UNIVERSITY OF SOUTHAMPTON

FACULTY OF PHYSICAL AND APPLIED SCIENCES

School of Electronics and Computer Science

Memory and Functional Unit Design for Vector Microprocessors

by

Matthias Boettcher

Thesis for the degree of Doctor of Philosophy

May 2014

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF PHYSICAL AND APPLIED SCIENCES
School of Electronics and Computer Science

Doctor of Philosophy

MEMORY AND FUNCTIONAL UNIT DESIGN FOR VECTOR MICROPROCESSORS

by Matthias Boettcher

Modern mobile devices employ SIMD datapaths to exploit small scale data-level parallelism to achieve the performance required to process a continuously growing number of computation intensive applications within a severely energy constrained environment. The introduction of advanced SIMD features expands the applicability of vector ISA extensions from media and signal processing algorithms to general purpose code. Considering the high memory bandwidth demands and the complexity of execution units associated with those features, this dissertation focuses on two main areas of investigation, the efficient handling of parallel memory accesses and the optimization of vector functional units.

A key observation, obtained from simulation based analysis on the type and frequency of memory access patterns exhibited by general purpose workloads, is the tendency of consecutive memory references to access the same page. Exploiting this and further observations, Page-Based Memory Access Grouping enables a level one data cache interface to utilize single-ported TLBs and cache banks to achieve performance similar to multi-ported components, while consuming significantly less energy. Page-Based Way Determination extends the proposed scheme with TLB-coupled structures holding way information on recently accessed lines. These structures improve the energy efficiency of the vast majority of memory references by enabling them to bypass tag-arrays and directly target individual cache ways.

A vector benchmarking environment - comprised of a flexible ISA extension, a parameterizable simulation framework and a corresponding benchmark suite - is developed and utilized in the second part of this thesis to facilitate investigations into the design aspects and potential performance benefits of advanced SIMD features. Based on it, a set of microarchitecture optimizations is introduced, including techniques to compute hardware interpretable masks for segmented operations, partition scans to allow specific energy - performance trade-offs, re-use existing multiplexers to process predicated and segmented vectors, accelerate scans on incomplete vectors, efficiently handle micro-ops fully comprised of predicated elements, and reference multiple physical registers within individual operands to improve the utilization of the vector register file.

Contents

List of Figures

List of Tables

Declaration of Authorship

I, Matthias Boettcher , declare that the thesis entitled *Memory and Functional Unit Design for Vector Microprocessors* and the work presented in the thesis are both my own, and have been generated by me as the result of my own original research. I confirm that:

- this work was done wholly or mainly while in candidature for a research degree at this University;
- where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated;
- where I have consulted the published work of others, this is always clearly attributed;
- where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work;
- I have acknowledged all main sources of help;
- where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself;
- parts of this work have been published as: listed in Sec. 1.6

Signed:...

Date:..

Acknowledgements

I would like to thank my academic advisor Professor Bashir M. Al-Hashimi for supporting me throughout this work and for his advice and encouragement while I pursued my interest in microprocessor design. I would also like to thank my industrial advisor Dr. Danny Kershaw who introduced me to ARM and provided initial guidance for the development of MALEC.

I am grateful to the ARM-ECS Research Centre at the School of Electronics and Computer Science (ECS), University of Southampton and ARM Ltd. for providing me with access to cutting edge IP and the opportunity to work next to leading industry experts at ARM R&D. Special thanks to Dr. Giacomo Gabrielli, Dr. Mbou Eyole, Alastair Reid and Stuart Biles for their advice, support, and in particular their contributions to ARGON and VBench. Further thanks to Professor Trevor Mudge (University of Michigan), the visiting students Thomas Sharp and Thomas Manville, and the former colleagues Harsh Kumar and Wojciech Meyer for their advice and coding efforts at various stages of this work.

Thanks to Robbie Berryman and David Horner from D Young & Co LLP for their efforts on translating parts of this thesis into the finest legalese, to Dr. Daryl Bradley and Adam Branscombe from ARM's Patent Department to verify the sanity of the resulting patent applications, and to my colleague Dr. Mbou Eyole for his help in proofreading them for technical correctness.

I am also grateful for the constructive discussions and genuinely great moments shared with me by good friends and colleagues over the last few years. They provided joy, inspiration and insight into the different perspectives of academic and industrial professionals. In addition to those mentioned above, these people include, but are by no means limited to: Jon Storey, Robert Rudolf, Taihai Chen, Sascha Bischoff, Dr. Matt Horsnell, Dr. Jatin Mistry, Dr. Sheng Yang, Matthias Klix, Corinna Reinholz, Maria Ziecher, Thomas Hickel, Benjamin Cruse and Anna Gorzkowska. My deepest thanks to all of you.

Finally, I would like to thank my mother Petra, my brother Christian and his wife Konstanze, for their continued love and support throughout these years. This work is dedicated to them and to the memory of my father Lothar.

Nomenclature

(Pipeline) Stage	Unity of combinational logic and registers that operate on values and preserve them between CPU cycles
(Processing) Step	Subset of operations performed in parallel as part of a computation sequence
(Scan) Part	Subset of consecutive vector elements to be processed concurrently
(Scan) Round	Subset of operations performed in parallel as part of a scan sequences
CompD	The Compute Descriptor Instruction
VBench	Vector Benchmark Suite
$\mathrm{VL}_{\mathrm{max}}$	Max. number of elements per vector register (data type dependent)
ALU	Arithmetic Logic Unit
BW	Bandwidth
CAM	Content Addressable Memory
CMP	Chip Multiprocessor
CPU	Central Processing Unit
FP	Floating Point
FU	Functional Unit
gas	GNU Assembler
gcc	GNU Compiler Collection
gem5	The gem5 Simulator System
HPC	High Performance Computer
ILP	Instruction Level Parallelism
IPC	Instructions Per Cycle
IQ	Issue Queue
ISA	Instruction Set Architecture
JSON	JavaScript Object Notation
L-Type	Double Length (Derivative of Multi-Register Operations; Section 7.7)
L1D	First Level Data Cache
L1I	First Level Instruction Cache
LFU	Least Frequently Used
LLVM	Low Level Virtual Machine
LRR	Least Recently Replaced
LRU	Least Recently Used
LSQ	Load-Store Queue
LSU	Load Store Unit
MALEC	Multiple Access Low Energy Cache
MB	Merge Buffer
MB2	MediaBench2
MBE	Merge Buffer Entry
MSHR	Miss Status Holding Register

OoO	out-of-order
OS	Operating System
PC	Program Counter
PIPT	Physically Indexed, Physically Tagged
ROB	Re-order Buffer
SB	Store Buffer
SIMD	Single Instruction Multiple Data
SMT	Simultaneous multithreading
SPEC	Standard Performance Evaluation Corporation
TLB	Translation Lookaside Buffer
uWT	Micro Way Table
VIPT	Virtually Indexed, Virtually Tagged
VL	Vector Length / Vector Length Register
VLIW	Very Long Instruction Word
VM	Vector Mask / Vector Mask Register
WAR	Write after Read
WAW	Write after Write
WT	Way Table

1 | Introduction

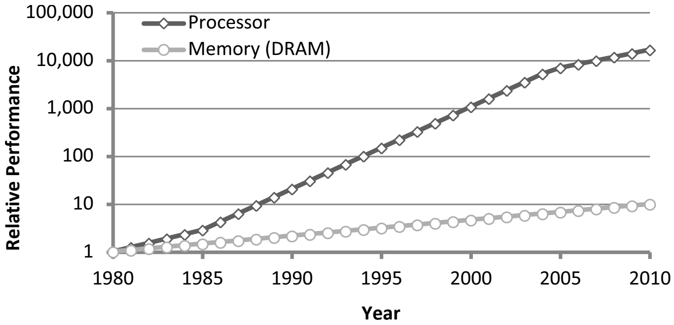


Fig. 1.1: Performance gain relative to 1980

Modern mobile devices exhibit a continuously growing demand to support computation intensive workloads such as multimedia and web applications. In consequence, the performance of mobile devices follows an uptrend similar to the one previously observed for desktop processors. Fig. 1.1 illustrates the performance evolution of Intel processors over the last 30 years relative to 1980 [1]. For the majority of this period, frequency scaling was the driving factor behind annual performance gains of approximately 40%. However, in the late-1990s submicron transistors reached a size where thermal noise and process variations would limit the achievable supply voltage scaling and therefore lead to increased power densities and consequently a higher on-chip temperature. The term "thermal wall" describes the point where temperature reductions due to energy improvements, advanced packaging techniques and thermal cooling become economically infeasible. From this point on, architectural improvements based on the exploration of instruction and data level parallelism became increasingly important. Examples are the progression from single-issue in-order to superscalar out-of-order processors, and the transition from uni- to multi-core systems. In mobile and desktop devices, the potential increase in energy efficiency due to parallelization enables reduced cooling costs, longer battery lifetimes and higher circuit reliability. Conditions for said increase are a high degree of resource utilization and the availability of power efficient states to benefit from shorter computation times.

Memory Technology	Typical Access Time	Price per Gbyte
SRAM	0.5-2.5ns	\$2000-\$50000
DRAM	50-70ns	\$20-\$75
Solid-State Drive	60,000-2,000,000ns	\$0.80-\$4
Magnetic Disk	3,000,000-20,000,000ns	\$0.20-\$2

Tab. 1.1: Comparison of memory technologies

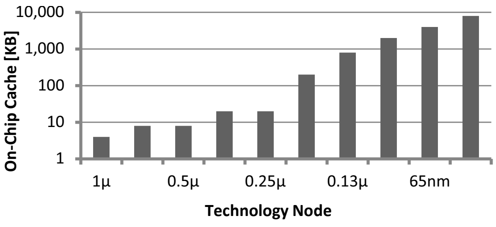


Fig. 1.2: On-chip cache capacity

CPUs capable of computing multiple instructions per cycle (IPC), while operating on a high frequency, require a fast interface between processing and storage structures. However, primary market force for storage structures is the demand for large capacities at minimum cost rather than speed [2]. Fig. 1.1 illustrates this discrepancy by showing processor speedups of approximately 40% per year in contrast to annual memory performance gains of less than

10%. To bridge the gap between CPU and main memory, modern microprocessors depend on up to three levels of small and fast caches that utilize data locality to provide high performance and low latency storage. The comparison in Tab. 1.1 emphasizes the economic background for the decision to use SRAMs, DRAMs and solid-state drives / magnetic disks for memories of increasing capacity and decreasing access speed [3]. Fig. 1.2 illustrates how on-chip cache sizes advance with every new technology node. Increasingly high capacities can be explained by the restricted energy budged of modern processors, which favors performance gains due to energy efficient caches rather than energy-intensive processing logic. Borkar et al. even extrapolate cache capacities of 80 MByte in 2018 [1].

This thesis investigates memory access patterns exhibited by high performance processors and derives techniques to improve the energy efficiency of the level one data cache interface without compromising performance. It furthermore evaluates the performance benefits obtainable by the implementation of advanced SIMD features and proposes microarchitecture optimizations to reduce the energy consumption associated with them. The remainder of this chapter gives background information relevant for the understanding of the subject matter underlying this thesis. Section 1.1 includes an overview of memory hierarchies with a focus on the L1 data memory interface. It is followed by an introduction to caches and their key characteristics (Section 1.2), a summary of concepts related to out-of-order superscalar processors (Section 1.3), and an overview of relevant parallel processing techniques (Section 1.4). The contributions of individual chapters are outlined in Section 1.5, followed by a list of the publications associated with this thesis in Section 1.6.

1.1 Memory Hierarchies and the L1 Data Memory Interface

1.1.1 Memory Hierarchies

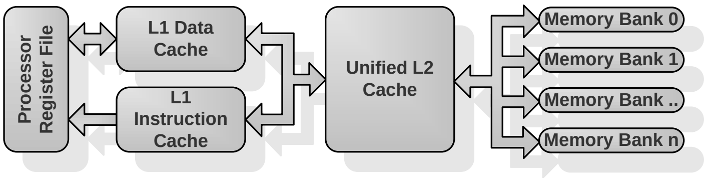


Fig. 1.3: Typical memory hierarchy for microprocessors

Fig. 1.3 depicts a typical memory hierarchy including multiple levels of increasing capacity but decreasing speed (Tab. 1.1). Register files are part of the processor and used to temporarily store data between computations. Located in close proximity of the processor, the L1 cache is split into L1I and L1D to account for the different access patterns of data and instruction streams. In particular, instruction streams exhibit high spatial locality and predictability, which favor fast and simple direct mapped caches (Section 1.2.2). In contrast, data caches usually implement set associative structures for high hit rates in the context of temporal rather than spatial locality. The remainder of this thesis is primarily focused on L1Ds and the processor to L1 interface. The roles of other components will be acknowledged in form of simulation components and parameters, but will not be discussed in greater detail. The next memory level is formed by a unified L2 cache that may be shared by multiple cores for data as well as instruction stream related accesses. It exhibits significantly higher capacity and access latency when compared to L1 structures (e.g. 8-32 KByte compared to 256 KByte - 2 MByte per core). Not shown in Fig. 1.3 are higher level caches, which might be implemented to service local processor clusters. Finally, the main memory itself consists of multiple banks to allow several parallel accesses and reduce energy costs and latencies. The access times for those memories vary from 1 cycle for register files and certain L1 caches to several hundred cycles for the main memory. This clearly underlines the necessity of keeping commonly used data as close as possible to the processor.

The bandwidth (BW) required to interconnect different cache levels depends on the underlying access policies. A look-through policy is suited for caches with low miss rates, as it saves BW by waiting for a response before directing queries triggered by a miss to the next lower cache level. In contrast, a look-aside policy may compensate high miss rates at the cost of BW by immediately dispatching requests to lower levels while waiting for responses. Especially important in the context of multi-core environments are write policies like write-through or write-back. The former updates data in the current as well as the next level and therefore allows simple implementations for the cost of increased BW requirements. Conversely, the latter conserves BW by writing only to the current cache level. However, it needs to track changes to individual lines and requires additional snooping logic to avoid memory coherency issues (Section 1.4.2). While read accesses to uncached elements generally lead to the caching of the corresponding lines (assuming the absence of stream detection mechanisms), write misses may show an alternative behavior. In a no write allocate cache such misses simply bypass the current level to be treated by the next. In contrast, a write allocate cache identifies a line to be replaced (victim; Section 1.2.2), writes it back to and request a refill from a lower level, before it updates the new line. Finally, evicted lines that were modified (dirty) can be treated in two general ways. An inclusive cache holds a full copy of its next lower level to allow the simple overwrite of lines with evicted data. This improves write-back and snooping speeds (Section 1.4.2), but reduces its capacity, requires the eviction of high level data on low level fills, and limits both levels to a common cache line size. In contrast, exclusive caches enforce the condition that data is stored in no more than a single location, but usually require copy-back or victim buffers to hold evicted lines between transfers. Hybrid schemes may trade off characteristics of both approaches by permitting high level data to reside in a low level cache without requiring it to do so.

1.1.2 L1 Data Memory Interface

Key components of a typical L1 data memory interface are depicted in Fig. 1.4. The Load Store Unit (LSU) is a processor internal structure that provides base addresses and offsets to compute virtual addresses. A Translation Lookaside Buffer (TLB) or a Page Table are used to look-up corresponding physical addresses for the resulting values. These addresses are compared to tags stored in the L1D to determine whether the requested cache line is present (cache hit). If this is the case, read requests are directly serviced by feeding

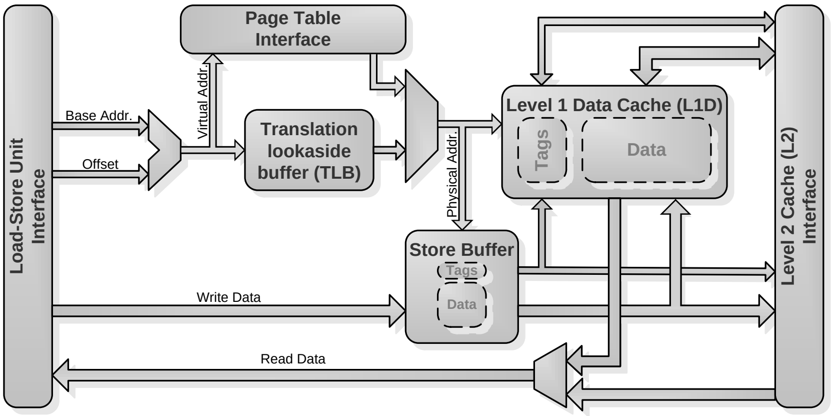


Fig. 1.4: L1 data memory interface

data back into the LSU. Conversely, cache misses might require the eviction of lines before data from lower cache levels can be received. As stores permanently alter architectural state, their handling is subject to design specific policies (Sections 1.1.1 and 1.4.2). For example, out-of-order (OoO) processors often employ Store Buffers (SBs) to temporarily hold information on in-flight stores and allow the forwarding of data to dependent loads (Section 1.3). Hence, stores might be computed ahead of loads to the same address without affecting their outcome. Equivalent structures can be implemented to speculatively execute loads and computations depending on them (i.e. Load Queues). Not illustrated in Fig. 1.4 but employed by the proposal in Chapter 4 are Merge Buffers (MBs), which attempt to reduce the number of L1 accesses by merging data from multiple stores to the same address region. When stores commit, the corresponding SB entry is evicted and sent to the MB, only if this entry cannot be merged or allocated to a free entry, the oldest MB entry is evicted and written to the L1. Loads usually access SB, MB and L1 in parallel to overlap their access latencies. Multiplexers combine the results received from all three sources prioritizing the SB over the MB and the MB over the L1.

The components used for an address computation depend on the corresponding address mode. Most common is the summation of a base address from a specific register or the program counter (PC) and an offset provided by the actual load/store instruction [3]. An adaptation for vector processing (Section 1.4.3) is the implementation of multiple adders to perform several parallel computations. However, in case of non-unit strides this might require multipliers to determine offsets between vector elements and the base address. Furthermore, by determining whether the first and last (for strides) or all (for indexed) accesses are to the same page, the number of address translations might be reduced.

Address translation describes the process of identifying a physical address corresponding to its virtual counterpart. The idea of virtual memory is to provide each program with its own address space, and therefore to allow multiple programs to share one physical memory without any knowledge about each other. Furthermore, by providing a larger virtual than physical address space, programs are effectively allowed to exceed the given main memory capacity; i.e. only currently active code segments need to reside in main memory while inactive intermediate segments can be omitted. Regions of virtual memory are allocated in so called pages and most commonly comprise an address space of 4 KByte (less common are 16 KByte or 64 KByte). Several small pages might be combined within 1 MByte, 2 MByte or 4 MByte regions to simplify management tasks [3, 4] (Intel's Ivy Bridge and ARM's Bulldozer microarchitecture even support 1 GByte pages [5, 6]). The operating system (OS) manages the mapping of pages and tracks them within page tables, located in memory. A common method to organize the virtual address spaces for multiple programs is a root page that contains indexes to a number of user pages. Each user page contains pages corresponding to a specific program [4]. In conclusion, each load/store would require three memory accesses (two to walk the page tables and one to access the data).

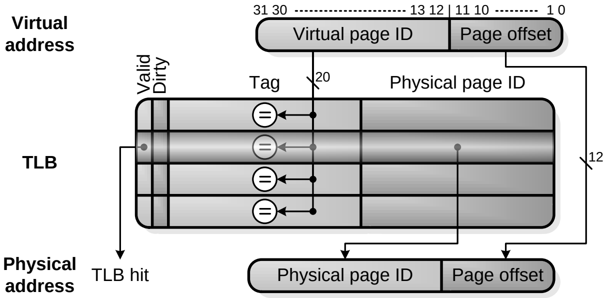


Fig. 1.5: TLB based address translation

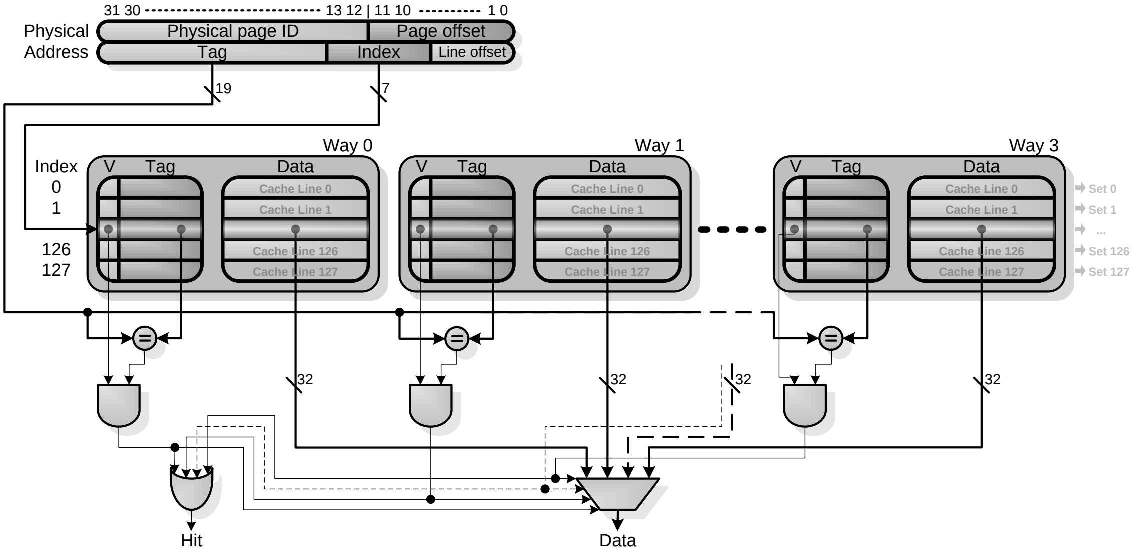
A common way to avoid multiple memory accesses is to cache recently accessed pages within Translation-Lookaside Buffers (TLBs) [3]. These small structures are close to the processor and usually fast enough to be accessed in parallel to caches. To increase the probability of page hits, there are usually separate TLBs for data and instruction caches, as well as bigger, higher level TLBs to back them up. Fig. 1.5 depicts a typical address translation for a 32-bit system with 4 KByte pages. The upper 20 address bits ($32 - \log_2 4096$) are compared to tags held by the TLB. In case of a matching valid entry, the corresponding physical page ID is merged with the original page offset to form the desired physical address. In contrast, a TLB miss results in a table walk and possibly a page fault, if the page is not present in memory. Each TLB entry might contain additional information on access rights, the presences of modified data (dirty bits), and so forth. Chapter 5 introduces Page-Based Way Determination that re-uses TLB results to simplify its own lookup structures.

Of particular importance for out-of-order processors (OoO; Section 1.3) is the requirement on the L1 data memory system to handle exceptions. Exceptions are special conditions that interrupt the normal flow of program execution. In the context of memory operations, exceptions might be raised in consequence of page faults, i.e. attempts to access an address outside a programs own virtual address space. Other exceptions might be triggered by interrupts (e.g. I/O device requests), arithmetic overflows, undefined instructions or user programs invoking the OS via system calls. The handling of exceptions in OoO machines can be "precise" or "imprecise" [7]. Precise exceptions require the processor to establish a state corresponding to the one a sequentially executed program would have at the time the exceptions was raised. Hence, it allows software to resolve problems and resume normal operation afterwards. In contrast, imprecise exceptions are usually irrecoverable and therefore result in the application being terminated. The ability to restore architectural state is relevant for all processor components that feature speculative execution.

