of order instruction schedules. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 541–553. IEEE, 2015.

[22] Michael J Lyons and David Brooks. The design of a bloom filter hardware accelerator for ultra low power systems. In *Proceedings of the 2009 ACM/IEEE international symposium on Low power electronics and design*, pages 371–376. ACM, 2009.

[23] Divya Mahajan, Jongse Park, Emmanuel Amaro, Hardik Sharma, Amir Yazdanbakhsh, Joon Kyung Kim, and Hadi Esmaeilzadeh. TABLA: A unified template-based framework for accelerating statistical machine learning. In *Proc. of the 22nd HPCA*, pages 14–26, 2016.

[24] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. Multifacet's general execution-driven multiprocessor simulator (gems) toolset. *SIGARCH Comput. Archit. News*, 33(4):92–99, November 2005.

[25] Markus Mock, Manuvir Das, Craig Chambers, and Susan J. Eggers. Dynamic points-to sets: A comparison with static analyses and potential applications in program understanding and optimization. Technical report, March 2001.

[26] Andreas Moshovos and Gurindar S Sohi. Speculative Memory Cloaking and Bypassing. *International Journal of Parallel Programming*, 1999.

[27] Mayur Naik and Alex Aiken. Conditional must not aliasing for static race detection. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '07, pages 327–338, New York, NY, USA, 2007. ACM.

[28] Tony Nowatzki, Vinay Gangadhar, and Karthikeyan Sankaralingam. Exploring the potential of heterogeneous von neumann/dataflow execution models. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 298–310. ACM, 2015.

[29] Tony Nowatzki and Karthikeyan Sankaralingam. Analyzing Behavior Specialized Acceleration. In *Proc. of the 21st ASPLOS*, pages 697–711, 2016.

[30] Shruti Padmanabha, Andrew Lukefahr, Reetuparna Das, and Scott A Mahlke. Trace based phase prediction for tightly-coupled heterogeneous cores. In *Proc. of the 46th MICRO*, pages 445–456, 2013.

[31] Hyunchul Park, Yongjun Park, and Scott Mahlke. Polymorphic pipeline array: a flexible multicore accelerator with virtualized execution for mobile multimedia applications. In *PROC of the 42nd MICRO*, 2009.

[32] Ganesan Ramalingam. The undecidability of aliasing. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(5):1467–1471, 1994.

[33] Elham Safi, Andreas Moshovos, and Andreas Veneris. L-cbf: a low-power, fast counting bloom filter architecture. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 16(6):628–638, 2008.

[34] S Sethumadhavan, R McDonald, D Burger, S S W Keckler, and R Desikan. Design and Implementation of the TRIPS Primary Memory System. In *Computer Design, 2006. ICCD 2006. International Conference on*, pages 470–476, 2006.

[35] Simha Sethumadhavan, Doug Burger, and Stephen W Keckler. Partition the banks, not the functionality, of large-window load-store queues. 2006.

[36] Simha Sethumadhavan, Rajagopalan Desikan, Doug Burger, Charles R Moore, and Stephen W Keckler. Scalable Hardware Memory Disambiguation for High ILP Processors. In *PROC of the 36th MICRO*, 2003.

[37] Simha Sethumadhavan, Franziska Roesner, Joel S Emer, Doug Burger, and Stephen W Keckler. Late-binding: enabling unordered load-store queues. In *PROC of the 34th ISCA*, 2007.

[38] Tingting Sha, Milo M K Martin, and Amir Roth. Scalable Store-Load Forwarding via Store Queue Index Prediction. In *PROC of the 38th MICRO*, 2005.

[39] Tingting Sha, Milo M K Martin, and Amir Roth. NoSQ: Store-Load Communication without a Store Queue. In *PROC of the 39th MICRO*, 2006.

[40] Yakun Sophia Shao, Brandon Reagen, Gu-Yeon Wei, and David M Brooks. Aladdin: A pre-RTL, power-performance accelerator simulator enabling large design space exploration of customized architectures. In *Proc. of the 41st ISCA*, pages 97–108, 2014.

[41] Yannis Smaragdakis and George Balatsouras. Pointer analysis. *Found. Trends Program. Lang.*, 2(1):1–69, April 2015.

[42] Aaron Smith, Jon Gibson, Bertrand A Maher, Nicholas Nethercote, Bill Yoder, Doug Burger, Kathryn S McKinley, and James H Burrill. Compiling for EDGE Architectures. *CGO*, pages 185–195, 2006.

[43] Sam S Stone, Kevin M Woley, and Matthew I Frank. Address-Indexed Memory Disambiguation and Store-to-Load Forwarding. In *PROC of the 38th MICRO*, 2005.

[44] Samantika Subramaniam and Gabriel H Loh. Fire-and-Forget: Load/Store Scheduling with No Store Queue at All. In *PROC of the 39th MICRO*, 2006.

[45] Steven Swanson, Ken Michelson, Andrew Schwerin, and Mark Oskin. WaveScalar. In *PROC of the 36th MICRO*, 2003.

[46] Georgia Tech. Macsim : Simulator for heterogeneous architecture - https://code.google.com/p/macsim/.

[47] Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson, and Michael Bedford Taylor. Conservation cores: reducing the energy of mature computations. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 205–218. ACM, 2010.

[48] Dani Voitsechov and Yoav Etsion. Single-graph multiple flows: Energy efficient design alternative for gpgpus. *ACM SIGARCH Computer Architecture News*, 42(3):205–216, 2014.

[49] Lisa Wu, Raymond J Barker, Martha A Kim, and Kenneth A Ross. Navigating big data with high-throughput, energy-efficient data partitioning. In *PROC of the 40th ISCA*, 2013.

[50] Guoqing Xu, Atanas Rountev, and Manu Sridharan. Scaling cfl-reachability-based points-to analysis using context-sensitive must-not-alias analysis. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, Genoa, pages 98–122, Berlin, Heidelberg, 2009. Springer-Verlag.

[51] Antonia Zhai, Christopher B Colohan, J Gregory Steffan, and Todd C Mowry. Compiler optimization of scalar value communication between speculative threads. In *PROC of the 10th ASPLOS*, 2002.

[52] Qirun Zhang, Michael R Lyu, Hao Yuan, and Zhendong Su. Fast algorithms for dyck-cfl-reachability with applications to alias analysis. In *ACM SIGPLAN Notices*, volume 48, pages 435–446. ACM, 2013.

[53] Xin Zheng and Radu Rugina. Demand-driven alias analysis for c. *ACM SIGPLAN Notices*, 43(1):197–208, 2008.