1.2 Caches

1.2.1 Basic Concept

The term cache describes a small, low latency structure intended to bridge the disparity of memory capacity and access latency introduced at the begin of this chapter. Caches take advantage of the locality of accesses, by storing a subset of data for faster and more energy efficient accesses. While temporal locality describes the tendency to repeatedly access the same data within a limited amount of time, spatial locality is characterized by a number of consecutive accesses to memory regions in close proximity of each other. The extend to which both types can be observed in the context of general purpose code are investigated in Chapter 3. Observations based on those investigations are then exploited in the subsequent chapters on Page-Based Memory Access Grouping and Page-Based Way Determination.



- Multiplexers for selection of words from cache lines not shown

Fig. 1.6: 4-way set-associative cache

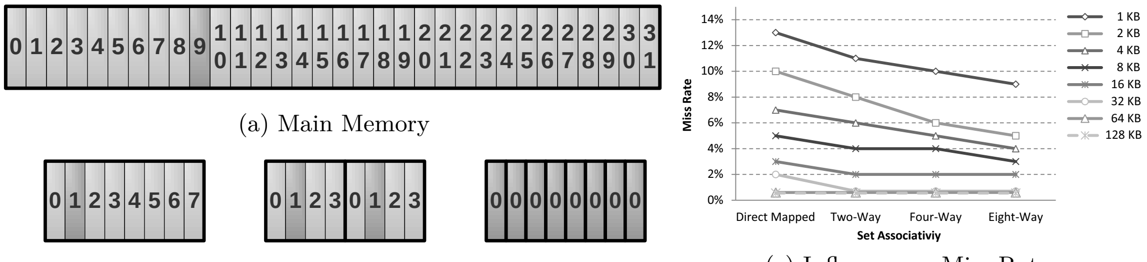
A block diagram of a 4-way set associative cache is depicted in Fig. 1.6. Physical addresses are split into tag, index and line offset [3], and routed to each of the four tag-/data-array pairs (ways). During a conventional read access, the index is used to determined one entry within each tag- and data-array to be activated. Following this, the appropriate tag-array entries are compared to the address tag. The comparison results and corresponding validity bits are then logically combined to identify a hit. Although there are four different ways, i.e. four potential locations for a datum to be stored, a datum can by definition only reside in one way at a time. Hence, the way hit information is used to select the appropriate datum (e.g. a 32-bit word) inside the cache line supplied by the data-arrays.

Note, Fig. 1.6 illustrates a physically indexed, physically tagged (PIPT) cache; i.e. it used physical addresses for both the index and the tag. This type of cache exhibits low complexity and avoids problems such as the mapping of multiple virtual onto one physical address (aliasing) or the mapping of the same virtual address of multiple threads onto different physical addresses (homonyms). However, it does require address translation prior to cache accesses and is therefore primarily implemented by energy oriented systems such as the Cortex-A15 [8] and the simulation framework used in Chapter 4. Virtually indexed, virtually tagged (VIPT) caches are a viable alternative that aim to improve the access latencies of the TLB-L1D subsystem by allowing TLB accesses to proceed in parallel with set-associative cache lookups. TLB lookups only have to be completed before the last step of the cache lookup, i.e. tag comparison and way selection. In order to take advantage of a VIPT scheme, however, all ways of a set-associative cache have to be probed during lookups, which inhibits potential energy savings obtained from activating only a single way. Hence, VIPT designs strongly favor low latency, performance focused systems such as Intel's Ivy Bridge architecture. Section 5.4 specifically compares PIPT and VIPT implementations in the context of Page-Based Way Determination.

Besides the obvious $N = 4$-way associativity, the cache in Fig. 1.6 is based on the following parameters (Note, letters are chosen arbitrarily, capital/lowercase symbols indicate decimal/binary values). An overall capacity C split over S pairs of tags and lines, whereby pairs with the same index inside each way are referred to as set (Section 1.2.2). Lines (also called blocks) form the basic unit of data transfer inside the memory hierarchy. Each line is composed of L consecutive data bytes. Assuming a $C = 32$ KByte cache with $L = 64$ Byte wide lines, the bit-fields tag t, index i and line offset b inside an 32-bit address a result to:

$$\begin{aligned} a &= t + i + b \\ t &= a - i - b \\ &= 32 - 7 - 6 = \underline{19} \end{aligned} \qquad \begin{aligned} b &= l = log_2 L \\ &= log_2 64 = \underline{6} \end{aligned} \qquad \begin{aligned} i &= s = log_2 S \\ &= log_2 128 = \underline{7} \end{aligned} \qquad \begin{aligned} S &= \frac{C}{N \cdot L} \\ &= \frac{32K}{4 \cdot 64} = 128 \end{aligned} \tag{1.1}$$

1.2.2 Placement Policies



(a) Main Memory

(b) Direct-mapped (c) Set associative (d) Fully associative

(e) Influence on Miss-Rates

Fig. 1.7: Data placement policies

Set-associative caches, in contrast to direct-mapped memories, are characterized by the separation of their tag- and data-arrays in two or more parts commonly referred to as ways. A line can be cached at exactly one location within each way. For example, Fig. 1.7 depicts possible placements for element 9 of the main memory within an 8-element wide cache. In contrast to the direct mapped configuration in Fig. 1.7b that restricts the element to exactly one location, the 2-way set-associative variant in Fig. 1.7c allows element 9 at position "9 mod 4 = 1" within both of its parts. Although both cache ways could hold the same data, control circuitry ensures the uniqueness of all cached elements. Fig. 1.7e illustrates the influence of associativity on the miss-rates of different sized caches for SPEC2000 [3]. It can be observed that small caches benefit significantly from an increased associativity. The reason for this is the reduced number of conflicts due to multiple elements mapping to the same location (e.g. $(1 \,||\, 9) \bmod 8 = 1$). Larger caches benefit less from this effect, because the probability for such conflicts reduces with increased capacity. The number of desirable ways is a compromise between achievable hit rate and required complexity. In particular, a fully-associative structure (Fig. 1.7d) can achieve high hit rates for very small memories, but it also introduces significant energy, latency and area overheads. A typical system might implement fully associative TLBs, 2/4-way and 8/16-way set associative L1D and L2 caches, respectively.

Allowing data to be stored in more than one potential location requires replacement policies to determine which cache line should be evicted (victim line) before a new line can be fetched. The four basic policies listed below vary in their effectiveness as well as their hardware and energy requirements. While Random replacement is widely applied for its simplicity, the harder to implement LRU policy often yields significantly higher hit rates. A basic LRU implementation consists of a stack holding cache lines in the order of their last access. It requires significant amounts of energy and time to re-order lines on every cache access and there are no known implementations for set-associative caches with more than two ways that are considered feasible. Pseudo-LRU approaches this problem by approximating LRU behavior based on binary trees [9]. LRR is an alternative to LRU that avoids overheads due to the re-ordering of cache lines. It arranges data inside a simple buffer structure with FIFO characteristic. However, it is limited to low hit-rates and moderate energy consumption. The second chance and the CLOCK algorithm alleviate these shortcomings by introducing an additional bit per line to indicate repeated accesses, and replacing the FIFO with a circular queue that avoids the energy intensive reordering of elements between accesses [10].

- LRU (Least Recently Used) : replace line holding least recently accessed record
- LRR (Least Recently Replaced) : replace line holding oldest record
  → also known as FIFO
- LFU (Least Frequently Used) : replace line that has been requested least often
- Random : replace a random line

If not explicitly stated otherwise, the L1D caches employed by subsequent chapters are 4-way set-associated and employ a random replacement policy. Reason for this is the concern that more specialized policies might skew the investigated memory accesses patterns by favoring or penalizing certain benchmarks disproportionately. Furthermore, the uTLB and TLB structures used in Chapter 4 and Chapter 5 rely on second chance and random replacement, respectively. The former was chosen to minimize control and energy overheads, the later to achieve high hit-rates at moderate energy consumption.

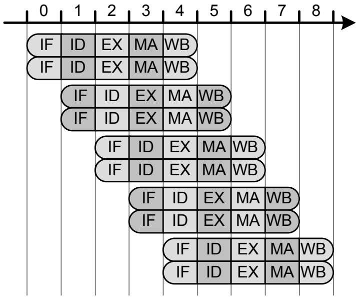
1.2.3 Summary of Key Parameters

This section described the principle concept of caches and introduced several related issues. The key parameters of a memory hierarchy may be summarized as:

- Memory Hierarchy:
  - ◦ Number of cache levels
  - ◦ Look up & write policies
    → BW between different levels
  - ◦ Coherency protocol (Section 1.4.2)
- Cache:
  - ◦ Capacity, line size
  - ◦ Associativity / number of ways
  - ◦ Replacement policy
  - ◦ Number of ports & banks (Section 2.1.2)
  - ◦ Use of physical or virtual addresses for indexes and tags

1.3 Out-of-Order Superscalar Processors

Introduced in the early 1960s in the context of supercomputers, pipelining describes the idea of breaking down instructions into a series of independent stages separated by storage elements. As it allows instructions to be issued at the speed of the slowest atomic step, the increase of the overall speedup directly dependents on the pipeline depth [11].



- IF: Instruction Fetch
- ID: Instruction Decode
- EX: Execute
- MA: Memory Access
- WB: Write-Back

Fig. 1.8: Example for the operation of a simplified superscalar pipelined processor

Building on the concept of pipelining, Very Long Instruction Word (VLIW) processors allow compilers to combine multiple independent instructions to be executed in parallel or specific sequence by a number of pipelines. Alternatively superscalar processors identify at run time which instructions might be executed simultaneously. The latter approach usually exploits a higher amount of parallelism for the cost of more complex hardware. Fig. 1.8 shows an example based on an implementation of two classic RISC pipelines [3]. It can be observed that two instructions finish on each clock cycle after an initial delay of four cycles. While some supercomputers feature pipeline depths of more than 100 stages, modern microprocessors usually implement 15-25 stages [12]. Main reason for this limitation is the problem of keeping every stage busy at all times. Conditional branches and data dependencies between successive operations provoke more and more stalls for increased pipeline depth and count. Common methods to reduce these limitations are branch prediction and out-of-order execution. However, besides their high implementation complexity, those techniques may require energy and time intensive pipeline flushes to recover from miss-predictions and exceptions.

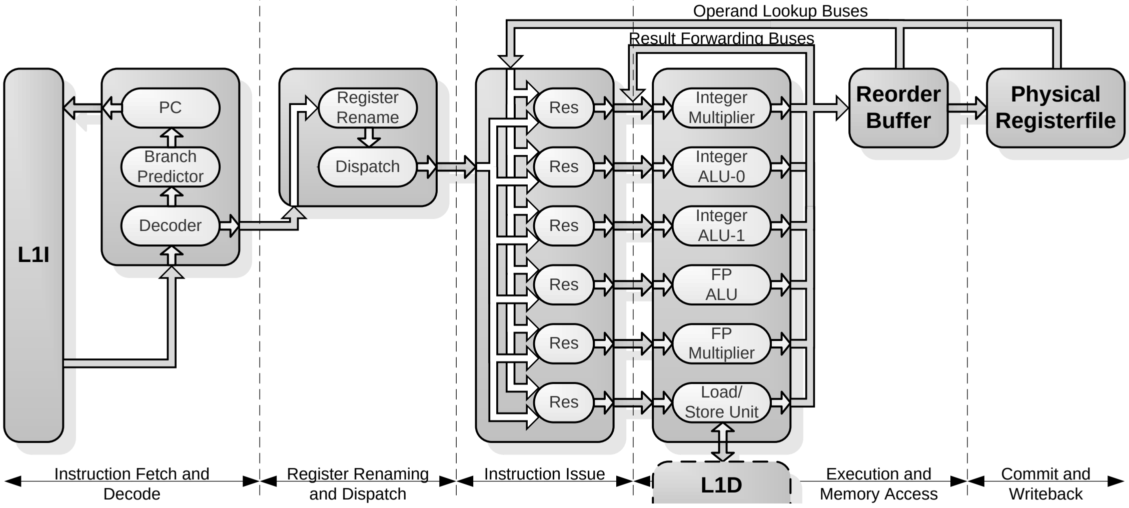


Fig. 1.9: Pipeline structure of a simplified out-of-order superscalar processor

The basic idea of superscalar processing is to exploit instruction level parallelism (ILP) by implementing hardware mechanisms to issue multiple instructions simultaneously while trying to avoid hazards and stalls. Fig. 1.9 depicts a simplified out-of-order superscalar datapath including a separation in five stages similar to Fig. 1.8 [13]. The following paragraphs describe the operation of a superscalar processor based on these five stages.

“Instruction Fetch and Decode” represent the first superscalar processing stage. On every cycle, multiple instructions are fetched from the L1I (Section 1.1.1) into a small buffer. There, instructions are held for times when fetching is stalled or restricted (e.g. an L1I miss). A subsequent bank of decoders breaks down complex instructions into basic operations (micro-ops) characterized by opcode and source and/or destination registers. Limiting factors for this stage are control dependencies in form of branch conditions and targets. Branch predictors mitigate this limitation by speculating over branch outcomes. Compiler based (static) predictors profile programs and insert flags or specific opcodes to indicate probabilities for certain outcomes. Alternatively, branch prediction buffers or branch history tables make predictions by utilizing information held on recently processed branches. While both hardware based structures are indexed by lower address bits, buffers are much simpler as they hold just one bit to indicate that a branch has recently been taken. In contrast, tables consist of counters that in-/decrease for taken/not taken branches. Similar structures might be implemented to track recent branch targets. In addition to their significant area and energy overheads, these mechanisms introduce problems associated with miss-prediction handling. In particular, should a later processor stage determine a branch condition or target that differs from the prediction, all instruction and results based on it need to be discarded and a state prior to it re-established.

The second stage in Fig. 1.9 is called “Register Renaming and Dispatch”. Register renaming describes a method to eliminate artificial data dependencies, namely Write after Write (WAW) and Write after Read (WAR) hazards. For example, an instruction might attempt to access a register that is currently locked by another instruction. Renaming avoids this by assigning logical identifiers instead of actual physical registers to operands. The two most common implementations are listed below. Note, the remainder of this report focuses on method (b), because ROB's can easily be extended to hold additional information such as an instruction's program counter (PC) and interrupt conditions. This simplifies the treatment of miss-predictions as it will be explained later on.

a) Extended physical register file
   - Physical register file exceeds number of logical registers
   - Mapping table associates a phys. reg. with current value of a logical reg.
   - Free list holds unassigned phys. regs. → stall dispatch when list empty
   - Reclaim of phys. regs. if no longer needed

b) Re-order Buffer (ROB)
   - Circular buffer, accessed in FIFO manner
   - Instr. dispatched in new entry at tail; results written to entry when ready
   - Instr. at top can commit (stall commit if not ready)
   - Mapping Table points to phys. reg. or result in ROB entry
   - Main problem: CAM[1] structure required to search ROB entries scales badly for wide instruction windows

Referring to the third superscalar processing stage, "Instruction Issue" describes a run-time check for the availability of data and resources; e.g. execution units, buses, ROB ports and registers. A common method for OoO processors is the assignment of so called reservation stations - also known as issue queues (IQs) - to specific instruction types or execution units (Fig. 1.9). When a new instruction is dispatch to a reservation station, its operands are copied from the ROB or the physical register file. Operands that are not available at this time are represented by their logical register handle. Every time a result enters the ROB, reservation stations check for matching handles to be replaced. Finally, an instruction issues as soon as all of its operands are known and required resources become available.

"Execution and Memory Access" designates the fourth superscalar processing stage. Fig. 1.9 includes a combination of functional units (FUs) as it might be found in a modern processor. Each FU executes a certain type of instruction using a sequence of discrete steps. Shortly before a result becomes available, the corresponding logical register handle is broadcast to stalled reservation station entries and the ROB. The forwarding of the actual result is arranged accordingly. Note, during this stage only read accesses to the L1D are performed. Writes are delayed until the next stage to ensure their correctness and therefore avoid the corruption of architectural state by speculative data (see Section 1.1.2 on Store Buffers).

The final superscalar processing stage is called "Commit and Writeback". Instructions at the ROB's head commit - also known as retire or complete - by writing their result into the physical register file or the L1D. Although the ROB might commit multiple instructions per cycle, the retirement takes place in program order.

[1] A content addressable memory (CAM) combines storage and comparison in one device. Instead of using an index (e.g. an address) to deliver a specific data, a CAM compares incoming data to its contents and returns an index to the matching entry [3].

As mentioned before, the handling of miss-predictions and exceptions is vital when realizing transparent OoO superscalar processors (Section 1.1.2). A popular method called Checkpointing uses a history buffer to store the state at the begin of each branch [13]. It is simple to implement, but shows a direct correlation between the number of allowed in-flight branches and buffer entries (i.e. its area and power consumption). An alternative based on the ROB may operate as follows:

- Identify possible hazards during dispatch; couple the resulting “hazard list” to the instruction on its way through reservation stations and the ROB
- Separate architectural and logical (in-order) state
    - Architectural state: physical register file updated on commit → always in-order
    - Logical state: ROB entries updated as soon as results become available
- Detect miss-predictions and exceptions during execution phase
- Identify undesired instructions by evaluating their hazard list
- Flush pipelines of already issued undesired inst., complete desired instr.
- Issue, execute and commit instr. that are younger than the hazard as normal

1.4 Parallel Processing

1.4.1 Overview

Until about 1986, parallelism in the context of microprocessors was primarily realized on the bit-level. Specifically, the increase from 8 to 16 and later to 32-bit word size led to a significant performance gain for full 32-bit operations. However, the recent advance to 64-bit is driven by demands for a larger address space and potentially reduces performance by introducing higher latencies due to an increased transistor count [14]. Following this period until the mid-1990s, instruction-level parallelism in the form of pipelining and OoO superscalar dominated the microprocessor market (Section 1.3).

Historically, Single Instruction Multiple Data (SIMD) datapaths and vector units have been developed in the context of data parallelism inside supercomputers. SIMD achieves parallel computations by replicating execution units. A vector unit might implement a SIMD datapath, but is also able to serialize vector instructions into several scalar (or narrow SIMD) operations to support wide vector lengths. The ability to compute instructions on multiple data points at once is widely used for scientific and engineering analyses as well as other data intensive tasks. Examples include:

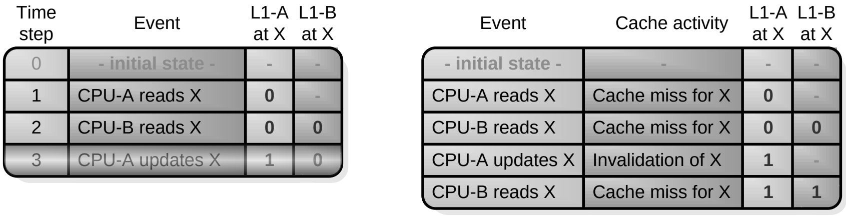
- Atmospheric and oceanic currents
- Applied, nuclear and particle physics
- Bioscience, biotechnology and genetics
- Geology and seismology
- Mathematics and computer science
- Mechanical structures and electrical circuits
- Data mining
- Oil exploration
- Web search engines
- Medical imaging and diagnosis
- Financial and economic modeling
- Nanotechnology

Beginning in the late 1990s, microprocessor vendors adapted these systems in the form of vector ISA extension for their high-end products. Nowadays, vector execution units are used for image processing in digital copiers, cameras, and camcorders as well as for broadband wireless systems [15, 16, 17, 18]. Other applications involve signal processing [19] and algorithms for efficient arithmetic, trigonometric, hyperbolic, exponential and logarithmic functions [20]. Particularly their energy efficiency qualifies vector execution units for use in mobile devices that aim to combine high performance with long battery lifetimes. Section 1.4.3 introduces ways to implement vector ISA extensions and Chapter 3 includes the analyses of their memory access patterns forming the basis for the energy efficient adaptations proposed in Chapters 4 and 5. Moreover, Chapter 6 includes a newly developed vector ISA extension that expands ARMv7 NEON's capabilities by introducing several advanced SIMD features previously exclusive to High Performance Computers (HPCs). Examples for well known vector ISA extensions in the context of commodity processors include:

- VIS (Sun Microsystems 1995)
- MMX (Intel 1996)
- 3DNow! (AMD 1998)
- AltiVec (Apple/IBM/Motorola 1998)
- SSE (Intel 1999)
- ARMv7 NEON (ARM 2004)
- AVX (Intel 2011)
- AVX2 (Intel 2013)
- ARMv8 NEON (ARM 2013)
- AVX-512 (Intel 2015)

The emergence of so called multi-core microprocessors beginning in the early 2000s completes this overview. Those systems implement two or more CPUs on a single chip to execute multiple independent instruction streams in parallel (Thread Level Parallelism). In 2004 Intel followed its competitors on this way to avoid heat issues caused by their former approach of continuously increasing the clock frequencies of their chips. By implementing several slow instead of one fast processor, they could reach the same performance at a lower heat quota [21]. Researchers predict that parallelism in combination with the intelligent use of specialized resources will direct market developments over this decade [1].

1.4.2 Multi-Processing and Memory Coherency



Time step	Event	L1-A at X	L1-B at X
0	- initial state -	-	-
1	CPU-A reads X	0	-
2	CPU-B reads X	0	0
3	CPU-A updates X	1	0

(a) Coherency Violation

Event	Cache activity	L1-A at X	L1-B at X
- initial state -	-	-	-
CPU-A reads X	Cache miss for X	0	-
CPU-B reads X	Cache miss for X	0	0
CPU-A updates X	Invalidation of X	1	-
CPU-B reads X	Cache miss for X	1	1

(b) Snooping Protocol

Fig. 1.10: Examples for coherency issues within a two-processor environment

The preceding section introduced the need for parallel processing in energy efficient high performance systems. However, sharing an address space between multiple processors can expose cache coherency problems. Fig. 1.10a illustrates this for two processors A and B. Initially neither L1 instance contains data for address X. Two steps later, both CPUs obtained local copies from a lower cache level. The actual problem appears in time step 3 when CPU-A performs a write access to X and renders the two L1 caches incoherent. Consequently, all subsequent reads of CPU-B from X operate on out dated (wrong) values. Hence, the sequential consistency model, which requires operations over all processors and for each individual processor to appear as if executed sequentially, was violated [3].

Depicted in Fig. 1.10b is the operation of a common snooping protocol called write invalidate. It overcomes coherency issues by enabling inter-processors communication via a bus structure [3]. In particular, the critical update of X by CPU-A is accompanied by the invalidation of this address within L1-B. Hence, the following read by CPU-B misses and receives the correct value from the lower cache level. This example assumes a write-back policy for L1-A to ensure that the data within the next lower level is up to date when CPU-B requests it (Section 1.1.1). The invalidation of X requires CPU-A to broadcast its intend to update X, and CPU-B to "snoop" the bus for messages concerning its cached data. Note that the shared bus structure common for snooping protocols is simple to implement but scales poorly. However, the introduction of more complex protocols like transactional memory for many-core systems is outside the scope of this document. Considerations regarding coherence protocols are relevant in the context of Page-Based Way Determination as described in Chapter 5. The proposed scheme was evaluated on a uni-core processor and needs to be adapted to efficiently service snooping requests associated with coherency related operations (Section 5.4.4).

1.4.3 Vector Processing

Overview

```
add A[ 0], B[ 0], C[ 0]
add A[ 1], B[ 1], C[ 1]
...
add A[31], B[31], C[31]
```

(a) Scalar Architecture

```
vAdd A[ 0.. 7], B[ 0.. 7], C[ 0.. 7]
vAdd A[ 8..15], B[ 8..15], C[ 8..15]
vAdd A[16..23], B[16..23], C[16..23]
vAdd A[24..31], B[24..31], C[24..31]
```

(b) Vector Architecture

Fig. 1.11: Example: addition of 32 data points

As stated in Section 1.4.2, vector ISA extensions in conjunction with SIMD datapaths build the foundation of this thesis. The following chapters include analyses of vector access patterns for energy efficient optimizations and several proposals based on those results. The key feature of vector processors is the support of instructions that operate on one-dimensional arrays of data (vectors) instead of individual data elements. Fig. 1.11 exemplifies this for the loop statement:

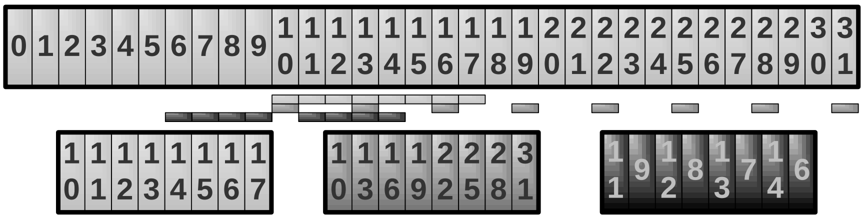
```
for(i=0; i<32; i++)
  C[i] = A[i] + B[i];
```

A scalar architecture requires at least 32 instructions (unrolled loop) to add pairs of 32 integers and even more when using branches for loop iterations (Fig. 1.11a). In contrast, the same results can be achieved by just four vector instructions and a vector length of $N = 8$ elements. This dramatically reduced instruction bandwidth can be explained by the homogeneity of vector operations. Only one set of logic circuits is required to decode and issue N operations to a group of identical functional units. As these units operate independently within so called lanes, it is not necessary to check for resource dependencies between operations contained within a single vector instruction. By further applying the superscalar techniques described in Section 1.3, long pipelines and consequently high clock frequencies can be realized.

Many modern vector ISA extension like NEON or SEE implement only narrow vector units (2-8 elements per vector) and basic instructions such as unit strides. The reasons for this are the energy and area overheads associated with more complex vector units and Amdahl's Law [22], which is often used to express the limitation of parallelism. For example, assuming that the loop above is enclosed by a function, its maximum effective vector length is 32. The function would not profit from wider vectors and requires at least two instructions (add and return to caller). However, the results in Section 3.7.2 show that even general purpose code can benefit from wide vector execution units, if these support more complex operations like non-unit strides and indexed accesses. Another point that supports the idea of wider vectors - e.g. 512-bit within Intel's upcoming AVX-512 - is the transition from 32 to 64-bit address spaces described in Section 1.4.

Section 6.5.2 investigates the dependency of performance results achieved based on the vectorization of general purpose code on datapath widths ranging from 128- to 512-bit. The proposals in Chapter 7 expand on this by introducing methods to process certain instructions on narrower datapaths without performance degradation, or employ multi-register operations to improve the utilization of existing datapaths. The preceding Chapters 4 to 6 introduce methods to mitigate memory bandwidth limitations often associated with vector processing circuitry and investigate the impact of advanced SIMD features that were previously exclusive to HPCs on the vectorizability of general purpose code and achievable performance gains.

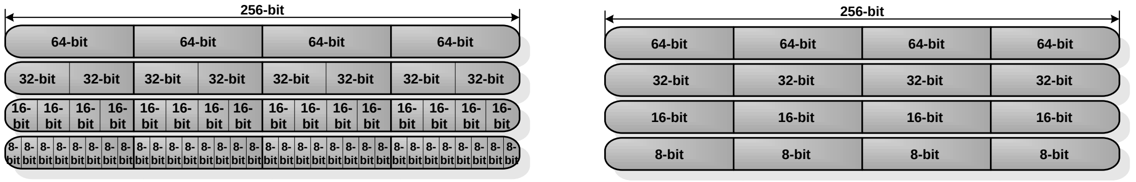
Vector Memory Accesses Patterns



(a) Unit stride (b) Non-unit stride (c) Indexed access

Fig. 1.12: Vector memory access patterns

Currently deployed vector ISA extensions are usually limited to the most basic memory access pattern, namely unit stride. The primary reason for this is the cost effective implementation of unit strides by re-using scalar components. For example, a load store unit capable of handling 64-bit elements might be employed to service two 32-bit, four 16-bit or eight 8-bit memory references. Hence, simple scalar memory accesses emulate vector loads/stores to a series of consecutive data elements. Fig. 1.12a illustrates this for an eight element access starting at address 10. Those instructions usually require only a base address and perform the remaining address computations implicitly; i.e. $base + i$ for element index i. The vector length for those operations is usually fixed, specified within the instruction encoding, or held within a dedicated register. Note that the example above assumes vector register elements to be stored in packed form. Fig. 1.13a illustrates this concept based on a 256-bit wide datapath and element sizes ranging from 8- to 64-bit. While this scheme is highly efficient in terms of resource utilization, its complexity may impose significant area, energy and latency overheads for wider datapaths. Unpacked registers are less efficient but simpler to implement. Based on a fixed number of lanes, a 256-bit datapath with 8 lanes may support eight 32-bit, eight 16-bit or eight 8-bit elements (Fig. 1.13b).



(a) Packed

(b) Unpacked

Fig. 1.13: Vector register packing schemes

A major problem of unit stride accesses is the time intensive rearranging of data to consecutive vector elements. Non-unit strides are less restrictive and allow strides unequal to one between vector elements. In the context of a high level programming language like C++, this allows the vectorization of complete arrays of structs. Hence, the example in Fig. 1.12b could be interpreted as a parallel access to the first entry of eight structs, each composed of three elements. Although this access pattern can be used to emulate unit strides, its higher complexity usually makes “pure“ unit strides more efficient. In particular, the multiplier required for the computation of element addresses and the additional hardware to determine TLB and cache line hits are less energy efficient.

Even more flexible than non-unit strides are indexed memory accesses. These instructions compute addresses for vector elements completely independent from each other. Fig. 1.12c shows an example corresponding to an otherwise unvectorizable loop. The addressing mode for this case involves the scalar base address 10 and a vector of offsets composed of 1|-1|2|-2|3|-3|4|-4. Other addressing modes such as base vector plus scalar offset or base vector plus vector of offset are also feasible but less applicable. Although indexed accesses can emulate both previously mentioned patterns, a complex address computation, increased instruction BW and high energy consumption render this approach less efficient. Section 3.7.2 shows the distribution of unit stride, non-unit stride and indexed access on a set of general purpose benchmarks.

A common issue for all three presented patterns are the high memory BW requirements associated with them. The analyses in Section 3.7.4 and Section 6.5.4 demonstrate how the limited number of loads/stores per cycle reduces the performance of microprocessors that simply serialize vector accesses into multiple scalar operations. Possible cache optimizations for unit strides include wide ports and address interleaved banks (Section 2.1.2) to access multiple elements per line and operate on adjacent lines in parallel, respectively. However, these adaptations are of limited use for non-unit strides and indexed access. Consequently, Chapter 3 analyses vector access patterns for possible cache optimizations, followed by descriptions of energy efficient implementations in Chapter 4 and Chapter 5. Moreover, Chapter 6 describes the design of a Vector Benchmark Suite that is among other things used to evaluate the impact of indexed memory accesses in the context of vectorized general purpose applications.

1.5 Thesis Organization

While the majority of modern processor designs implement multiple cores and throughput accelerators to exploit parallelism for performance purposes, simpler and often more energy efficient vector execution units are often underutilized. Considering the high memory bandwidth demands and potential energy costs associated with vector processing, this dissertation focuses on two main areas of investigation: the efficient handling of parallel memory accesses and the optimization of vector functional units. It aims for energy savings due to improved efficiency as well as reduced computation times that increase the proportion of time spend in low power states. The subsequent chapters are organized as follows:

Ch. 2 Literature Survey

This chapter surveys research in the areas of high level memory designs and parallel processing. Moreover, it outlines the objectives underlying this thesis.

Ch. 3 Analysis of Memory Access Patterns to Enable Energy Efficient Parallel Accesses

This chapter investigates memory access patterns exhibited by general purpose code with the aim of enabling the design of energy efficient high level memory interfaces. It includes statistics on the ratio of load, store and computation instructions, the characteristic of consecutive cache accesses, and the influence of vectorization on said memory access patterns.

Ch. 4 Page-Based Memory Access Grouping

Based on conclusions derived from the preceding analyses, this chapter introduces the idea of Page-Based Memory Access Grouping. The proposed Multiple Access Low Energy Cache (MALEC) exploits the observation that consecutive memory references tend to access the same page of memory. It efficiently utilizes single-ported structures to achieve performance similar to state-of-the art multi-ported designs, while exhibiting significantly lower energy consumption.

Ch. 5 Page-Based Way Determination

This chapter extends MALEC with the concept of a way determination scheme specifically designed to service multiple parallel memory accesses in an energy efficient manner. Page-Based Way Determination holds way information of recently accessed cache lines in small memories that are closely coupled to TLB lookups. It builds upon MALEC's restriction to accesses to only one page per cycle, in order to efficiently provide way information corresponding to all memory accesses in any given cycle.

Ch. 6 Vector Benchmarking

The evaluation of MALEC based on scalar implementations of general purpose workloads revealed that even high performance processor configurations do not exert sufficient memory pressure to fully utilize its capabilities. This chapter introduces ARGON, i.e. a derivative of ARMv7 NEON extended to support several advanced SIMD features that were previously exclusive to high performance computers and only recently considered for commercial microprocessors. It furthermore describes the development of a Vector Benchmark Suite (VBench), which provides scalar and vectorized implementations of several popular algorithms. The combination of ARGON and VBench allows the evaluation of high performance memory interfaces as well as the impact of particular features such as increased datapath widths, per-lane predication and indexed memory accesses.

Ch. 7 Microarchitecture Optimizations for Energy Efficient SIMD Datapaths

This chapter addresses several concerns that arose during the implementation and evaluation of advanced SIMD features within the ARGON simulator framework. It introduces micro-architectural optimizations aiming to increase the performance and energy efficiency of vectorized code in general as well as of specific execution units. These optimizations include techniques to

- compute hardware interpretable masks for segmented operations,
- partition scans to allow specific energy - performance trade-offs,
- re-use existing multiplexers to process predicated and segmented vectors,
- accelerate scans on incomplete vectors,
- efficiently handle micro-ops fully comprised of predicated elements, and
- reference multiple physical registers within individual operands to improve the utilization of the vector register file.

Ch. 8 Conclusions and Future Work

This chapter summarizes the contributions introduced in the preceding chapters and evaluates them in context of the objectives underlying this thesis. Furthermore, a number of areas for future research that would improve upon or extend the proposed techniques are briefly discussed.

1.6 Contributions

The objectives of this thesis as derived from the literature review are discussed in Section 2.3. The contributions related to those objectives are described in Chapters 3 to 7, summarized in Section 8.1, and have been published as follows:

Academic Publications

[1] M. Boettcher, G. Gabrielli, B. M. Al-Hashimi, and D. Kershaw, "MALEC: A Multiple Access Low Energy Cache" in *DATE*, 2013.

[2] M. Boettcher, G. Gabrielli, M. Eyole, B. M. Al-Hashimi, and A. Reid, "Advanced SIMD: Extending the Reach of Contemporary SIMD Architectures" in *DATE*, 2014.

[3] M. Boettcher, G. Gabrielli, M. Eyole, B. M. Al-Hashimi, and A. Reid, "Evaluation of advanced SIMD Features in the Context of general purpose Algorithms" submitted for publication.

Patent Applications

[3] M. Boettcher and D. Kershaw, "United States Patent Application: Data processing Apparatus having Cache and Translation Lookaside Buffer", United States Application No. 13/468,548, 2012.

[4] M. Boettcher, M. Eyole, and G. Gabrielli, "United States Patent Application: A Data Processing Apparatus and Method for Performing Scan Operations", United States Application No. 14/165,967, 2014.

[5] M. Eyole, M. Boettcher, and G. Gabrielli, "United States Patent Application: A Data Processing Apparatus and Method for performing segmented Operations", United States Application No. 14/175,268, 2014.

[6] M. Boettcher, M. Eyole, and G. Gabrielli, "United Kingdom Patent Application: A Data Processing Apparatus and Method for performing vector scan Operations", United Kingdom Patent Application No. 1403955.6, 2014.

[7] M. Boettcher, M. Eyole, and G. Gabrielli, "United Kingdom Patent Application: A Data Processing Apparatus and Method for processing Vector Operands", United Kingdom Patent Application No. 1404037.2, 2014.

2 | Literature Survey

The area of efficient vector processing techniques has been subject to extensive research since the onset of vector supercomputers in the 1970's [23]. It has gained more recent interest due to the introduction of advanced SIMD features into high-performance oriented vector ISA extensions, and the desire to meet increasing performance demands within energy constrained environments [24]. Of particular concern are the significant memory bandwidth requirements implied by the utilization of SIMD datapaths. In conjunction with the discrepancy observed in the performance development of processing and storage elements - often referred to as the "Memory Wall" [2] - adequate cache structures are imperative.

This chapter surveys state-of-the-art techniques in the domain of efficient high level data cache designs and vector processing circuitry. It expands on the basic information presented in Section 1.2 by discussing the primary challenges associated with L1D interfaces and recent proposals on approaches to overcome them (Section 2.1). This includes attempts to reduce dynamic energy consumption by limiting the number of transistors activated during cache accesses, and to improve hit-rates while reducing miss penalties. Furthermore, Section 2.2 elaborates on the concept of vector processing introduced in Section 1.4. It focuses on the migration of advanced SIMD features from supercomputers to vector ISA extensions, and the efficient implementation of said extensions in the context of energy constrained general purpose microprocessors. The aims and objectives of this thesis are presented in Section 2.3, followed by concluding remarks in Section 2.4.

2.1 Cache Optimizations Techniques

The subsequent sections give a brief overview of research in the area of efficient high level data cache designs. They include techniques to reduce the energy consumed during individual accesses and improve hit-rates while reducing penalties associated with misses (Section 2.1.1). Furthermore, common multi-porting schemes are introduced and evaluated in Section 2.1.2, followed by a summary of way prediction schemes commonly employed to further reduce the dynamic energy consumption of data cache accesses (Section 2.1.3).

2.1.1 Techniques to Reduce Dynamic Energy Consumption and Increase Hit-Rates

A common approach to improve the energy efficiency of caches is to reduce the number of transistors activated during accesses. For example, Kamble et al. [25] and Su et al. [26] analyze so called line-buffer that store the last accessed cache line to avoid activating the whole L1 for consecutive accesses to the same line. The authors find those buffers especially suitable for L1Is, due to the high spatial locality of instruction streams. They also evaluate sub-banked data arrays, which partition cache lines over multiple independent banks to allow accesses to specific elements without activating complete lines. As the efficiency of those banks depends on line size and not data locality, they are applicable for both instruction and data caches. Although these techniques are standard in modern microarchitectures, they are mutual exclusive. In particular, the narrow access ports (usually 128-bit wide) associated with sub-banking do not support the transfer of whole cache lines into a line-buffer. Note that the terms sub-banking and sub-blocking are often used interchangeable in the literature. This document - in particular Chapter 4 and Chapter 6 that use this technique in the context merging schemes - adhere to the following terminology:

- **Sub-Banking:** concept of mapping cache lines over multiple independent data arrays
- **Sub-Bank:** one of said arrays
- **Sub-Block:** part of cache line held by one of said arrays

Rivers et al. [27] combine line-buffers and multi-ported cache designs as described in Section 2.1.2 within their locality-based interleaved cache. They propose the implementation of a number of simple cache banks with few ports and a single physically multi-ported line-buffer per bank. Kin et al. [28] and Nicolaescu et al. [29] adapt the principle of line-buffers for small memories between L1 and processor. Both proposals attempt to reduce L1 activity by holding very small subsets of recently accessed lines. The problem with these techniques are penalties on performance and energy consumption for programs with low spatial locality. An alternative approach to reduce the internal transistor activity of a cache is described in Section 2.1.3. The reason for the separate, in depth discussion of way prediction is its relevance for the proposal in Chapter 5.

The concepts of line-buffers and interleaved caches have also been studied in the context of TLB structures [30]. Austin and Sohi propose the utilization of banking and multi-level designs to realize multi-porting and reduce TLB energy consumption [31]. They furthermore describe the idea of piggyback ports that exploit spatial locality by comparing virtual page IDs of incoming requests against all in-flight translations. Chapter 4 expands on this idea by introducing an “Input Buffer” capable of holding access requests between clock cycles, handling evicted store buffer results, and simplifying bank allocations as well as the merging of read accesses. The authors furthermore propose pretranslation as a method to reuse translation results without further TLB accesses. For this purpose, translation results are attached to registers values the first time they are used as base address during address generation. However, the increase in registers size may increase access latency and energy consumption. In conjunction with the need for circuitry to handle mispredictions, propagate attached values following pointer updates and invalidate them on TLB updates, pretranslation is considered unsuitable in the context of vector processing.

Orthogonal to the methods above, several authors have proposed to optimize caches by improving hit-rates. In this context, Sections 1.2.2 and 2.1.2 discuss advanced replacement policies and alternative memory mapping schemes. Chapter 3 investigates several cache parameters for their influence on hit-rates. Non-blocking caches do not impact hit-rates, but attempt to hide latencies associated with misses by avoiding stalls [3]. In particular, “hit under miss” implementations can service multiple hits during an outstanding miss. Similarly, “miss under miss” caches support multiple outstanding misses and can therefore overlap corresponding latencies. Disadvantage of these structures are high BW requirements between adjacent cache levels to handle multiple parallel line refills. Modern caches combine both mechanisms to realize high access rates. The simulation frameworks underlying the following chapters employ fill buffers and miss status holding registers (MSHRs) for this purpose. In addition, they employ a simple scalar prefetcher for their second level cache. The idea behind prefetching is to request data before it is actually needed. Hence, missing cache lines are obtained without any delay perceivable to the processor. Primary concerns of this technique are the identification of relevant data and the potential for cache pollution; i.e. overwriting still useful data with prefetched lines. Compiler and hardware based prefetchers haven been extensively studied by academia and are common place in modern microprocessors. Although their benefits have been proven in the context of certain vector processors [32, 33], they are considered beyond the scope of this work. Other optimization techniques that are considered out of scope, too, include areas such as trace-caches, reconfigurable caches, non-uniform caches, real time compression schemes, cache bank clock/power gating and cache decay.

2.1.2 Multi-Porting

Multi-porting describes methods used to service several memory accesses per CPU cycle. It is a key feature for the parallel computation of instructions and/or data points. For example, a simple add operation usually requires two read and one write access to store the sum of two operands into a destination. Juan et al. distinguish and compare four primary approaches [34]:

- **True multi-porting:** Increased physical number of memory ports by increasing transistor count per memory cell
  - + Conflict free → very high performance
  - – Very high energy and area consumption
  - – Increased latency per access
- **Time division multiplexing (cache overclocking):** Physically single-ported; cache clock set to multiple of processor clock
  - + Conflict free → very high performance
  - + No extra area cost
  - – Hard to realize (meet timing constrains), low power efficiency
- **Mirroring (cloning):** Multiple identical devices holding the same data
  - + Easy to implement
  - – Multiple read but shared write ports (write to all instances to ensure coherence) → moderate performance
  - – High area and energy consumption

- **Interleaving/Banking:** Cache partitioned into several independent segments (banks) that can be queried in parallel
  - + Minor area and energy overhead
  - + Easy to implement
  - – Performance limited by serialization of multiple requests to the same bank

Due to its high area and energy costs, true multi-porting is primarily employed for processor internal, high speed register files. The significantly larger L1 and L2 structures are often implemented using 2-4 and 8-16 independent banks, respectively [8, 6, 5]. A primary problem of such banking schemes are bank conflicts. These prevent the parallelization of accesses in consequence of multiple consecutive references to the same bank. To reduce the probability of such conflicts, several data organization techniques have been proposed. For example, Tong et al. [35] and Rau [36] introduce prime-mapped and pseudo-randomly interleaved caches, respectively. While both techniques reduce conflicts caused by access sequences separated by a constant stride, they also introduce complexity due to additional address computations. A more far-reaching proposal by Hallnor et al. suggests pointers inside each tag to locate specific lines. As each line can be placed at any location, line conflicts become unlikely [37]. However, this approach resembles a fully associative structure and is unsuitable for efficient banking.

Cho et al. apply the concept of prime-mapped caches to a 16-element wide SIMD datapath [38]. Assuming a heavily banked (16-banks) cache, they show that potential performance benefits due to an additional bank are limited to algorithms exhibiting very specific strides. This is particularly concerning when considering that the energy and latency overhead introduced by the more complex address computation circuitry affects all memory accesses equally. An alternative approach to handle line conflicts is to specifically target non-unit strides and separate accesses corresponding to individual elements into multiple conflict-free sets. In particular, McKee et al. introduce compiler inserted hints to identify non-unit strides within scalar code segments [39], whereas Espasa et al. and Seznec et al. consider a SIMD datapath directly interacting with a second level cache [40, 41]. The proposal described in Chapter 4 is similar to this approach in that it uses banking and access grouping to achieve performance benefits. However, as it operations on L1 instead of L2 caches, it is unable to hide latencies associated with the additional lookup structures of [40] behind comparatively slow L2 accesses. It furthermore allows scalar, uni-stride, non-unit stride and indexed memory access to be grouped together.

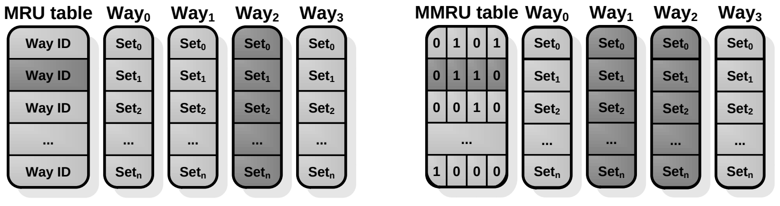
2.1.3 Cache Way Prediction Schemes

The following paragraphs summarize existing way prediction schemes to outline the context of Page-Based Way Determination proposed in Chapter 5. In contrast to the schemes listed here, which were primarily designed for single-access caches, the later proposal attempts to enable multiple L1 accesses per cycle[1]. Way prediction schemes may be categories based on their accuracy into three distinct groups. The first group includes actual prediction schemes that rely on run-time statistics to predict the way a certain datum is most likely located in. The second group avoids penalties associated with mispredictions by determining rather than predicting ways based on information stored on a subset of recently accessed cache lines. Finally, the third group provides estimates in form of sets referencing one or more ways, each set is thereby guaranteed to include the way in which the desired datum is held. An important factor for all three groups is the type of address data utilized [42]:

- Virtual Address:
  - ○ Accuracy : very high
  - ○ Availability : after address computation → might be time critical
- Register Contents and Offset:
  - ○ Accuracy : high
  - ○ Availability : before address computation
  - ○ Example : XOR register contents and offset to index prediction mechanism
- Register Number and Offset:
  - ○ Accuracy : moderate
  - ○ Availability : several cycles before address computation
  - ○ Problems :
    - Changing register contents
    - Register number + small offsets may cluster around certain indexes
    - Some registers are more frequently used than others (e.g. stack pointer)
- PC and previous references:
  - ○ Accuracy : low
  - ○ Availability : very early

In contrast to the sources identified by Calder et al. (listed above), the prediction scheme introduced in Chapter 5 is based on physical addresses [42]. This choice implies very high hit rates by an even further delayed availability. In fact, the scheme is most suitable for energy constrained systems that employ PIPT rather than VIPT caches (Section 1.2).

Way Prediction



(a) MRU Way Table

(b) MMRU Way Table

Fig. 2.1: Examples for prediction schemes on a 4-way set-associative cache

Fig. 2.1a depicts an early way prediction scheme proposed by Inoue et al. [43]. A MRU table includes bit flags to indicate the most recently used way corresponding to each cache

[1] Several other proposals increase the energy efficiency of way prediction schemes by judging their efficiency at run-time and en-/disabling them accordingly. As these proposals operate orthogonal to the proposal in Chapter 5, they are not further discussed in this report.

set. Assuming a 16 KByte, 4-way cache and a line size of 32 bytes, the MRU table includes 128 2-bit entries within a size of just 32 byte. One issue of this implementation is the delay introduced by the MRU-table look-up prior to the cache access. It might be mitigated by using the previously introduced method of performing an exclusive OR operation on address register contents and corresponding offsets. The prediction accuracy of Inoue's and similar proposals is a critical factor in their realization. Specifically, false predictions require a second memory cycle to access all the remaining ways; hence, an additional delay is introduced and no energy saved.

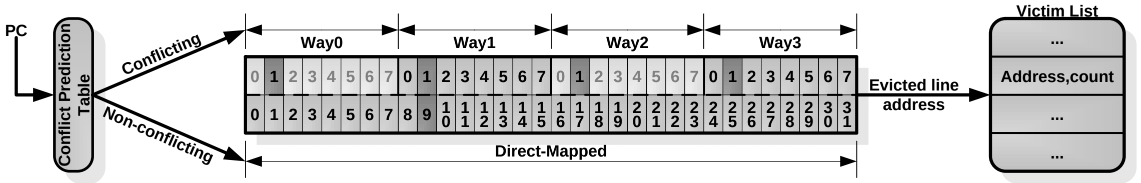


Fig. 2.2: Cache structure for selective direct-mapping

A method to increase the accuracy of the above scheme was proposed by Powell et al. [44] (Min et al. proposed the same design for L2 caches [45]). The authors suggested a combination of selective direct-mapping and way-prediction. They claim that approximately 70% of L1 read accesses can avoid way-prediction by utilizing additional address bits to assign cache lines in a direct-mapped manner (stores do not undergo way prediction, because writes based on false predictions would invalidate architectural state). Only those cache lines corresponding to the remaining accesses would need to be assigned set-associatively in order to avoid frequent evictions by conflicting lines. Fig. 2.2 illustrates how a cache line might map to exactly one of four different locations based on direct or 4-way set-associative mapping, respectively. The actual way-prediction method is orthogonal to this scheme and might be adapted from any proposal introduced in this section.

An alternative approach to achieving higher prediction accuracy is to increase the number of ways activated per access. Keramidas et al. describe Multi-MRU (MMRU) that returns up to n recently used ways for an n-way set-associative cache [46]. Their MMRU entries are composed of n fields that indicate which of the n ways has been accessed recently. Fields might be implemented with mono-stable circuits that are set on hits to their corresponding way-set pair and discharge over time. Alternatively, counters could be incremented on hits to specific way-set pairs. As long as the difference between their values and an additional counter (for hits to the set in general) do not exceed a given threshold, corresponding ways are considered recently accessed.

Way Determination

A major problem of the schemes introduced so far is the performance loss following false predictions. Way determination schemes avoid multiple memory accesses by guaranteeing 100% accuracy for their predictions. An early way determination scheme specifically designed for L1Is was proposed by Ma et al. [47]. The authors claim that the majority of L1I accesses are performed in a sequential manner, i.e. consecutive instructions are fetched from adjacent addresses. In case of intra-line accesses, tag look-ups can be re-used by re-accessing the previous cache line. To handle inter-line accesses, a way field within each cache line indicates the way the “next” line is located in. Although this scheme delivers good results for L1Is, it is rather hard to adapt to superscalar processor that fetch multiple instructions per cycle that may correspond to different branch outcomes. Note that the connection of L1I lines with way fields might be interpreted as a simplified form of a trace cache [48].

Nicolaescu et al. propose a small memory structure called way determination unit (WDU) that holds address–way pairs of recently accessed cache lines [49]. The structure is looked up prior to the cache and supplies a single way. In case of a WDU hit, the corresponding memory reference is guaranteed to hit in the predicted way. Otherwise all ways are looked up in parallel and a new WDU entry is allocated (FIFO policy). Consequently, cache accesses require only a single memory cycle independent from the prediction result. Furthermore, it is not necessary to invalidate WDU entries on cache line evictions; as the desired line cannot be present in any other way, accesses to invalid WDU entries simply result in cache misses. WDUs form the foundation of Page-Based Way Determination introduced in Chapter 5.

Way Estimation

A major disadvantage of way determination schemes is their limited number of predictable addresses. In particular, a WDU may be designed to hold 16 address–way pairs. For instruction streams with poor temporal locality, insufficient WDU space might result in low WDU hit rates; hence, the majority of instructions would need to access all cache ways. Way estimation schemes avoid this problem and still guarantee correct predictions. They predict a set of ways in which a specific cache line is guaranteed to be found (assuming it has been caches at all).

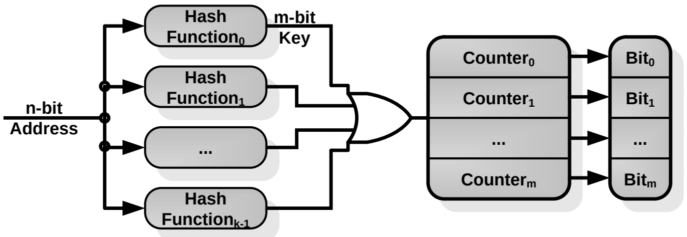


Fig. 2.3: Counting bloom filter

Ghosh et al. propose a design based on bloom filters [50]. In principle, it uses the n-bit address of each new cache line to compute k independent m-bit hash keys, combines them, and adds them bit-wise to a set of counters. By interpreting counter values not equal to zero as logic 1, an m-bit key is composed (Fig. 2.3). Prior to each cache access, the hash functions of the desired address are calculated and compared against this key. The presence of a specific address can be ruled out, if the bit-wise comparison reveals a logic 1 inside one or more hashes instead of a logic 0 for the same position inside the key, By assigning one key to each cache way, and decrementing the counter values for evicted lines, this scheme estimates in which way(s) a certain address might be found. Note that if the prediction scheme does not return any way, it is guaranteed that the corresponding address is not cached. Hence, the equivalent instruction might directly be forwarded to a lower cache level. However, even a positive result for a specific way does not guarantee that the data can actually be found there. Two disadvantages of this and similar schemes are the need for tag-array accesses to confirm prediction results and the energy consumed for unnecessary data-array accesses [51, 52].

2.2 Efficient Vector Processing Techniques

The following sections summarize academic and commercial efforts in the area of vector processing. Section 2.2.1 introduces the origins of vector processing in the context of supercomputers, and its migrations via highly application specific vector microprocessors into the domain of vector ISA extensions for general purpose microprocessors. Section 2.2.2 further elaborates on the implementation of these vector ISA extensions in general and the advanced SIMD features analyzed in later chapter in particular.

2.2.1 Vector Microprocessors and ISA Extensions

The concept of vector processing originates from the area of supercomputing. One of the earliest commercially available vector computers was the Cray-1 [23]. It introduced the concept of vector register files in form of a set of eight registers, each able to hold sixty-four 64-bit words. Due to the excessive cost of transistors at the time, the Cray-1 employed chaining to serialize vector registers onto scalar execution units. Later generations such as the Cray-2, the Cray X-MP and the Cray Y-MP migrated to SIMD datapaths in order to compute multiple elements in parallel [53, 54, 55]. Beginning in the early to mid 90s, supercomputers shifted to massively parallel solutions rather than wider SIMD units. However, the ever increasing cooling and energy costs reawakened the interest in vector processing in recent years. Nowadays, supercomputers employ a combination of SIMD and massively parallel processing [56, 57], and use throughput accelerators such as specialized co-processors and general purpose GPUs to achieve high performance and energy efficiency [58, 59, 60]. For instance, 53 of the top 500 supercomputers - including ranks 1, 2, 6 and 7 - currently employ NVIDIA GPUs and/or Xeon Phi co-processors [61]. Further research is also conducted in the area of high-performance reconfigurable computing, which suggest the use of FPGAs or similar reconfigurable devices to provide “custom fit” hardware solutions on a per workload or even per program phase basis [62].

Vector microprocessors represent an intermediate step between vector supercomputers and vector ISA extensions for general purpose microprocessors. The T0 vector microprocessor is a series of systems based on an extended MIPS ISA developed within the International Computer Science Institute (ICSI), and later in collaboration with U. C. Berkeley [63]. The processor employs sixteen 32-element vector registers, each element able to hold 8-, 16- or 32-bit values in unpacked form. Its SIMD datapath is comprised of eight 32-bit wide lanes and supports unit-stride, non-unit stride, and indexed memory accesses. As it does not possess a dedicated mask register file, it reuses regular vector registers to hold masks. Besides the T0, other vector microprocessors have been developed in the context of highly parallel media and scientific workloads and implemented either in form of ASICs or FPGA designs [64, 40, 65, 66, 67].

Beginning in the late 1990s microprocessor vendors introduced vector ISA extension for their high-end products (Section 1.4). The most recent and popular extensions are ARM NEON and Intel AVX. ARMv7 NEON is particularly relevant for this thesis as the ARGON ISA extension developed in Chapter 6 was derived from it. The architecture extension was design for the execution of media and DSP workloads within energy constrained devices. It reuses its FP register file to hold up to thirty-two 64-bit doubleword or sixteen 128-bit quadword registers [68]. Note that the recently released ARMv8 NEON does provide a dedicated vector register file and employs a different mapping scheme (Section 7.7.2). The supported instruction types include 8-, 16-, 32-, and 64-bit signed and unsigned integers, 32-bit single-precision FP, and 8- and 16-bit polynomials. Whereas low- to mid-range implementations, such as the Cortex-A8 and A9, execute vector instructions in two parts on a 64-bit datapath, high-end cores, such as the Cortex-A15, employ a 128-bit wide SIMD datapath. One reason for the development of ARGON was NEON's lacking support for indexed memory accesses; i.e. it is limited to unit strides and non-unit strides over 2, 3 or 4 elements [69]. As of now, it is uncertain to what degree ARM's recent entry into the 64-bit server market[2] will drive the adaptation of features investigated by ARGON. One indicator may be the recent development of AVX, with its focused on high performance desktop, server and HPC solutions. The newest iteration called AVX-512 or AVX3 defines thirty-two 512-bit vector registers (ZMM). The lower 256 and 128 bits of a ZMM are aliased to corresponding 256-bit YMM and 128-bit XMM registers, respectively [24]. Furthermore, it introduces a mask register file including 8 masks with merging and zeroing capability, and support for indexed memory access. AVX-512 will be used as part of Intel's Knights Landing and Many Integrated Core architecture (MIC); the latter also being known as Xeon Phi incorporates work of the earlier Larrabee architecture [58, 70].

2.2.2 Increasing Vectorizability and Datapath Utilization

The key considerations for vector ISA extension may be summarized to vectorizability and resource utilization. While the former determines the degree to which applications may be vectorized and therefore the limits of potential performance gains, the latter represents a metric for the efficiency of this process. In the context of general purpose code - as it is targeted by this thesis - the primary factors restricting vectorizability are irregular computation patterns, data dependencies and the need for balanced computation paths. Govindaraju et al. attempt to overcome these limitations by employing a flexible SIMD datapath

[2] Recently announced 64-bit ARM server platforms include the AMD Opteron A1100 series, the Applied Micro X-Gene and the HP Moonshot System. The European research project called Mont-Blanc takes this a step further and investigates ARM-based HPCs.

comprised of a heterogeneous grid network of functional units and switches, per-lane predication and a limited support for indexed stores [71]. One downside of this approach is the significantly increased datapath complexity, which imposes considerable area and energy costs in addition to increased latencies; hence, a practical implementation of the proposal would likely have to operate on a reduced clock speed. Furthermore, the authors expose microarchitectural features at the compiler level to permit some form of auto-vectorization.

The problem of endowing a compiler with sufficient intelligence in order to successfully map arbitrary programs to SIMD has been well-studied [72, 73, 74]. The capabilities of those techniques are usually limited by loops including ambiguous data dependencies, non-inlined calls to subroutines and functions, I/O statements, unvectorizable intrinsics, exception handling and runtime dynamic type casts or other type manipulations [75]. Consequently, architecture specific libraries may be utilized to achieve high degrees of vectorization for performance critical code segments without relying on auto-vectorization [76, 77]. Furthermore, tools such as Pareon can assist manual vectorization and data partitioning efforts by identifying hot spots, outlining data dependencies and communication patterns, and providing compile-time performance estimates [78].

The majority of SIMD datapaths implemented in commercially available general purpose microprocessors are based on simple, narrow functional units controlled by a small set of flexible primitives (Section 2.2.1). Only recently, has the continued scaling of transistor technologies lead to the adaptation of advanced SIMD features, previously exclusive to vector supercomputers, into the domain of high-end vector ISA extensions [24]. Gebis and Patterson studied the steps required to transform basic SIMD into more versatile vector instructions for the 80x86 and PowerPC ISAs [79]. In particular, the authors emphasized the need to support per-lane predication, indexed memory accesses and wider vector registers. Chapter 6 extends this list to the feature set given below and focuses on general purpose rather than media and signal processing applications. It furthermore investigates the performance impact of individual features and analyzes them in terms of their potential to improve vectorizability and datapath utilization. Based on the simulation framework used for these analyzes, Chapter 7 proposes a set of microarchitecture optimization to improve their performance and/or energy efficiency. Note that for the purpose of a more coherent presentation, further details on particular features are provided in form of background sections corresponding to specific proposals.

- Per-lane predication (Sections 6.2.1 to 7.6.2)
- Indexed memory accesses (Sections 6.2.2 and 1.4.3)
- Scans (Sections 7.3.2 to 6.2.3)
- Segmented scans (Sections 7.2.2 and 6.2.4)
- Flexible datapath widths and double-length register operations (Sections 7.7.2 and 6.2.5)

2.3 Objectives

The recent interest in the SIMD processing paradigm, fostered by its adaption into performance oriented vector ISA extensions [24], marks an important step in its transition from supercomputers [23], over vector microprocessors [63] to widespread utilization in general purpose computing. A key concern persisting throughout this process are high memory bandwidth requirements. In the context of modern, energy constrained memory systems, the design of efficient caches that support an increasing number of parallel accesses becomes imperative. Research into line-buffers, cache banking, sub-banking and similar techniques show ways of reducing dynamic energy consumption by limiting the number of transistors activated during accesses [25, 26, 27, 28, 29]. Similarly, the domain of way prediction attempts to avoid energy consumed by unnecessary data-array activations [43, 52, 49]. However, the requirement to service multiple accesses in parallel introduces new challenges in terms of scalability and general applicability to traditionally single-access oriented techniques. In particular, the problem of line conflicts and the cost of physically multi-ported structures motivate research such as McKee et al.'s hardware assisted access ordering scheme [39]. This and similar schemes efficiently process non-unit strides in form of sets of non-conflicting accesses [40, 41]. Another challenge is the handling of general purpose code, contrary to well behaved media and DSP kernels, which implies the potential presence of irregular computation and memory access patterns. Primary focus of this thesis is the development of techniques for the efficient handling of parallel memory accesses and the optimization of vector functional units in the context of general purpose workloads. Its key objectives are:

- Identify to what extend parallelization affects the memory access behavior of general purpose algorithms, and derive implications for the design of appropriately adapted memory systems.
- Utilize these insights for the design of a scalable high level cache interface capable of providing the desired degree of parallelism within an energy constrained system.
- Establish clear statements regarding the performance impact of advanced SIMD features in the context of vector ISA extensions utilized for the execution of general purpose algorithms.
- Develop techniques to improve the applicability of those advance SIMD features within energy constrained systems, by reducing the complexity and improving the utilization of functional units associated with them.

2.4 Concluding Remarks

This chapter presented a cross section of state-of-the-art research and technology in the areas of efficient cache design and vector processing. The ongoing rise in demand for ever faster and more efficient processing circuitry continues to drive wide spread research efforts and shapes commercial interests. Main challenges regarding high level data cache designs are the reduction of the number of transistors activated during accesses in order to conserve energy, and the improvement of hit-rates in conjunction with the reduction of penalties associated with misses. One particular issue is the rising pressure on the L1D interface to sustain the demands of high performance scalar computation units in combination with vector processing circuitry. In recent years, the capabilities of vector ISA extensions have been continuously improved by introducing advanced SIMD features previously exclusive to HPCs. The adaptation of those features in the context of general purpose microprocessors continues to proof challenging. All the above drive the motivation for the research presented in the following chapters. In particular, the analyzes of memory access patterns to enable energy efficient L1D interfaces (Chapter 3), the concepts of Page-Based Memory Access Grouping and Page-Based Way Determination originating from them (Chapter 4, Chapter 5), the evaluation of advanced SIMD features utilizing a custom vector ISA extension and benchmark suite (Chapter 6), and the set of microarchitecture optimizations addressing shortcoming identified during the implementation of said ISA extension (Chapter 7).

3 | Analysis of Memory Access Patterns to Enable Energy Efficient Parallel Accesses

3.1 Motivation

In order to develop energy efficient optimizations for memory systems and estimate their impact, it is necessary to understand how frequently certain access patterns occur. For example, if the majority of vectorized accesses were unit-strides, line buffers as mentioned in Section 2.1.1 would be highly efficient. However, should accesses be spaced by distances larger than a cache line, such microarchitectures would be underutilized. The first section following this introduction describes the simulation environment and benchmarks underlying the subsequent simulations (Section 3.2). Next, the ratio of loads and stores to computation instructions is determined to emphasize the important role of memory accesses (particularly loads) in general purpose instruction streams (Section 3.3). Moreover, Section 3.4 introduces the principle of predicated loads and highlights their influence on memory systems.

Composed of four parts, Section 3.5 analyzes references between a processor and L1D to determine the number of loads and stores that would benefit from mechanisms combining multiple accesses to the same line. It derives average results for complete instruction intervals and analyzes their variation over the course of a programs execution. Following this, Section 3.6 summarizes similar investigations based on page rather than cache line granularity. Section 3.7 employs a slightly different approach; i.e. it investigates instruction streams rather than references to analyze access patterns specific to vectorized code. In particular, it determines the ratio of unit strides, non-unit strides and indexed accesses, and breaks down non-unit strides in several groups distinguished by the distance between two vector elements. Besides that, Section 3.7.4 studies the average number of cache lines accessed per vector and the average number of elements accessed per line. Finally, this chapter gives a brief summary of its contents and conclusion drawn from them.

3.2 Simulation Environment and Benchmark Suites

The analyses in Sections 3.3 to 3.6 are based on an extended version of the gem5 simulator system (gem5). The original system was developed by the University of Michigan and its industrial partners [80]. It allows event driven simulations of freely arrangeable, parameterizable, and replaceable processor and memory models. For the purpose of the following analyses, the underlying C++ and Python framework was extended to gather and evaluate information regarding the relation of consecutive cache accesses over time. The resulting statistics are thereby able to characterize trends over a series of instruction intervals, or the whole of the executed benchmark. To avoid the pollution of statistics by OS operations, all simulations - if not explicitly stated otherwise - are performed in gem5's Syscall emulation mode. This mode simulates the execution of binaries directly from memory and only approximates the behavior of an OS. The actual statistics are generated by monitoring and analyzing the reference stream between processor and L1D at run-time. The design parameters are as follows:

- Processor:
  - in-oder, single-core
  - ARMv7 Instruction Set Architecture (ISA)
  - LSU allowing one L1D access per cycle
- L1D:
  - 4-way set-associative
  - 32 KByte capacity
  - 64 Byte line size
  - LRU replacement policy

To study access patterns of specific instructions rather than the reference stream between processor and cache, Section 3.7 employs the open source instrumentation framework Valgrind [81]. Although similar simulations could be implemented in gem5, a Valgrind interface to analyze programs on instruction level was already in place at ARM (the industrial partner of this dissertation). Moreover, as the evaluation of instruction streams is hardware independent, the emulation of a processor-cache environment can be avoided in favor of shorter computation times. To enable the analyses in Section 3.7, the framework was extended to track and categorize the memory addresses accessed by those instructions identified as vectorizable. Note that contrary to the ARMv7 ISA used for gem5 based simulations, the mentioned Valgrind interface was designed for x86.

Widely used in industry and academia for the comparison of processors, the benchmark suite provided by the Standard Performance Evaluation Corporation (SPEC) is composed of a variety of computation-intensive Integer (SPEC-Int) and Floating Point (SPEC-FP) workloads [82]. Specifically, SPEC2000 is used for gem5 based analyses, because it is already applied by gem5 developers and therefore allows simple comparisons to existing studies. In contrast, the benchmark suite's 2006 version is used for Valgrind based designs. It is considered acceptable to choose two different versions, because the performed simulations are too different in nature to be compared. In addition to SPEC, a benchmark suite developed by the MediaBench Consortium is used to investigate the differences between general purpose and media applications. MediaBench2 (MB2) includes encoder and decoder kernels for a selection of image and video compression standards [83]. The main reason for using this suite is the concern that media applications are underrepresented in SPEC (only 464.h264ref in SPEC2006) but benefit most from vector processing techniques. A brief overview of the analyzed benchmarks can be found in Tab. A.1. All benchmarks were compiled in gcc or the equivalent cross compiler for ARMv7, using the highest optimization flags (-O3).

To limit computation times particularly for SPEC2000 executed on gem5, SimPoint v3.1 was used to identify each benchmarks most representative program phase [84]. For this purpose gem5 was extended to generate information on basic block vectors as required by SimPoint. Tab. A.2 lists simulation points and weighting factors obtained for program intervals comprised of 1 billion instructions. The choice of one wide instead of multiple narrow intervals reduces the number of required gem5 simulations and mitigates the influence of secondary effects such as cache warm-ups. It can be observed that some programs are distributed over a number of almost equally representative phases while others concentrate most of their operations on a single interval. In particular, *lucas* exhibits a very homogeneous behavior with 28 intervals mostly weighted 5% or less. In contrast, *bzip2_source* spends almost 50% of its execution time inside a single interval (looping a compression algorithm over its workload). In case of multiple intervals exhibiting the same weighting factor, the earliest simulation point was chosen to reduce the computations require when generating snapshots. To verify simulation points, the statistics introduced in Section 3.5.2 were reproduced for complete runs of selected benchmarks. In addition, a simple L2 cache was implemented and its miss-rate over the course of a programs execution recorded. Using this collection of statistics, the separation in particular program phases and corresponding weighing factors were verified. To allow fair comparisons, SimPoint was used to obtain simulation intervals for MB2, too. As the Valgrind interface did not permit the execution of specific program phases, results presented in Section 3.7 are based on MiniSPEC instead of SPEC2006's reference working set.

3.3 Ratio of Load, Store and Computation Instructions

A primary consideration for vector and superscalar architectures is the number of elements to be processed in parallel. In modern commodity processors the upper bound for this value is often given by area and energy costs related to wide datapaths. However, another limiting factor is the availability of data to be computed by functional units. As described at the begin of Chapter 1, memory accesses are executed significantly slower than computation instructions. Hence, the higher the ratio between these two instruction types and the wider the datapath, the more loads/stores need to be serviced in parallel to avoid stalls.

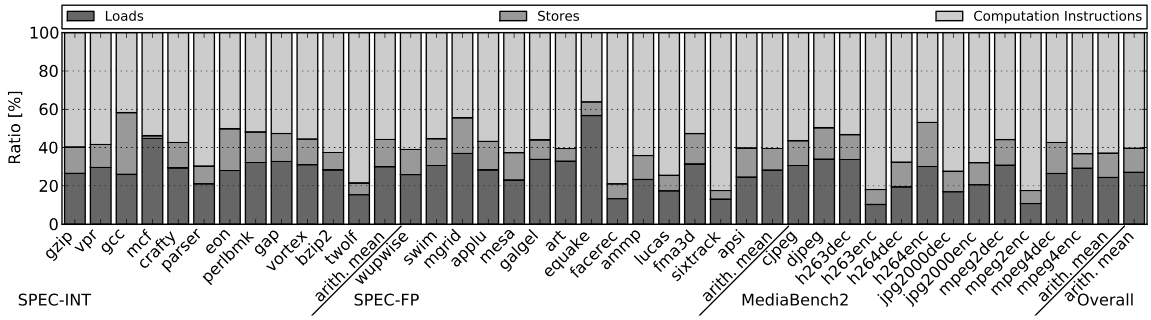


Fig. 3.1: Ratio of load, store and computation instructions

Fig. 3.1 shows the ratio of load, store and computation instructions executed by SPEC2000 and MediaBench2 suites. For the majority of benchmarks, it can be observed that approximately 40% of instructions executed are memory accesses. This ratio is significantly lower for certain SPEC-FP benchmarks as well as *h263enc* and *mpeg2enc*, indicating more structured code that frequently allows operations to reuse values residing in the processors register file.

Another observation from Fig. 3.1 is the ratio of approximately 2 to 1 between loads and stores, which is not unexpected, because most computations rely on two source and one destination operand. Major exceptions to this are *gcc*, *equake* and *mcf* that exhibit unusually high and low load/store ratios, respectively. On the one hand, the input source code provided to the GNU Compiler Collection underlying *gcc* exhibits a relatively small memory footprint. However, as the compiler shuffles through its tuples in search of the best optimizations, it generates an extensive data base, which requires an unusual high number of stores. On the other hand, the algorithm used to simulate seismic wave propagations within *equake* performs computations based on an unstructured mesh that locally resolves wavelengths using a finite element method [82]. Hence, while it requires a vast number of loads to access the unstructured grid topology forming its input data, the amount of newly generated data to be stored in memory is comparatively minor. The single-depot vehicle scheduling algorithm implemented by *mcf* exhibits a significant number of instructions that depend on multiple loads [82]. Its memory access patterns are typical for streaming applications, which load vast amounts of single-use data. Fig. A.2 confirms this observation by revealing a exceptionally high L1D miss rate for this benchmark.

Considering above values in the context of a microarchitecture designed to compute five instructions in parallel, a memory system would have to service at least two requests per cycle. In practice, the number of supported loads would need to be even higher to compensate for clustered memory accesses. This is not the case for stores, because subsequent instructions depending on them might be serviced by load forwarding (Section 1.3). Only if the capacities of mechanisms like store buffers are exceeded, stores may cause processor stalls. In conclusion, when designing a processor able to compute *n* instructions in parallel, the absolute minimum of memory accesses to be serviced per cycle is *n/4*.

3.4 Ratio of Conditional to Unconditional Loads

```
if condition
  block A                  condition block A
else                       !condition block B
  block B
```

Fig. 3.2: Example for branch predication

In order to improve the performance of out-of-order processors, modern instruction set architectures like ARMv7 support predicated instructions. Fig. 3.2 lists examples for the conditional execution of the two code blocks A and B. While conventional processors (left) would predict the branch outcome and execute either block A or B, predication (right) allows a system to execute both blocks in parallel and decide later which result to commit. The main purpose of this technique is to avoid delays due to small sequences of conditional code; i.e. allowing more instructions to be issued while waiting for a branch outcome to be determined.

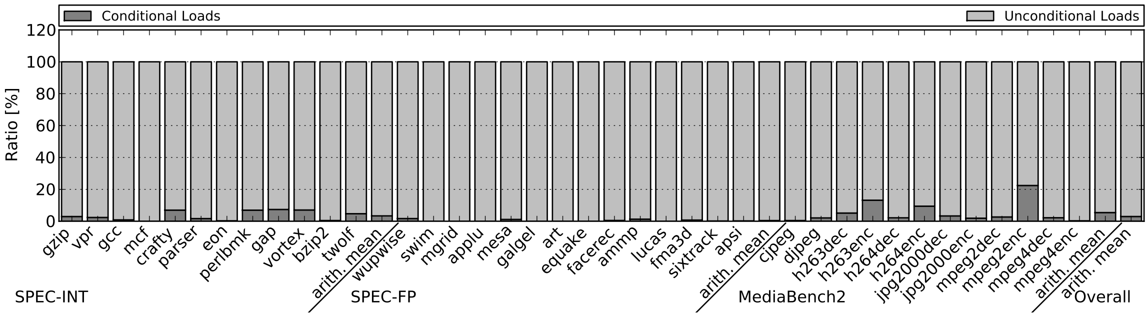


Fig. 3.3: Ratio of conditional to unconditional loads

For the design of the memory access schemes and microarchitecture optimizations in the following chapters, it is necessary to estimate the impact of conditional loads and consequently decide how much effort should be spent on optimizing their execution. Fig. 3.3 displays the ratio of conditional to unconditional loads for each analyzed benchmark. While most benchmarks exhibit values of 5% or less, four applications of the MediaBench2 suite show significantly higher results. This might be traced back to a small number of frequently executed, conditional code blocks specific to these kernels. Overall, an average of less than 5% for SPEC and MB2 is considered insufficient to justify further investigations at this point.

3.5 Consecutive Accesses per Cache Line

3.5.1 Basic Analyses

Introduced in Section 2.1.1, one approach to optimize a processor-cache-interface is to bundle multiple memory accesses to the same cache line, and service them simultaneously using small, multi-ported line buffers. This reduces the number of accesses to the cache itself and furthermore frees ports to service other outstanding requests. Besides the obvious performance impact of this approach, the small number of transistor activations required for accesses to a line buffer instead of complete cache arrays implies lower energy consumption. In order to estimate the efficiency of such schemes and define specific architectural features (e.g. the number of read/write ports per line buffer), it is necessary to be able to estimate the number of loads and stores that could potentially be bundled.

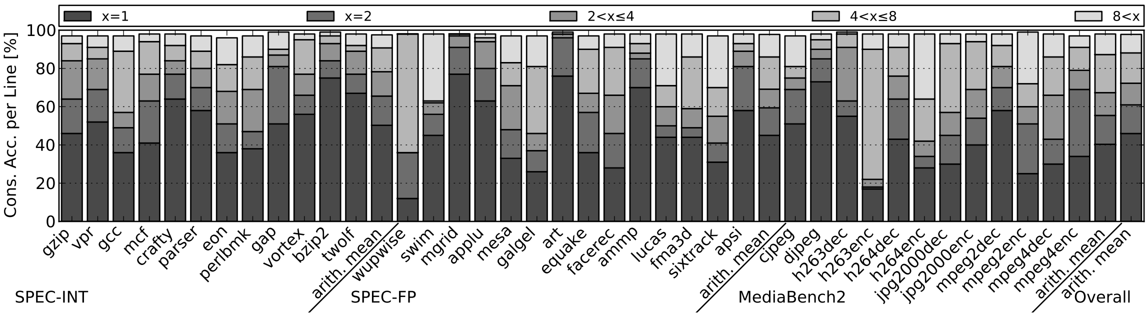


Fig. 3.4: Consecutive read accesses per cache line

Fig. 3.4 illustrates the number of read accesses to consecutive addresses received by a L1D. The corresponding simulation and design parameters are introduced and explained in Section 3.2. To improve the readability of the graphical representation, groups of consecutive accesses are combined within five bins; e.g. approximately 20% of all read accesses generated by *gzip_source* fall into the bin “2<x<=4” and are therefore part of a group of three to four consecutive request to the same cache line. It can be observed that on average less than 50% of all read requests generated by SPEC2000 and MediaBench2 benchmarks are not immediately followed by requests to the same cache lane. However, this also implies that more than fifty percent of all accesses could be combined with one or more instructions inside some form of line buffer. Exceptionally well suited for combined accesses are *wupwise* and *h263enc*, which exhibit particularly high spatial memory access locality.

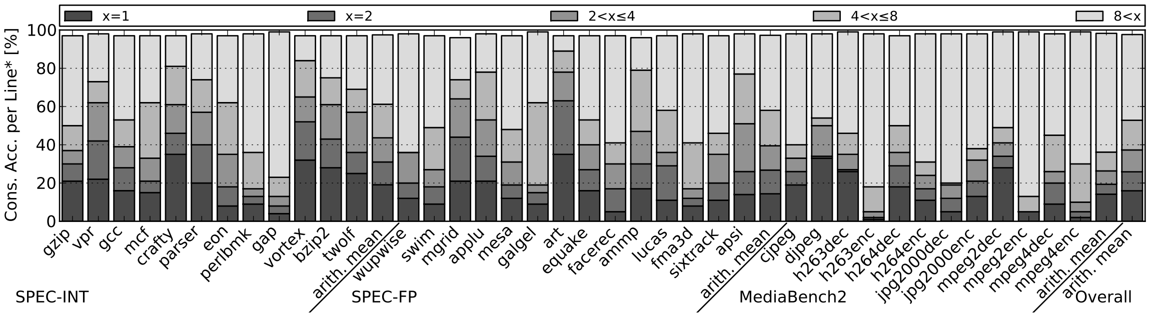
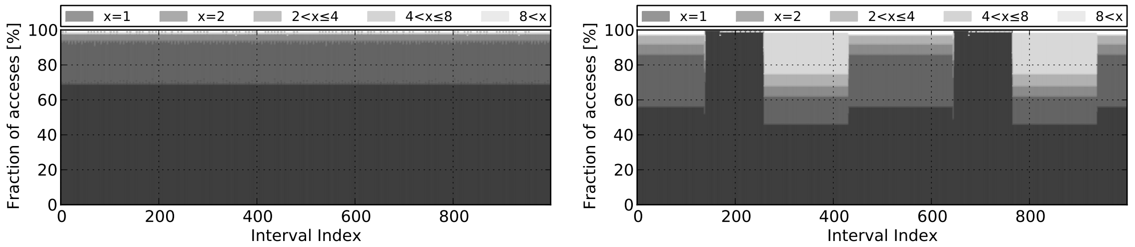


Fig. 3.5: Consecutive read accesses per cache line allowing 4 intermediate accesses

An extension to Fig. 3.4 is presented in Fig. 3.5. Based on the same simulation environment and parameters, this analysis also groups accesses to the same cache line that are separated by up to four accesses to other lines. For example, a sequence of loads to the cache lines **A**, B, C, **A**, B, D, E, **A**, C would be interpreted as one group of three accesses to A, one group of two accesses to B and three single accesses to C, D, E and C again. Fig. 3.5 shows a significant shift of loads into bins grouping larger sequences of consecutive accesses. On average, 83% of all loads can be gathered in groups of two or more accesses. In consequence, a mechanism to reorder loads, or even to stall their execution while waiting for additional instructions to be issued, would significantly improve the number of loads benefiting from combined accesses.

Similar analyses with respect to write accesses reveal that approximately 75% and 90% of consecutive stores can be bundled to groups of 2 or more accesses, when allowing zero and four intermediate accesses, respectively (Fig. A.1). Although this indicates a higher suitability of stores for combined accesses, as discussed in Section 3.3, it is still considered more beneficial to optimize architectures for load rather than store instructions. Chapter 4 proposes several architectural features; i.e. an Input Buffer that allows re-ordering of cache accesses, an Arbitration Unit capable of grouping multiple loads to the same cache line, and a modified store and merge buffers to exploit the locality of stores.

3.5.2 Variation over Time



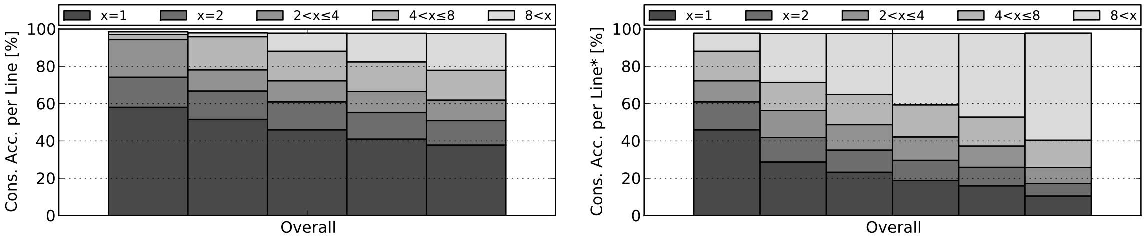
(a) Invariant behavior (sixtrack)

(b) Phased behavior (swim)

Fig. 3.6: Consecutive read accesses per cache line over time

To observe the variation of the results presented in the previous section over time, this section uses the same simulation parameters, but divides the analyzed instruction interval into segments of 1 million instructions. The behavior of the analyzed benchmarks can be categorized as either invariant or phased. The former can be observed for the majority of benchmarks and is illustrated by Fig. 3.6a on the example of *sixtrack*. It emphasizes that the average values previously presented in Fig. 3.4 are highly representative for this and similar benchmarks. The later behavior indicates the presence of loop structures or repetitive function calls. In particular, Fig. 3.6b exhibits three distinctive phases that are repeated twice within the analyzed instruction interval. This emphasizes the importance of identifying and utilizing large instruction intervals that are representative for the majority of a benchmarks execution time (Section 3.2).

3.5.3 Sweep of Analysis Parameters



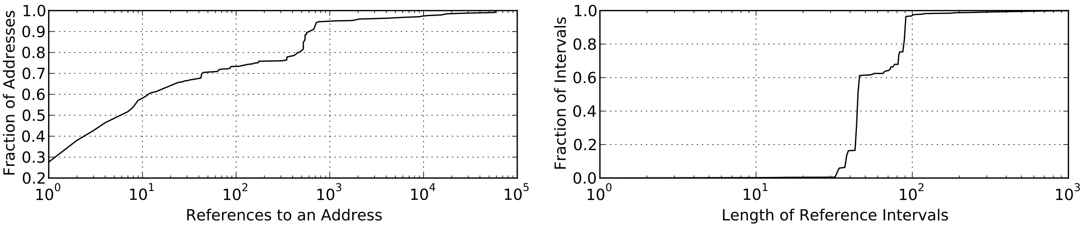
(a) Cache Line Size (left to right): 16, 32, 64, 128 and 256 byte

(b) Number of allowed Intermediate Request (left to right): 0, 1, 2, 3, 4 and 8

Fig. 3.7: Consecutive read accesses per cache line (parameter sweep)

The results presented in Sections 3.5.1 and 3.5.2 are obtained for a fixed set of parameters introduced and explained in Section 3.2. Conversely, Fig. 3.7a and (b) illustrate values for varying cache line sizes and the allowed number of intermediate requests, respectively. As expected due to the higher number of elements per cache line, the number of combinable loads increases with wider line sizes. Similarly, allowing more intermediate request results in higher possibilities for combinable accesses to the same line. However, while longer cache lines may improve the effect of mechanisms that exploit intra cache line locality, they also increase the power consumption of conventional cache accesses by activating more transistors per access. Although this effect can be mitigated for the L1D-processor interface by the application of sub-banking (Section 2.1.1), it still influences transfers between different cache levels (lines sizes are usually consistent within a memory hierarchy; Section 1.1.1). Wider lines also influence miss-rates as the increased amount of data may either pollute the cache or act as implicit pre-fetching (Fig. A.2). Moreover, multi-core systems are often limited to a maximum line size of 128 byte, to allow the application of common memory coherency policies (Section 1.4.2). In conclusion, simulations results presented in this and the following chapters employ 64 byte cache lines by default.

3.5.4 Cache Independent Values



(a) Number of References to an Address

(b) Length of Reference Intervals

Fig. 3.8: Reference count and intervals for L1D loads (swim)

Fig. 3.8a and (b) illustrate the number of times a certain address is referenced and the interval between two consecutive references to the same address. It can be observed that approximately 40% of all addresses are accessed 10 times or more and that about 60% of these repeated accesses are no further then 50 references apart. While both figures are obtained for *swim*, all analyzed benchmarks exhibit similar results. The results are in accordance with the spatial and temporal locality expected from most programs. An in depth analysis based on these results has not been conducted thus far, because they were primarily used to give initial ideas about the structure of the analyzed benchmarks; e.g. the question: “Do general purpose programs like SPEC2000 benchmarks exhibit a significant amount of re-referenced data?”. However, further analyses based on instruction streams evaluated in Valgrind are discussed in Section 3.7.

3.6 Consecutive Accesses per Page

Fig. 3.9 illustrates simulation results similar to those obtained in Section 3.5, but employing page rather than cache line granularity. In average 70% of consecutive loads access the same page. Allowing up to four intermediate accesses increases this ratio to more than 95%. Based on this, Chapters 4 and 5 introduce Page-Based Memory Access Grouping and Page-Based Way Determination. The former describes the idea to deliberately limit all L1D accesses within a particular cycle to a single page. This allows the utilization of simpler and more energy efficient components without causing significant performance degradation. The latter extends this idea and proposes a way determination scheme that achieves high scalability and low energy consumption, by assuming that the majority of parallel executed L1D accesses map to the same page.

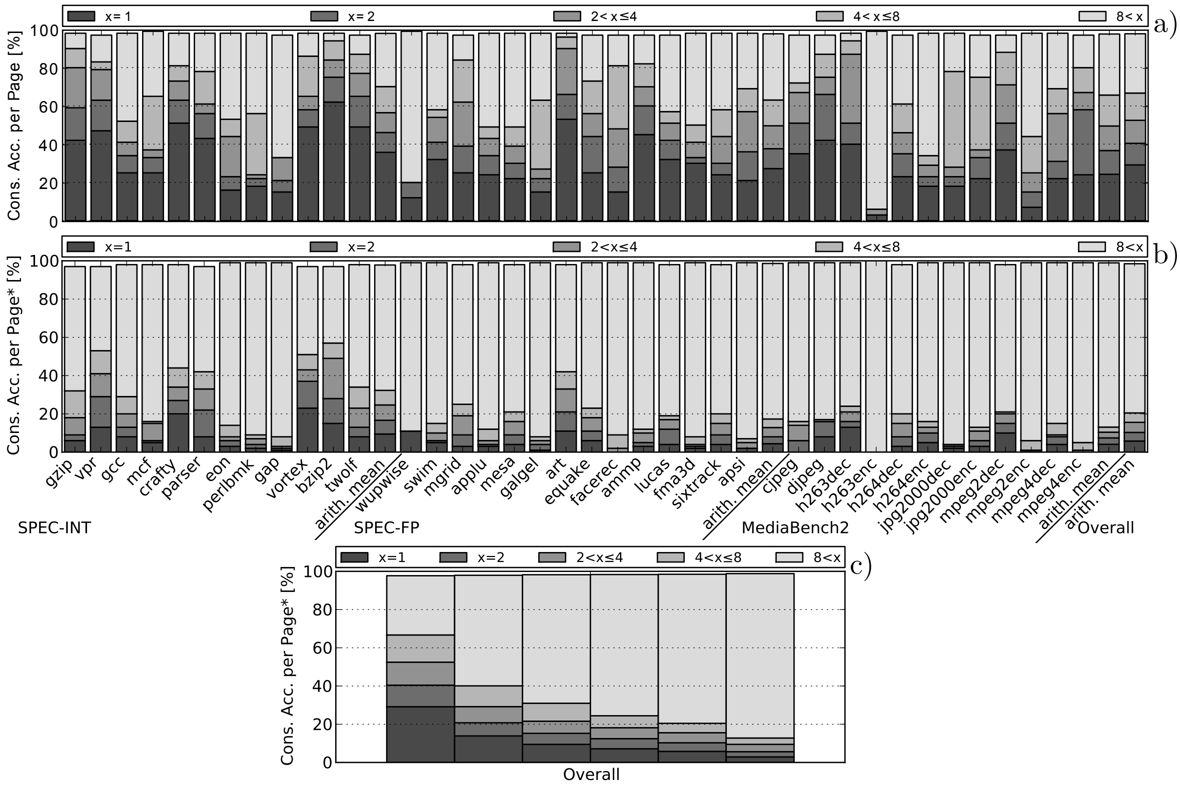


Fig. 3.9: Consecutive read accesses per page allowing zero (a), four (b), and n=0/1/2/3/4/8 (c) intermediate accesses

3.7 Influence of Vectorization on Memory Access Patterns

3.7.1 Overview

Based on instruction streams rather than the references between processor and L1D analyzed so far, this section investigates characteristics of vector memory accesses and attempts to estimate consequences for vector execution units that aim for energy efficiency. The following results were obtained from a Valgrind interface that monitors loads and stores within scalar instruction streams and interprets them as vector memory accesses.

```
for (i=0; i<8; i++) {
    for (j=0; j<4; j++) {
        ld A[i,j]
        ld B[i,j]
        add A[i,j], A[i,j],B[i,j]
        st A[i,j]
    }
}
```

(a) Scalar Version

```
for (i=0; i<8; i++) {
    vLd4 A[i]
    vLd4 B[i]
    vAdd4 A[i], A[i],B[i]
    vSt4 A[i]
}
```

(b) Vectorized inner Loop

```
for (j=0; j<4; j++) {
    vLd8 A[j]
    vLd8 B[j]
    vAdd8 A[j], A[j],B[j]
    vSt8 A[j]
}
```

(c) Vectorized outer Loop

Fig. 3.10: Examples of vectorization strategies

A major challenge for the analysis of scalar code is the decision on how to vectorize given instructions. For example, Fig. 3.10 presents two approaches for the vectorization of an instruction sequence encompassed by two nested loops. Assuming the absence of inter-instruction dependencies, an ideal compiler could vectorize either the inner or outer loop. The former option is easy to implement and common within currently available compilers. In contrast, the later requires additional profiling effort, but yields a higher degree of vectorization; i.e. only 16 instead of 32 instructions remain and a vector execution unit could compute eight instead of four operations in parallel. However, while the Valgrind interface employed here allows the identification of nested loops, the nature of scalar instruction streams complicates investigations based on the vectorization of outer loops. Hence, for the following simulations, instructions are always vectorized with respect to their innermost loop. Furthermore, only those loops are vectorized that do not exhibit inter-instruction dependencies.

3.7.2 Categorization and Distribution of Vector Accesses

In order to estimate to what extent cache optimization techniques would benefit from adjustments to vector instructions, it is necessary to understand how often certain memory access patterns occur. For example, if the majority of vectorized accesses are unit-strides, mechanisms that merge access to the same line would be highly efficient (Section 1.4.3 for general categories of vector access patterns). However, should elements of vector accesses be spaced by distances exceeding a cache line, such microarchitecture would be underutilized. For this purpose, the Valgrind interface introduced in the previous section was modified to monitor the addresses accessed by instances of a given instruction. In this context, the term instance describes all occurrences of an instruction within a specific context (i.e. a sequence of branches leading up to it). For example, an instance might be inside a function without a loop structure. Assuming that the function would be independently called by two different loops, a compiler might decide to replicate it to allow the vectorization of both loops. This process might deliver the address stream “1000; 1004; 1008; 1012” for one instance of a specific load. Depending on the corresponding data type, this load would be categorized as unit stride (data type size = 32-bit) or non-unit stride (data type size < 32-bit). The remaining categories are single, static and indexed access, which describe instances that are executed only once, accesses the same address multiple times, and exhibit incoherent access patterns, respectively.

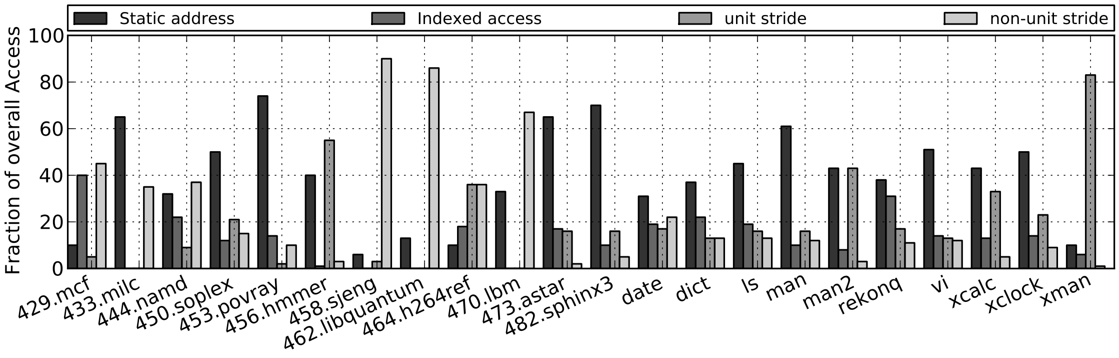


Fig. 3.11: Distribution of memory access patterns (loads)

Fig. 3.11 shows load access patterns for selected SPEC2006 benchmarks and a small set of linux based programs. The additional programs are supposed to indicate behavior of non SPEC code, but are not meant to be highly representative (their results depend on the chosen working sets and vary slightly from system to system). A similar graph for stores can be found in Fig. A.3. It is not listed here, because the observable patterns are similar to those in Fig. 3.11, and as stated in Section 3.3, it is more efficient to optimize memory systems for loads rather than for stores. The average values for single, static, indexed, unit stride and non-unit stride accesses are 3%,40%,13%,20% and 24% for loads and 5%,55%,7%,13% and 20% for stores. Hence, the number of loads/stores executed only once is negligible, except for *date* and *dict* that show values of 10% and 15%. However, it can be assumed that both programs are simply too small to exhibit extensive loop structures suitable for vectorization. Also not vectorizable, static accesses are highly represented in most benchmarks (except *458.sjeng*). Those accesses can be traced back to the limited size of register files, which imposes the requirement for repeated data transfers between processor and cache.

Vectorizable and therefore of particular interest for this study are the remaining categories. It can be observed that indexed accesses are least common, followed by unit strides and non-unit strides. Remarkable in this context are *458.sjeng* and *462.libquantum* that exhibit highly structured memory access patterns primarily consisting of non-unit strides. In conclusion, the relatively low number of indexed accesses suggests that designers should consider spending more time optimizing non-unit strides and implement indexed accesses in a more rudimentary form. Furthermore, as unit strides are fairly frequent and easy to implement, additional optimizations for them should be investigated. In addition to the analysis given here, Chapter 6 describes the development of a Vector Benchmark Suite, which involves the optimizations of several general purpose algorithms to study their suitability for vectorization.

3.7.3 Distribution of Non-Unit Stride Distances

As mentioned in the previous section, non-unit strides are the most common vectorizable access pattern observed within the analyzed benchmarks. A determining factor for the design of microarchitectures optimized for non-unit strides is the address distance between consecutive accesses. Fig. 3.12 lists an extract of results obtained for one of the most representative benchmarks, namely *464.h264ref*. Besides the general categorization of access patterns discussed in Section 3.7.2, the listing includes non-unit stride distances grouped by distance. For example, a vector access striding over eight elements between two consecutive accesses would be placed inside the logarithmic bin [8..15] and the linear bin [8] (stride distance in number of elements $= \frac{\text{address}-\text{previous address}}{\text{size of data type}}$). Each entry within a bin is further described by the ratio of accesses showing this particular pattern relative to the overall access count, the total number of its appearances, the average number of elements consecutively accessed by it, and the average number of accesses covered by it for the underlying instance.

Benchmark	Bin	%Accs
429.mcf	1024..inf	29%
	-31..-16	10%
450.soplex	4..7	5%
	-1	6%
433.milc	6	20%
	256.. 511	13%

Benchmark	Bin	%Accs
444.namd	4	12%
	6	8%
	12	13%
458.sjeng	128..255	89%
462.libquantum	2	85%
470.lbm	64..127	11%
	128..255	23%

Tab. 3.1: Major exceptions concerning non-unit stride distances

While the example in Fig. 3.12 is considered representative for all analyzed benchmarks, Tab. 3.1 lists major exceptions. In general, the analyzed code sample shows high access concentrations within a small number of bins. Those bins primarily correspond to positive distance less than or equal 256 elements. Their entries usually consist of vectors composed of 10 to 100 elements on average, which are executed several dozen to several hundred times. The results obtained for stores exhibit a similar distribution of patterns.

In conclusion, the high number of vectors composed of more than 10 elements confirms the suitability of the investigated benchmarks for vectorization in general. The high number of unit-strides (≈36%; Fig. 3.12) supports the previously discussed idea of implementing circuitry to merge accesses to the same cache line (Section 3.5). However, the low number of non-unit strides with distances between two and eight elements (<2%; Fig. 3.12) implies that those circuits would not directly benefit non-unit strides. Another observation is that the majority of vector accesses concentrates strides inside bins smaller than 256 elements (≈94%; Fig. 3.12). Hence, it can be assumed that most of those accesses map to the same page, which supports the ideas of Page-Based Memory Access Grouping and Page-Based Way Determination proposed in the following chapters.

3.7.4 Composition of Vector Memory Accesses

Number of Coalesce Accs.	Proportion of all Accesses	Total Number of Times Executed
1	41%	2020005
2	4%	89219
3	1%	16307
4	10%	119963
5	1%	6525
6	9%	70436
7	4%	25438
8	32%	200256

Average Number of Groups per Vector Access: 4.0
Average Number of Elements per Group : 1.9

(a) Coalesce Vector Accesses

Number of Coalesce Accs.	Proportion of all Accesses	Total Number of Times Executed
1	41%	2054504
2	4%	104923
3	2%	26164
4	10%	128354
5	1%	10721
6	8%	65002
7	1%	8664
8	32%	200256

Average Number of Groups per Vector Access: 4.1
Average Number of Elements per Group : 1.9

(b) Consecutive Coalesce Vector Accesses

- Maximum allowed Vector Length : 8.0
- Average observed Vector Length : 7.8
- Total Number of analyzed Accesses : 4962571
- Total Number of identified Vector Accesses : 639849

Tab. 3.2: Groups of coalesced elements within vector accesses for *464.h264ref*

While Section 3.7.3 analyzes access patterns in their entirety, this section focuses only on vectorizable patterns (i.e. excluding single and static accesses) and introduces limits on the maximum supported vector length (8-element SIMD datapath; Section 1.4). The support of eight elements per vector is considers reasonable within the context of microprocessors, because it allows sufficient speed ups and only a moderate risk of underutilization by general purpose code. Equivalent to Fig. 3.12 of the previous section, Tab. 3.2a and 3.2b list

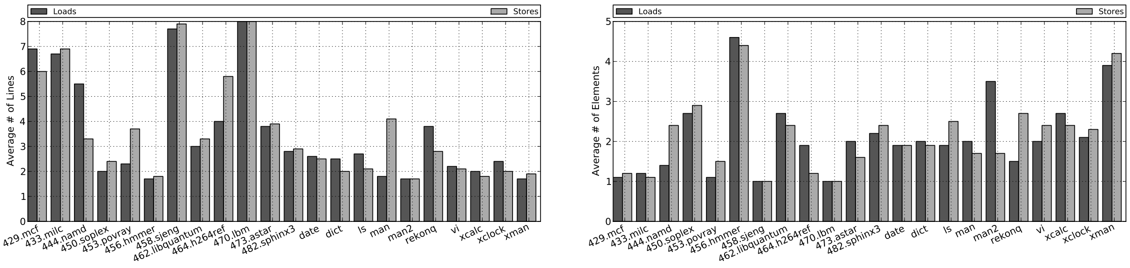
```
                    %Accs       Cnt   AvgAcc %Cvrg
Single access :       0%,    21347
Static address:      10%,    12591,       41, 100%
Indexed access:      18%,      912,     1085, 100%
Strides
  unit stride     : 36%,     28809,       68, 100%
  non-unit stride: 36%,      26359,       75, 100%
  Logarithmic bins
                    %Accs       Cnt  AvgAcc %Cvrg                    %Accs       Cnt  AvgAcc %Cvrg
    [    1..    1]: 36%,     28809,       68, 100%  [    -1..    -1]:  0%,       227,        4,  96%
    [    2..    3]:  0%,       166,       34,  99%  [    -3..    -2]:  0%,        26,        2,  76%
    [    4..    7]:  0%,       358,       30,  99%  [    -7..    -4]:  1%,      4017,        8,  88%
    [    8..   15]:  0%,       104,       28,  99%  [   -15..    -8]:  0%,        40,        3, 100%
    [   16..   31]:  0%,        89,       47, 100%  [   -31..   -16]:  0%,        27,       55,  99%
    [   32..   63]:  0%,      2573,        9, 100%  [   -63..   -32]:  0%,       273,        8,  85%
    [   64..  127]: 11%,      8808,       71, 100%  [  -127..   -64]:  0%,         0,        0,   0%
    [  128..  255]: 23%,      9192,      138, 100%  [  -255..  -128]:  0%,         3,        2, 100%
    [  256..  511]:  0%,       322,        4, 100%  [  -511..  -256]:  0%,         0,        0,   0%
    [  512..1023]:  0%,         47,        3, 100%  [-1023..  -512]:  0%,         3,        2, 100%
    [1024.. inf]:  0%,          77,        2, 100%  [ -inf..-1024]:  0%,         7,        2,  83%
  Linear bins
              %Accs       Cnt  AvgAcc %Cvrg               %Accs       Cnt  AvgAcc %Cvrg
    [   1]: 36%,     28809,       68, 100%  [   -1]:  0%,       227,        4,  96%
    [   2]:  0%,       105,        4,  93%  [   -2]:  0%,        24,        3,  75%
    [   3]:  0%,        61,       86, 100%  [   -3]:  0%,         2,        2, 100%
    [   4]:  0%,        36,       64, 100%  [   -4]:  1%,      3861,        8,  88%
    [   5]:  0%,         4,        2, 100%  [   -5]:  0%,         1,        2, 100%
    [   6]:  0%,        88,       17, 100%  [   -6]:  0%,       155,        7, 100%
    [   7]:  0%,       230,       30,  99%  [   -7]:  0%,         0,        0,   0%
    [   8]:  0%,        22,        2,  90%  [   -8]:  0%,         1,        2, 100%
    [   9]:  0%,         5,        6,  81%  [   -9]:  0%,         0,        0,   0%
    [  10]:  0%,        15,        9,  99%  [  -10]:  0%,        12,        8, 100%
    [  11]:  0%,         0,        0,   0%  [  -11]:  0%,         0,        0,   0%
    [  12]:  0%,         4,      536, 100%  [  -12]:  0%,        27,        2, 100%
    [  13]:  0%,         0,        0,   0%  [  -13]:  0%,         0,        0,   0%
    [  14]:  0%,        51,       10,  95%  [  -14]:  0%,         0,        0,   0%
    [  15]:  0%,         7,        2, 100%  [  -15]:  0%,         0,        0,   0%
    [  16]:  0%,        67,       18, 100%  [  -16]:  0%,         0,        0,   0%
    [  17]:  0%,         0,        0,   0%  [  -17]:  0%,         0,        0,   0%
    [  18]:  0%,         0,        0,   0%  [  -18]:  0%,         0,        0,   0%
    [  19]:  0%,         0,        0,   0%  [  -19]:  0%,         0,        0,   0%
    [  20]:  0%,        20,       44, 100%  [  -20]:  0%,        16,       92,  99%
    [  21]:  0%,         0,        0,   0%  [  -21]:  0%,         0,        0,   0%
    [  22]:  0%,         0,        0,   0%  [  -22]:  0%,         0,        0,   0%
    [  23]:  0%,         0,        0,   0%  [  -23]:  0%,         0,        0,   0%
    [  24]:  0%,         2,     1070, 100%  [  -24]:  0%,         7,        2, 100%
    [  25]:  0%,         0,        0,   0%  [  -25]:  0%,         0,        0,   0%
    [  26]:  0%,         0,        0,   0%  [  -26]:  0%,         0,        0,   0%
    [  27]:  0%,         0,        0,   0%  [  -27]:  0%,         0,        0,   0%
    [  28]:  0%,         0,        0,   0%  [  -28]:  0%,         4,        3, 100%
    [  29]:  0%,         0,        0,   0%  [  -29]:  0%,         0,        0,   0%
    [  30]:  0%,         0,        0,   0%  [  -30]:  0%,         0,        0,   0%
    [  31]:  0%,         0,        0,   0%  [  -31]:  0%,         0,        0,   0%
    [  32]:  0%,         0,        0,   0%  [  -32]:  0%,         6,        2, 100%
    [  33]:  0%,         0,        0,   0%  [  -33]:  0%,         0,        0,   0%
    [  34]:  0%,         0,        0,   0%  [  -34]:  0%,         0,        0,   0%
    [  35]:  0%,         0,        0,   0%  [  -35]:  0%,         0,        0,   0%
    [  36]:  0%,        58,        2, 100%  [  -36]:  0%,         0,        0,   0%
    [  37]:  0%,         1,        2, 100%  [  -37]:  0%,         0,        0,   0%
    [  38]:  0%,         0,        0,   0%  [  -38]:  0%,         0,        0,   0%
    [  39]:  0%,         0,        0,   0%  [  -39]:  0%,         0,        0,   0%
    [  40]:  0%,         2,        2, 100%  [  -40]:  0%,         0,        0,   0%
    [  41]:  0%,         0,        0,   0%  [  -41]:  0%,         0,        0,   0%
    [  42]:  0%,         0,        0,   0%  [  -42]:  0%,         0,        0,   0%
    [  43]:  0%,         0,        0,   0%  [  -43]:  0%,         0,        0,   0%
    [  44]:  0%,         0,        0,   0%  [  -44]:  0%,         1,        3,  75%
    [  45]:  0%,         0,        0,   0%  [  -45]:  0%,         0,        0,   0%
    [  46]:  0%,      2496,       10, 100%  [  -46]:  0%,       256,        9,  85%
    [  47]:  0%,         0,        0,   0%  [  -47]:  0%,         0,        0,   0%
    [  48]:  0%,         0,        0,   0%  [  -48]:  0%,         8,        2, 100%
    [  49]:  0%,         0,        0,   0%  [  -49]:  0%,         0,        0,   0%
    [  50]:  0%,         0,        0,   0%  [  -50]:  0%,         0,        0,   0%
    [  51]:  0%,         0,        0,   0%  [  -51]:  0%,         0,        0,   0%
    [  52]:  0%,         1,        2, 100%  [  -52]:  0%,         0,        0,   0%
    [  53]:  0%,         0,        0,   0%  [  -53]:  0%,         0,        0,   0%
    [  54]:  0%,         0,        0,   0%  [  -54]:  0%,         0,        0,   0%
    [  55]:  0%,         0,        0,   0%  [  -55]:  0%,         0,        0,   0%
    [  56]:  0%,         1,        2, 100%  [  -56]:  0%,         2,        3, 100%
    [  57]:  0%,         0,        0,   0%  [  -57]:  0%,         0,        0,   0%
    [  58]:  0%,        14,       44, 100%  [  -58]:  0%,         0,        0,   0%
    [  59]:  0%,         0,        0,   0%  [  -59]:  0%,         0,        0,   0%
    [  60]:  0%,         0,        0,   0%  [  -60]:  0%,         0,        0,   0%
    [  61]:  0%,         0,        0,   0%  [  -61]:  0%,         0,        0,   0%
    [  62]:  0%,         0,        0,   0%  [  -62]:  0%,         0,        0,   0%
    [  63]:  0%,         0,        0,   0%  [  -63]:  0%,         0,        0,   0%
    [  64]:  0%,         0,        0,   0%  [  -64]:  0%,         0,        0,   0%
    [>64]: 35%,      18446,      103, 100%  [<-64]:  0%,        13,        2,  89%
```

- %Accs : ratio of accesses showing pattern x to overall accesses count
- Cnt : total number of times pattern x appears
- AvgAcc : average number of elements consecutively accessed with pattern x
- %Cvrg : average number of accessed covered by pattern x for an instance
- Note, coverage of less than 100% indicates that an instance exhibits a pattern for the majority of its executions but not at all times. Only those patterns with coverage of at least 75% are considered non-unit stride, the remainder is interpreted as indexed access. The parameter is meant to indicate that more memory accesses could be vectorized to non-unit strides by replicating certain instructions within a program.

Fig. 3.12: Extract: distribution of load access patterns for *464.h264ref*

results obtained for loads of *464.h264ref*. It can be observed that the benchmark's average vector length of 7.8 elements comes very close to the maximal allowed value of 8.0. Hence, *464.h264ref* shows a good utilization of the given SIMD datapath, and only a minority of short vectors or vector fragments (remaining after subdividing long vectors into SIMD sized segments).

Tab. 3.2a differs from 3.2b by grouping all elements of a vector that access the same cache line instead of only grouping those that are on consecutive positions within the underlying vectors. The reason for this separation is the assumption that it is more energy efficient to perform address comparisons only between consecutive rather than all elements. Remarkable in this context, Tab. 3.2a exhibits only a slightly higher proportion of vectors with six or seven consecutive accesses. As this is also the case for stores and all other investigated benchmarks, it is highly recommended to base the identification of consecutive access on comparisons between consecutive and not all vector elements (Arbitration Unit; Section 4.3.3).

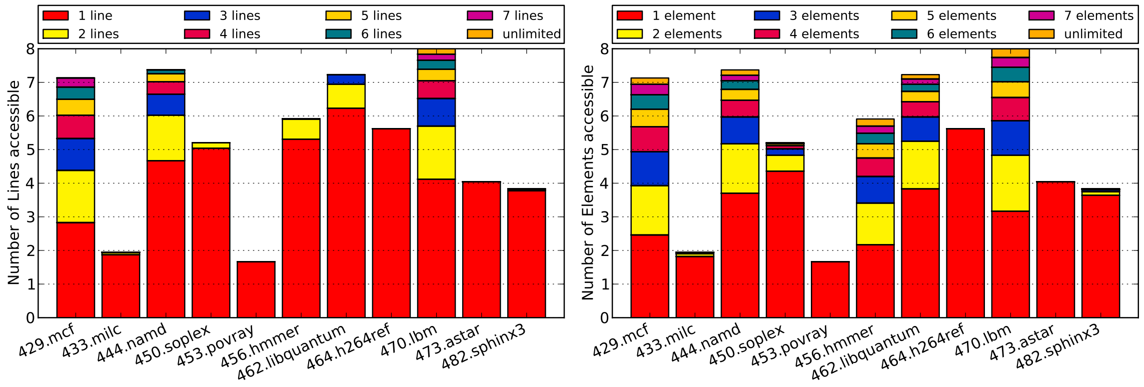


(a) Average Number of Lines accessed per Vector (b) Average Number of Elements accessed per Line

Fig. 3.13: Average values for consecutive load and store accesses

Fig. 3.13a and (b) show the average number of cache lines accessed per vector and the average number of elements accessed per line. It can be observed that most benchmarks exhibit vectors accessing approximately four different lines with an average of two elements per line. Exceptions are *429.mcf* and *470.lbm* with very low degrees of shared line accesses. The behavior of both benchmarks is not unexpected, because the previous section already identified the bins [64..127],[128..255] and [-31..-16],[1024..inf] as dominant for their corresponding non-unit strides. Concluding from Fig. 3.13, a vector execution unit should support accesses to at least four different cache lines per cycle, each line access servicing a minimum of two elements.

Fig. 3.14 illustrates the maximal speedups to be achieved from vectorizing selected SPEC2006 benchmarks for a SIMD width of eight. The speedups are based on an ARM internal study on the vectorization potential of scalar code. Although specific details can not be disclosed at this point, the part of the study that is relevant here assumes that all dependence free inner loops are fully vectorizable (including direct and indirect function, but excluding system calls). Individual segments within both graphs represent the performance degradation due to a limited number of accessible cache lines per cycle and a limited number of element accessible per line per cycle, respectively. For example, the bottom most bar in Fig. 3.14a



(a) Limited Number of accessible Lines (b) Limited Number of accessible Elements per Line

Fig. 3.14: Speedup degradation due to limited number of memory accesses per cycle

illustrates a maximal performance gain of approximately 2.7x (270%) for the execution of *429.mcf* when the corresponding vector execution unit is limited to one load and one store per cycle. The choice of merging results of loads and stores is based on the generally lower impact of stores on the memory system. Concluding from both graphs, while several benchmarks are hardly sensitive to memory system limitations, others benefit significantly from corresponding optimizations. In particular, *456.hmmer* requires only two line accesses per cycle but up to eight element accesses per line per cycle to reach its full potential.

3.8 Concluding Remarks

This chapter investigated memory access patterns to deduce conclusions relevant to the design of energy efficient memory systems. The first section introduced the underlying simulation environment and design parameters. The analyses of references between processor and L1D allowed the following observations (Sections 3.3 to 3.6):

- A processor should be able to service at least $n/4$ memory accesses per cycle to allow the efficient computation of n instructions in parallel
- An average of approximately 5% of loads are issued conditionally; the potential impact in terms of additional memory pressure is considered negligible
- 50% of the analyzed loads and even more stores would profit from mechanisms that combine multiple accesses to the same line
- Allowing a small number of memory references between mergeable loads/stores further improves this idea
- An even higher access locality can be observed on page granularity; supporting the ideas of Page-Based Memory Access Grouping and Page-Based Way Determination

Based on instruction streams rather than references between processor and L1D cache, the second part of this chapter focused on the analysis of vector memory access patterns. The results of Section 3.7 can be summarized to:

- Non-unit strides are the most common vectorizable access pattern; hence, they should be the primary target for optimization efforts

- Unit strides are less common; however, their simplicity allows them to benefit even from basic optimizations
- Indexed accesses are the least common and most complex vectorizable access pattern; consequently, optimizations are likely to yield comparatively minor performance benefits, but introduce significant hardware and energy overheads
- The majority of vector accesses do not cross page boundaries
- A vector execution unit designed for an eight element wide SIMD datapath should support accesses to at least four different cache lines per cycle, each line access servicing a minimum of two elements

The engineering contributions underlying this chapter are described in Section 3.2. They include modifications to gem5 allowing it to gather and evaluate information regarding the relation of consecutive cache accesses over time, and to generate statistics on basic block vectors used to identify relevant simulation intervals in SimPoint. Furthermore, a Valgrind based evaluation framework was extended to track and categorize the memory addresses accessed by instructions that were identified as vectorizable.

4 | Page-Based Memory Access Grouping

4.1 Motivation

The conclusions drawn from the analyses of vector memory access patterns in the previous chapter suggest that vector execution units would greatly benefit from the ability to service multiple accesses per cycle. In particular, Section 3.3 emphasizes the importance of memory references by stating that they account for approximately 40% of all instructions with a load:store ratio of approximately 2:1. Performance estimates indicate that an L1D interface servicing an 8-element wide SIMD datapath should support accesses to at least four different cache lines per cycle, each line servicing a minimum of two elements (Section 3.7.4). By mitigating the performance bottleneck between processor and memory, such a cache interface can improve performance as well as energy consumption. The latter is the result of improved resource usage leading to faster and more efficient computations, and therefore to more time spend in energy conserving states.

The ARM CortexA-15 can simultaneously load and store one 32-bit register each cycle (addresses must be aligned at 64-bit boundaries) [8]. Similarly, Intel's Core and Nehalem architectures support one 128-bit load and one 128-bit store in parallel. More powerful in terms of L1 accesses are Intel's Ivy Bridge and AMD's Family 15h (Bulldozer) microarchitectures, which allow up to two 128-bit loads and one 128-bit store per cycle [5, 6]. As far as assessable from the corresponding reference manuals, all of these microarchitectures require some multi-ported components to realize parallel memory accesses. For example, the CortexA-15 implements multi-ported address translation circuitry in form of separate TLBs for loads and stores (see Section 2.1.2 on Mirroring), while using a set of single-ported banks for its L1D; the latter requires loads and stores to access different banks in order to be serviced in parallel. Ivy Bridge removes this limitation by providing one load and one load/store port for its L1D. The key feature of the following Multiple Access Low Energy Cache interface (MALEC) is the efficient utilization of single-ported structures to avoid the energy consumption induced by replication and/or physical multi-porting (Section 2.1.2). The primary contributions of MALEC are:

- Deliberate restriction of accesses to only 1 page per cycle
- Re-use of TLB accesses to service multiple memory references to the same page
- A mechanism to accelerate certain translations, while delaying others
- Simplification of lookup structures for store and merge buffers in the context of multiple parallel loads
- Allowing a set of narrow comparators to identify loads accessing the same cache line

The following sections introduce the basic cache configuration underlying MALEC and

Page-Based Way Determination (Section 4.2, Chapter 5), describe the modification of existing and the design of new components as part of MALEC (Section 4.3), present performance and energy evaluations (Section 4.4), and perform a sensitivity analysis (Section 4.5). This chapter closes with a brief summary of its contents and conclusion drawn from them in Section 4.6.

4.2 The Underlying L1 Data Cache Interface

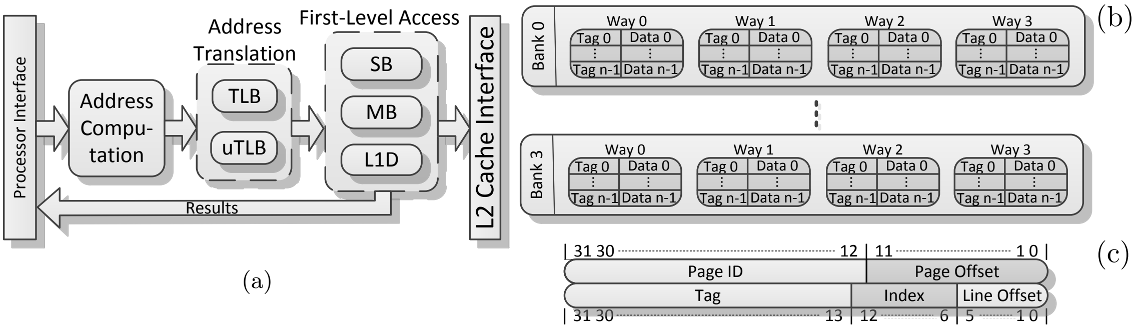


Fig. 4.1: Processor-L1D interface (a), four independent 4-way set-associative cache banks (b), and naming conventions for memory address bit fields (c)

To ease the understanding of the components introduced by Page-Based Memory Access Grouping in the next section, Fig. 4.1 summarizes the characteristics and parameters of the underlying L1D interface. Detailed descriptions of the components depicted in Fig. 4.1a are given in Section 1.1.2. In summary, each issued load and store undergoes address computation, address translation and data access. Application specific virtual addresses are translated by TLBs and if necessary lower level page tables into physical addresses. The sequential implementation of address translation and L1 access as displayed in Fig. 4.1a is strongly suggested for Page-Based Memory Access Grouping and Page-Based Way Determination (Chapter 5). In particular, the components to be introduced in the next section are designed to hide their own latency by operating in parallel to TLB and prior to L1 lookups. The separate implementation of Page-Based Memory Access Grouping prior to parallel TLB and L1 lookups would be possible, but less efficient due to higher memory access latencies. As energy oriented memory systems favor physically indexed physically tagged (PIPT) caches, the following section assumes the use of a PIPT data cache (Section 1.2). The micro TLB (uTLB) shown in Fig. 4.1a is a small, fast and energy efficient structure that is commonly implemented to reduce the latency and energy consumption of address translations. While this structure is not required for the proposals in Chapters 4 and 5, it particularly improves the efficiency of Page-Based Way Determination. Finally, store (SB) and merge buffers (MB) are widely used to allow the speculative execution of stores and exploit their memory address locality to reduce the number of L1 accesses (Section 1.1.2). As both structures significantly alter the frequency and locality of L1 accesses, they are considered relevant for the evaluation in Section 4.4.

Fig. 4.1b shows a 4-way set-associative cache comprised of four independent banks. This structure is based on Fig. 1.6 in Section 1.2. State-of-the-art designs usually employ 2, 4, 8 or even 16 L1D banks. While a higher number of banks can potentially reduce the energy consumption and latency per access, as well as the probability for bank conflicts (two or more parallel accesses competing for the same bank), it also requires more complex routing networks. The choice of four banks in Chapters 4 and 5 is a trade-off between performance, energy and complexity considerations. Similarly, the associativity of L1Ds is usually chosen between two and 4 ways to achieve low miss rates using only moderately complex L1 data access and management circuitry. Four ways are considered appropriate for the 32 KByte L1D described in the following sections. The remaining parameters for the basic cache configuration are equivalent to those described in Section 1.2; namely, a 32-bit address space, 4 KByte pages and a 4-way set-associative cache holding 32 KByte of data in 64 Byte wide lines. Fig. 4.1c illustrates the dimensions of the resulting memory address bit-fields. A sensitivity analysis concerning the influence of key parameters on Page-Based Memory Access Grouping and Page-Based Way Determination can be found in Sections 4.5 and 5.4.4, respectively.

4.3 The Multiple Access Low Energy Cache Interface

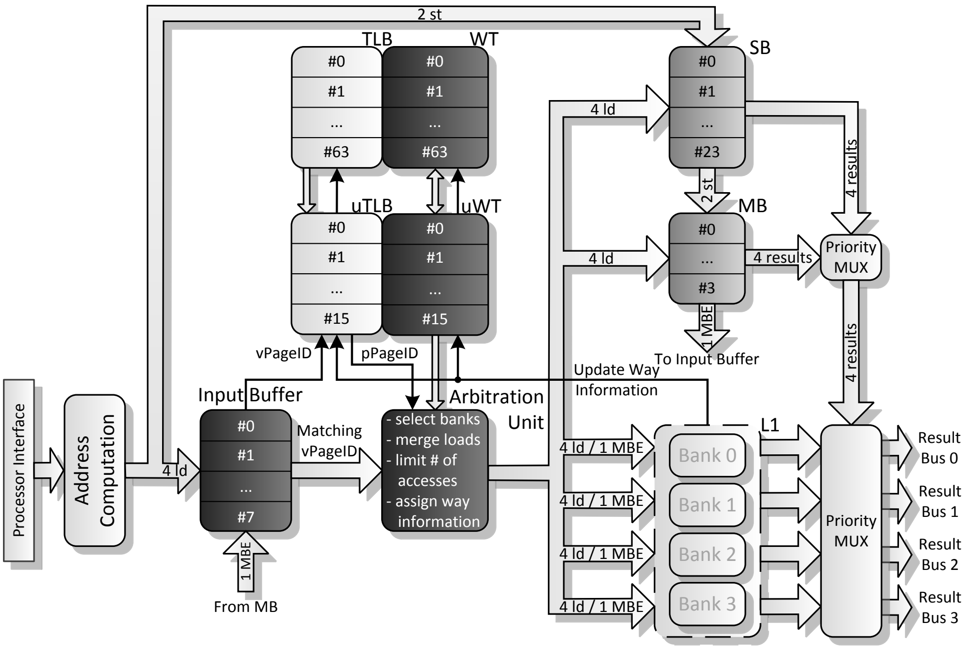


Fig. 4.2: The multiple accesses low energy cache interface (MALEC) based on Fig. 1.4, parameterized to service up to four loads and two stores in parallel

Fig. 4.2 gives a high level representation of MALEC's key elements based on to the processor-L1D interface described in the previous section (Fig. 4.1a). The newly introduced (black) and modified (dark grey) components will be discussed below. In order to demonstrate MALEC's scalability in terms of parallel memory accesses per cycle, the shown implementation is parameterized to service up to four loads and two stores in parallel. The underlying concept is to service multiple instructions in parallel by employing conventional techniques like cache banking and merge buffers. In addition, small architectural changes are introduced to facilitate Page-Based Memory Access Grouping, as suggested in Section 3.6. A key feature of the MALEC configuration presented here is its deliberate restriction to access only 1 page per cycle. This allows a number of micro-architectural simplifications to increase its energy efficiency for the cost of small performance degradation. While, Section 4.4 presents simulation results supporting this restriction, Section 4.5 discusses the effects of allowing accesses to two or more pages per cycle.

4.3.1 General Operation

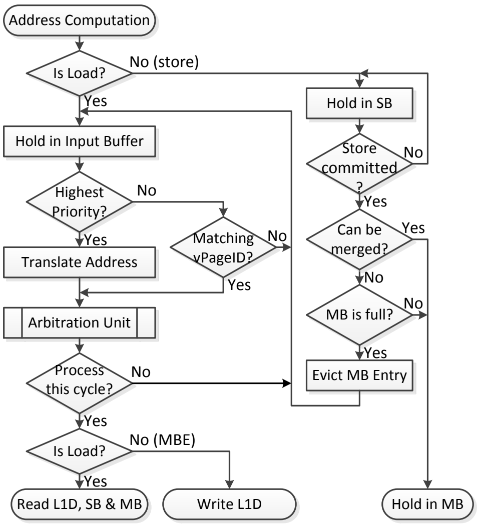


Fig. 4.3: Flowchart depicting the handling of loads and stores based on Fig. 4.2

To aid the understanding of Fig. 4.2, a simplified flowchart regarding the handling of loads and stores is depicted in Fig. 4.3. Stores finishing address computation are directly sent to the SB and remain there until they commit into the MB. Evicted MB entries (MBEs) and loads finishing address computation are forwarded to the Input Buffer. Each cycle, this buffer identifies a group of elements (loads / MBEs) sharing one virtual page ID (vPageID, Fig. 4.1c) and performs an address translation. All elements of the group are sent to the Arbitration Unit, where up to four loads and one MBE are selected to be serviced within the current cycle. The selected elements then access the L1D and in case of loads also the SB and MB. All elements not selected by the Arbitration Unit, as well as those not part of the current group, remain inside the Input Buffer until the next cycle.

4.3.2 Input Buffer

The implementation given in Fig. 4.2 assumes an address computation unit capable of servicing up to four loads, or two loads & two stores per cycle. While stores finishing address computation are simply forwarded to the SB, loads are directed to the so called Input Buffer. The purpose of this structure is to identify a group of memory instructions accessing the same page, which might then be serviced within the current cycle. It effectively reduces the number of required address translations and allows the simplification of subsequent structures (i.e. the Arbitration Unit (Section 4.3.3) and the SB&MB (Section 4.3.4)).

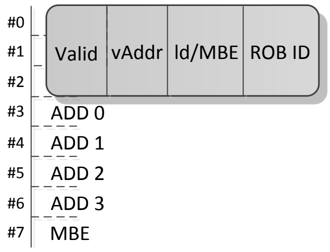


Fig. 4.4: Example configuration: input buffer

Fig. 4.4 gives an example of a possible Input Buffer implementation designed with the MALEC configuration of Fig. 4.2 in mind. It is composed of three storage elements holding information on loads that could not be serviced in pervious cycles, four elements representing loads that just finished address computation, and one element representing a merge buffer entry (MBE) to be evicted. Each entry is characterized by one or more status bits (e.g. validity), a virtual address (vAddr), its type (load or MBE) and in case of loads with some form of age information (e.g. a re-order buffer ID). The priority among entries is - from high to low: old loads, new loads and evicted MBEs. The reason for the low priority of evicted MBEs is the fact that stores represented by them are already committed and therefore no longer time critical.

At the start of each cycle, the vPageID of the highest priority Input Buffer entry is passed to the uTLB for address translation. Simultaneously, this vPageID is compared against all remaining, currently valid buffer entries, and all matching entries are then passed to the Arbitration Unit (Section 4.3.3). Finally, unmatched loads (i.e. vPageID varies from highest priority Input Buffer entry) and those loads rejected by the Arbitration Unit are stored until the next cycle. Should the three storage elements of the eight element Input Buffer in Fig. 4.2 be insufficient to hold all remaining loads, one or more address computation units are stalled. Note that the implicit re-ordering of elements by servicing them in groups based on pPageIDs does not introduce additional data hazards (i.e. RAW, WAR or WAW). Pre-existing hazards are identified similarly to the basic processor-L1D interface in Section 4.2 by performing SB and MB lookups. Note that alternative designs may also perform repeated load-store queue (LSQ) lookups subsequent to address computations.

4.3.3 Arbitration Unit

Fig. 4.5 illustrates a possible implementation of an Arbitration Unit designed for the MALEC configuration of Fig. 4.2. Being supplied with loads and MBEs from the Input Buffer, it selects up to four non-conflicting loads and one MBE to be serviced within the current cycle. It starts by determining on a per cache bank basis the access with the highest priority given by its order within Input Buffer. In case of a load, the unit then attempts to identify other loads to the same cache line by performing partial address comparisons for up to three references in consecutive Input Buffer positions relative to the initial, high

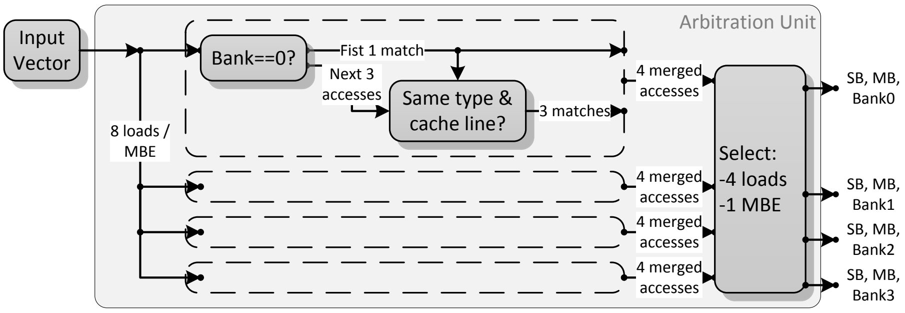


Fig. 4.5: Example configuration: arbitration unit

priority reference. There are two reasons for this behavior. First, address comparisons related to all banks are performed in parallel to minimize the overall delay. Allowing up to six candidate loads per comparison would require a significantly higher energy. Second, Tab. 3.2 in Section 3.7.4 indicates that read accesses to the same cache line tend to occur consecutively[1]. Next, while the Input Buffer of the configuration in Fig. 4.2 may identify up to seven loads and one MBE with a matching vPageID, the system only has four result buses. Consequently, the Arbitration Unit limits the number of loads to be serviced to the number of available result buses by selecting the four highest priority loads. An alternative approach would be to determine the combination of loads that requires the least number of cache accesses. However, the increased complexity of such circuitry would impose higher energy consumption and computation time per cycle. Similar considerations regarding the complexity of the involved routing network support the limitation to a maximum of four instead of seven result busses.

$$comparator_{bit} = address\_space_{bit} - PageID_{bit} - line\_offset_{bit} \tag{4.1}$$

$$comparator\_widened_{bit} = comparator_{bit} + number\_of\_sub - banks_{bit} \tag{4.2}$$

Considering that all loads/MBEs handled by the Arbitration Unit are guaranteed to share one PageID, its comparators are relatively narrow (Eqn. 4.1). A special case are sub-banked caches that attempt to save energy by splitting data arrays in smaller independent banks (usually 128 bit wide). To increase the probability for loads to be able to share data read from the cache, MALEC expects those caches to return data from two adjacent sub-banks on every read access, instead of only on those accesses that exceed one sub-bank; i.e. unaligned accesses. In consequence, the Arbitration Unit's comparators need to be widened according to Eqn. 4.2. For the example given in Fig. 4.5 and the MALEC configuration specified in Section 4.3 the components of the Arbitration Unit result to:

- One 2-bit priority comparators to identify 1 out of 8 Input Buffer elements (per bank)
- Three 8-bit comparators to identify up to 3 loads (consecutive to initial load, per bank)
- One single-bit priority selection circuit to identify up to four loads and one MBE

[1] Simulations based on the environment described in Section 4.4.1 reveal a performance difference of less than 0.5% between an unrestricted system and a system limited to only three candidate loads for its comparisons. This minor advantage is even further reduced due to latencies introduced by the additional comparator logic.

4.3.4 SB, MB and L1

Memory accesses selected by the Arbitration Unit are sent to the L1D and in case of loads also to the SB and MB. The cache itself is unmodified to allow the re-use of existing, highly optimized designs. An exception are sub-banked caches that are expected to return data from two adjacent sub-banks on every read access, instead of only on those that exceed one sub-bank (Section 4.3.3). While this effectively doubles the probability for loads to be able to share results read from cache, it slightly increases the energy consumed by the Arbitration Unit, because its comparators need to evaluate those address bits describing the desired sub-bank, too.

To service up to four loads per cycle, the SB and MB lookup structures need to be extended wit additional ports. However, MALEC's restriction to access only one page per cycle allows the simplification of said lookup structures in order to reduce their energy impact. For this purpose, the structures are split into two segments. One for address bits corresponding to the vPageID shared among all four loads, and the other for the remaining bits that identify access specific address regions (i.e. cache lines & sub-banks). For a system using 32-bit addresses and 4 KByte pages, this saves three 20-bit comparisons. Furthermore, as both segments can be looked up simultaneously, the MB and SB energy requirements are reduced without introducing additional latencies. The decision to refer store address translations until after merging is based on the desire to reduce the pressure on address translation units (i.e. uTLB, TLB and page tables) and save energy by avoiding multiple translations for stores accessing the same address region. The performance penalty of this approach due to additional processor stalls is insignificant, because stores commit to the MB instead of the L1. However, this approach requires the maintenance of additional information inside SB and MB to map virtual addresses to specific applications. While a MALEC implementation that performs address translations prior to SB allocations is possible, the higher number of address translations and the complexity introduced for page-based way determination are undesirable. Note that as stores bypass the Input Buffer when accessing the SB, they are not restricted to one page; hence, the corresponding SB write ports (two for Fig. 4.2) do not benefit from Page-Based Memory Access Grouping.

4.3.5 WT, uWT and other Components

The Way Table (WT) and micro Way Table (uWT) are not essential for the operation of MALEC and will be explained as part of the way determination scheme in Chapter 5. Furthermore, components that are not specifically altered by MALEC (light gray within Fig. 4.2) resemble those in Fig. 4.1a. However, they are assumed to support up to four parallel memory accesses. A naive approach to achieve this capability simply increases the number of ports of the corresponding queues and replicates address computation and priority multiplexing circuitry.

4.4 Evaluation

4.4.1 Methodology

In order to evaluate the impact of MALEC on performance and energy consumption and to compare it to existing microarchitectures, the gem5 Simulator System (Section 3.2) was extended to support an enhanced processor-cache interface capable of modeling the micro-architectural aspects involved in this study with cycle-level accuracy. Access statistics obtained from gem5 are combined with energy estimates calculated using CACTI v.6.5 [85] to determine the energy consumption of the L1D subsystem, including both static and dynamic components. The evaluation includes energy contribution of the following structures: L1D (tag&data SRAM arrays and control logic), uTLB and TLB. While the modeled L1D interface includes other structures - such as LQ, SB, and MB - their contribution to the overall energy consumption is not taken into account for two reasons. First, L1 and TLB account for the majority of transistors of the L1 interface, and therefore its leakage power. Second, the energy contributed by other components like LQ, SB and MB is very similar between MALEC and the analyzed baselines. Our simulations show that this is also the case for lower memory levels, i.e. L2 cache and main memory, as Page-Based Memory Access Grouping alters the timing of L2 accesses, but does not significantly impact their number or miss rate. The results generated by CACTI were verified using ARM internally available data from simulations and actual measurements of existing L1, TLB and uTLB structures.

The energy contributions of the newly introduced components, namely the Input Buffer and the Arbitration Unit, are considered negligible. In particular, the Input Buffer described in Section 4.3.2 consists of a set of comparators for eight 20-bit elements plus storage structures to facilitate up to three loads. This makes it effectively smaller than the uTLB, which contributes to only 0.3% and 2.1% of the overall leakage and dynamic energy consumption, respectively. As the MALEC implementation in the following sections is scaled down to allow fair comparisons to the analyzed baselines, the influence of its Input Buffer (five comparators and two storage elements) and the even smaller Arbitration Unit is further decreased. Moreover, the simplified lookup structures for SB and MB (Section 4.3.4) reduce the energy consumption of those components, and therefore offset additional energy introduced by other MALEC components.

Component	Parameter
Processor	single-core, out-of-order, 1 GHz clock, 168 ROB entries, 6 elem. fetch/decode/rename and dispatch, 8 elem. issue
L1 interface	64 TLB entries, 16 uTLB entries, 40 LQ entries, 24 SB entries, 4 MB entries, 32-bit address space, 4 KByte pages
L1D	32 KByte, 2 cycle latency, 64 byte lines, 4-way set-assoc., 128-bit sub-blocks per line, 4 banks, physically indexed, physically tagged, 6 MSHRs (8 targets each)
L2 cache	1 MByte, 12 cycle latency, 16-way set-assoc.
DRAM	512 MByte, 30 cycle latency
CACTI	32nm technology, design objective low dynamic power, cell type low standby power for data & tag arrays and high performance for peripherals, L1 with ECC

Tab. 4.1: Relevant simulation parameters

The configuration of the simulated system is based on an ARMv7-compatible single-core out-of-order processor operating at 1 GHz. Relevant configuration parameters of the analyzed processor-cache interface are summarized in Tab. 4.1 and justified in Section 4.5. The benchmark suites used in the following sections - i.e. Media Bench 2 (MB2) [83], SPEC CPU2000 Int and FP [82] - represent a set of workloads with a multitude of different memory access behaviors. In order to reduce simulation times, SimPoint v.3.1 [84] was used to identify the most representative execution phase of each benchmark. Each phase includes 1 billion instructions of the corresponding reference working set. Note that while Chapter 6 describes the development of a benchmark suite specifically designed to utilize advanced SIMD instructions and features such as per-lane predication, indexed memory accesses, scans and segmented scans, the results presented here evaluate MALEC for scalar workloads only. Hence, they are not capable of fully utilizing the previously presented MALEC configuration.

	Address Computations per Cycle	uTLB/TLB ports	Cache Ports per Bank
Base1ldst	1 ld/st	1 rd/wt	1 rd/wt
Base2ld1st	2 ld + 1 st	1 rd/wt + 2 rd	1 rd/wt + 1 rd
MALEC	1 ld + 2 ld/st	1 rd/wt	1 rd/wt

Tab. 4.2: Analyzed configurations - basic parameters

Tab. 4.2 characterizes the analyzed baselines and the chosen MALEC configuration in terms of potential address computations per cycle (ld.. load, st.. store, ld/st.. load or store), as well as the number and type of uTLB, TLB and cache ports (rd.. read, wt.. write, rd/wt.. read or write). While *Base1ldst* is restricted to a single load or store per cycle, *Base2ld1st* represents a high performance configuration allowing up to two loads and one store in parallel. The underlying processor configuration is optimized for *Base2ld1st*, and as previously mentioned not capable of fully utilizing the MALEC configuration introduced in Section 4.3. Hence, the MALEC configuration analyzed here is scaled down to service a maximum of three loads, or two loads & one store in parallel (Tab. 4.2). To allow fair comparisons, it operates on the same number of LQ, SB and MB ports as well as address computation units as *Base2ld1st*.

4.4.2 Performance

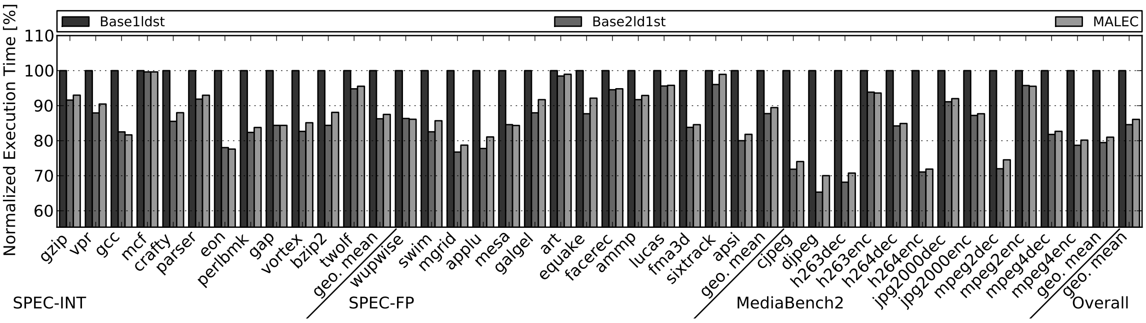


Fig. 4.6: Normalized performance of *Base1ldst*, *Base2ld1st* and MALEC based on Tab. 4.2

Fig. 4.6 illustrates the performance of *Base2ld1st* and MALEC relative to *Base1ldst* in terms of CPU cycles required per benchmark. Based on the geometric mean over all analyzed benchmarks ('Overall' entry in Fig. 4.6), it can be observed that the analyzed MALEC configuration achieves a performance improvement of 14% over *Base1ldst*, which is only 1% less than *Base2ld1st*. Consequently, it achieves performance similar to *Base2ld1st*, without the need for a physically multi-ported uTLB, TLB or L1D. Considering Fig. 3.9a, which indicates that only about 70% of all loads within the analyzed benchmarks are within groups of two or more consecutive accesses to the same page, an overall performance difference of just 1% may appear unexpectedly small. However, MALEC's capability to reorder accesses queued up within its Input Buffer effectively allows it to accelerate certain loads while delaying others; i.e. ld A, ld B, ld A might be processed as ld, A ld A followed by ld B (A and B representing cache lines). Furthermore, as the underlying processor is optimized for *Base2ld1st*, the actual number of loads and stores issued in parallel is rather low (around 1.6 in average). Hence, the approximately 60% of loads within bzip and equake that are not directly succeeded by any other loads to the same page (Fig. 3.9a) may actually be issued in separate cycles, resulting in only moderate performance penalties of 5% and 6% for those benchmarks.

Comparing SPEC-Int, SPEC-FP and MB2 averages reveals performance improvements of 14%, 12% and 21%. One reason for the increased benefits of SPEC-Int over SPEC-FP is the higher ratio of memory accesses to computation instructions within these benchmarks; i.e. Fig. 3.1 shows ratios of approximately 45% and 40% for SPEC-Int and SPEC-FP, respectively (Section 3.3). MB2 exhibits a lower ratio of memory accesses to computation instructions as SPEC-Int, but its media kernels rely on frequent, highly structured memory accesses. Consequently, MB2 benefits more from page-based access grouping and sharing of data among loads to the same cache line. Exceptionally low improvements over *Base1ldst* are shown by mcf and art. Reasons for this are large working sets combined with low memory access localities leading to high miss rates that do not benefit from faster L1 accesses. In contrast, djpeg and h263dec exhibit excellent access localities and tend to execute a high number of memory accesses in parallel, resulting in speedups of approximately 30%.

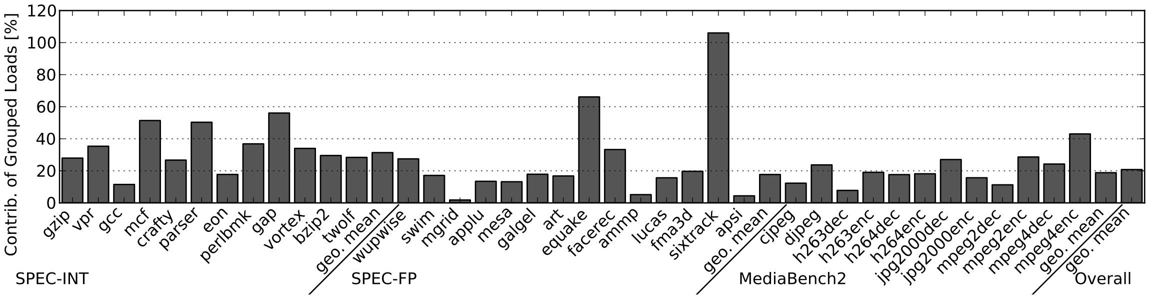


Fig. 4.7: Contribution of grouped loads to the performance improvement of MALEC over *Base1ldst*

The performance benefits provided by MALEC over *Base1ldst* in Fig. 4.6, primarily originate from two mechanisms: grouping of loads to the same cache line and accessing multiple cache banks in parallel. Both mechanisms require multiple address translations per cycle, and therefore take advantage of the ability to share address translation results among accesses to the same page. To demonstrate the contribution of grouped loads to the performance improvement of MALEC over *Base1ldst*, Fig. 4.7 compares the data previously presented in Fig. 4.6 to a MALEC configuration without the ability to group loads. Based on the number of executed CPU cycles (C), the contribution is determined by $\frac{C_{\text{MALEC\_no\_load\_grouping}} - C_{\text{MALEC}}}{C_{Base1ldst} - C_{\text{MALEC}}}$. On average, the grouping of loads amounts to approximately 21% of MALEC's overall performance improvement. The benchmarks gap and equake achieve significantly higher percentages of 56% and 66%, due to particularly suitable memory access patterns; i.e. groups of loads to the same cache line executed within a short period of time. In contrast, mgrid shows a value of less than 2%, implying loads with a relatively low spatial (intra cache line) or temporal locality. The exceptionally high value of 106% for sixtrack is an artifact based on the fact that the MALEC configuration without the ability to group loads is slower than *Base1ldst*. An unexpected result may be the higher contribution of grouped loads to SPEC-INT (30%) over MB2 (20%). It can be concluded that MB2 exhibits a high number of structured memory accesses that particularly benefit from cache banking rather than load grouping; e.g. loading every n_th element of an array, or loading all variables A of an array of structs composed of variables A and B. In summary, an overall performance difference of just 1% between MALEC and *Base2ld1st* in Fig. 4.6 confirms the observations made in Section 3.6, which imply that it is sufficient to handle only those instructions in one cycle that access the same page.

4.4.3 Energy

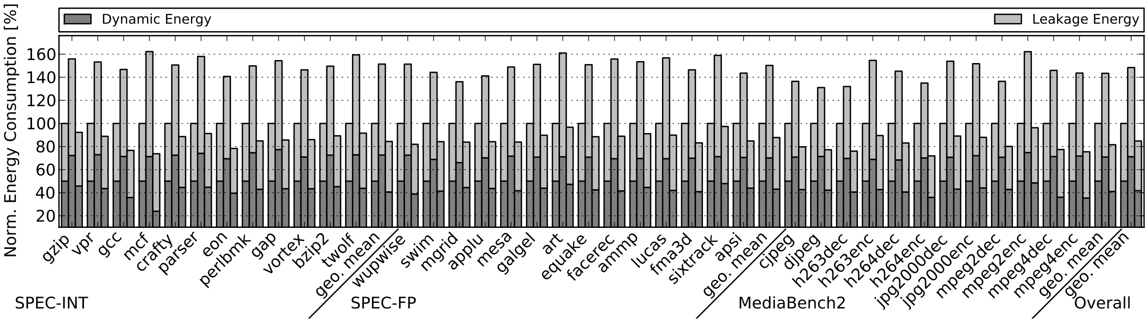


Fig. 4.8: Normalized energy consumption of *Base1ldst* (left), *Base2ld1st* (centre) and MALEC (right) based on Tab. 4.2

Similar to the performance analysis in Fig. 4.6, the dynamic and overall energy consumption of *Base2ld1st* and MALEC relative to *Base1ldst* is illustrated in Fig. 4.8. Considering the geometric mean over all analyzed benchmarks, *Base2ld1st* leads to an increase in dynamic energy consumption of 42%. The primary cause of this are the additional physical ports of its uTLB, TLB and L1D (Tab. 4.2), required to achieve its high performance operation. In contrast, MALEC saves 23% of dynamic energy compared to *Base1ldst*, by utilizing Page-Based Memory Access Grouping to share address translation results and data read from L1 among multiple accesses. The unusually high savings of MALEC for mcf originate in the exceptionally high miss rate of the benchmark (about 7 times the overall average, Fig. A.2). As MALEC attempts to share L1 data among loads addressing the same cache line, the effective number of loads accessing and missing the cache is reduced.

As leakage contributes to about 50% of the overall energy consumption in the analyzed 32nm technology library, it is important to account for it. Fig. 4.8 reveals that *Base2ld1st*'s average energy consumption actually lays 48% above *Base1ldst*. Reason for this is the leakage power introduced by its additional uTLB, TLB and L1 ports, which outweighs savings due to reduced computation times; for example: the additional read port increases L1 leakage power by approx. 80%, but the average computation time is only reduced by 15% (Section 4.4.2). In contrast, MALEC exhibits the same number of uTLB, TLB and L1 ports as *Base1ldst*. As mentioned in Section 4.4.1, its additional components (i.e. Input Buffer and Arbitration Unit) have an even smaller transistor count than the uTLB. Hence, their contribution to the overall leakage is negligible. In consequence, MALEC's energy saving remains 15% relative to *Base1ldst*, but increases to 43% relative to *Base2ld1st*. Note that the values presented here are primarily intended for the comparison of MALEC and *Base2ld1st*. Fair comparison to *Base1ldst* would require the consideration of energy contributions originating from other processor internal circuitry (queues, datapaths, ect.), adapted to support the increased number of parallel memory accesses.

4.5 Sensitivity Analysis

Suitability for SMT and CMPs

The majority of parameters specified in Tab. 4.1 are typical for modern high performance microarchitectures such as Intel's Ivy Bridge [5]. One exception is the number of simulated cores. While MALEC might be implemented on any number of cores, the proposal described in Section 4.3 does not explicitly target synchronization, memory sharing or other key aspects of Chip Multiprocessors (CMPs). In this sense, CMPs are considered an orthogonal aspect to the presented research. Furthermore, the analyzed benchmark suites SPEC2000 and MediaBench2 are single-threaded and would not benefit from a CMP environment. Suites like PARSEC that specifically target multi-threaded systems, are primarily focused on scientific rather than general purpose applications. Hence, they are outside the scope of the proposed MALEC configuration, which was designed in the context of energy constrained general purpose processing.

Simultaneous multithreading (SMT) - i.e. the parallel execution of two or more threads within one core - is considered mostly orthogonal to MALEC, too. A simple SMT implementation might introduce additional address bits to associate memory references with specific threads. Beyond that, a designer may decide to either use the basic MALEC implementation described in the previous sections, or an extended version optimized for SMT. The latter may split certain MALEC components (i.e. Input Buffer, Arbitration Unit and SB&MB page lookup mechanisms) into two or more, smaller, independent versions, and provide additional access ports for the uTLB and TLB. This would allow accesses to two pages per cycle, originating from different threads. Similarly to the basic cache interface

(Section 4.2), cross-thread memory dependencies would need to be handled by corresponding hardware and/or software mechanisms. Note that the partial replication of MALEC components to allow accesses to two or more pages per cycle might also be considered for systems that require high performance implementations for frequent mem-copy operations; i.e. load data from one memory location and directly store it at another.

Latency Overhead

The analyzed frequency of 1 GHz was chosen with respect to power restricted mobile processors. Desktop or server implementations might operate on higher frequencies, leading to reduced computation times. However, the benefits of shorter computation times for the overall leakage energy would be mitigated and potentially outweighed by the increased leakage of faster transistor technologies. In consequence, MALEC's energy consumption relative to *Base1ldst* and *Base2ld1st* would be widely unaffected.

In the context of higher frequencies, additional latencies introduced by MALEC gain relevance. To reduce their impact, the latencies of the Input Buffer (vPageID comparisons) and Arbitration Unit (partial address comparisons) are hidden behind uTLB/TLB lookups. This requires serialized address translation and data access, as described in Section 4.2. In contrast, cache interfaces that perform address translation and data access in parallel (i.e. VIPT or certain PIPT caches) would need to implement Page-Based Memory Access Grouping prior to TLB and L1 lookups, and therefore potentially require an additional cycle per memory access. The impact of additional cycles on MALEC's performance and energy consumption is further analyzed in Section 5.4.

Depending on the spatial locality within a specific application, loads may reside inside the Input Buffer for several cycles. Consequently, MALEC increases the variability in load latency. The worst case scenario for the implementation in Section 4.3 describes eight loads issued within two consecutive cycles, which access different pages and/or cannot be merged by the arbitration unit. Hence, a load might be delayed by up to six cycles. While design specifics for issue queues are outside the scope of this project, a conservative implementation might introduce an additional status bit for loads within the IQ. This bit might be set as soon as it is known that a particular load will be serviced in the next cycle (e.g. by the Arbitration Unit). Hence, load dependent instructions may have to wait for this bit to be set before they can issue (this policy is used for the simulations in Section 4.4). Alternatively, very aggressive systems may issue all load dependent instructions speculatively and trigger replays similarly to load misses if necessary. A middle way might issue load dependent instructions based on a speculative latency (i.e. assume that all loads will be serviced within n cycles). Results obtained earlier than expected will be buffered until needed; results obtained too late will require replays. This approach would potentially reduce MALECs performance benefits, but still save energy due to more efficient (merged) TLB, SB, MB and L1D accesses.

Scalability

A key feature of MALEC is its scalability in terms of parallel L1D accesses. In fact, its efficiency is proportional to the number of memory request simultaneously issued by the processor. The reason for this is the increasing probability of being able to accumulate multiple accesses to the same page or even cache line within the Input Buffer. Consequently, more accesses may share address translations and - in case of loads to the same cache line - data read from the L1D. Although the parameters specified in Tab. 4.1 (Section 4.4.1) are typical for a modern high performance microarchitecture, the actual number of parallel issued memory instructions for the analyzed benchmarks is rather low (around 1.6 in average). The compilation of MB2 on gcc with enabled NEON support does not visibly increase this number. One reason for this is that less than 3% of all instructions committed by MB2 benchmarks are actually vectorized. While highly optimized (handwritten) NEON benchmarks are available, the NEON ISA itself is considered insufficient to fully utilize MALEC, too. Besides the limited datapath width of 128-bit, the primary concern regarding NEON is its limitation to basic vector memory access patters such as unit-stride and specific non-unit strides (distance 1, 2, 3 or 4). This severely restricts the amount of vectorizable general purpose code. Moreover, NEON's ability to merge unit stride accesses of small data types (e.g. 8/16-bit) into wider accesses (e.g. 64-bit) effectively reduces the number of L1 accesses visible to MALEC. Note that Chapter 6 describes the development and evaluation of an extension to NEON that does support advanced SIMD features including indexed memory access, non-unit strides over arbitrary distances, and wider datapaths.

Cache/Memory Parameter Dependencies

Of particular importance for MALEC are cache specific parameters such as capacity, number of banks and sub-banks, and address space width. First, Page-Based Memory Access Grouping does not directly depend on the cache capacity. However, smaller caches usually exhibit higher miss rates and therefore increase the number of re-issued memory access requests leading to a better utilization of MALEC. Second, the probability of loads to the same cache line being mergeable is proportional to the number of sub-banks. In particular, as conventional L1 caches only activate two adjacent sub-banks per read access, the number of mergeable loads increases with smaller cache lines or wider sub-blocks. Third, a higher number of cache banks potentially increases MALEC's performance benefits, while decreasing its energy efficiency due to a more complex Arbitration Unit. Finally, the energy needed for address comparisons within the Input Buffer for systems supporting 40- or 48-bit address spaces or wider pages (> 4 KByte) is outweighed by higher savings due to shared address translation results and simplified SB and MB lookups.

Exception Handling

A major design concern for out-of-order processors is the handling of precise exceptions, i.e. restoring the architectural state corresponding to an in-order processor interrupted at the same instruction (Section 1.1). The amount of speculative state held by MALEC itself is limited to loads within the Input Buffer and the Arbitration Unit. All other information, e.g. in form of SB entries or MB entries, is the same as without MALEC or non-speculative and therefore does not impact any recovery mechanism.

4.6 Concluding Remarks

This chapter introduced an energy efficient L1D interface designed for high performance out-of-order superscalar processors. The proposed Multiple Access Low Energy Cache (MALEC) is based on the observation that consecutive memory references are very likely to access the same page. It shares memory address translation results between multiple loads and stores, simplifies store and merge buffer lookup structures and shares L1 data among loads accessing the same cache line. The design was evaluated based on simulations of a 32nm implementation employing a 32 KByte, 4-way set-associative L1D with 64 Byte wide lines and an aggressive out-of-order processor to execute SPEC2000 and MB2 benchmarks. Compared to a basic cache interface, capable of servicing one load or store per cycle, the chosen MALEC configuration achieved 14% speedup using 15% less energy. In contrast, a conventional interface that achieved only a slightly higher speedup of 15% consumed 48% more energy with respect to the baseline.

The engineering contributions underlying this chapter are described in Section 4.4.1. To facilitate MALEC, the gem5 cache model was extended to support banking, sub-banking, ports (read, write and read/write) and enforce the corresponding contentions. Furthermore, the existing SB model was improved and additional components (MB, uTLB, Input Buffer and Arbitration Unit) introduced. The energy estimates obtained from CACTI were verified against ARM internally available data from RTL simulations and actual measurements.

5 | Page-Based Way Determination

5.1 Motivation

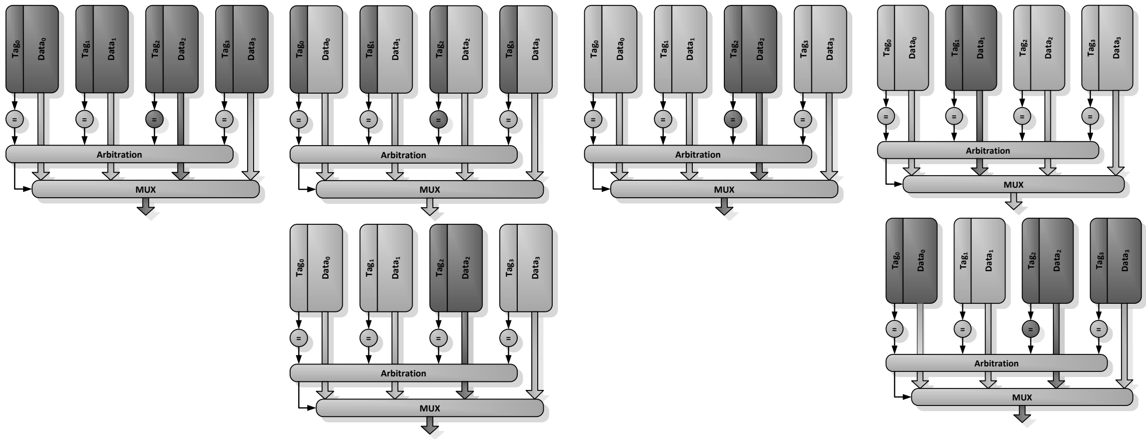
The previous chapter described Page-Based Memory Access Grouping in the context of a Multiple Access Low Energy Cache interface. This chapter expands on MALEC by proposing a novel way determination scheme (Section 2.1.3) specifically designed to service multiple parallel memory accesses in an energy efficient manner. Exploiting MALEC's restriction to access only one page per cycle, it is highly scalable in terms of parallel memory accesses, while imposing a minor performance penalty due to additional latencies. The key contributions of Page-Based Way Determination are:

- Re-use of uTLB/TLB lookups to avoid address comparisons during way determination (uWT/WT lookups)
- Processing of way information at page granularity (determine ways for all lines within a given page at once)
- Use of validity information (100% accuracy) to allow memory references to completely avoid tag comparisons and directly access desired cache lines (possible due to use of physical addresses as prediction source; i.e. avoids aliasing issues of virtual addresses)
- An efficient storage format combining way and validity information (Section 5.3.3)

The following sections describe how way prediction schemes can be applied to achieve energy efficiency and introduce the concept of Page-Based Way Determination (Sections 5.2 and 5.3). Section 5.4 then provides performance and energy analyses, followed by a sensitivity analysis in Section 5.4.4. The concept of miss-predictions - i.e. the prediction of L1D missed - as it can be used to completely avoid tag-array accesses for all processor internal memory references, is introduced and discussed in Section 5.5 before the chapter concludes in Section 5.6.

5.2 Reducing Set-Associative Cache Energy via Way-Pred.

Each of the four independent cache banks underlying the MALEC implementation presented in the previous chapter is 4-way set-associative. While set-associative structures usually exhibit lower miss-rates than equivalently sized direct-mapped structures, they introduce an additional level of complexity; i.e. the need to determine in which way a particular datum is located (Fig. 1.6 in Section 1.2). Fig. 5.1a illustrates the naive approach to locate and access data within such a set-associative cache. First, all available tag and data arrays are accessed in parallel; second, the matching tag is identified, and finally the desired data is selected. Due to its simplicity and high performance, this scheme is commonly used in L1Ds. The primary drawback of this scheme is the energy consumed for the activation of the whole cache on each access. The approach depicted in Fig. 5.1b avoids redundant data-array accesses by probing tag- and data-arrays sequentially. However, as



(a) Parallel (b) Sequential (c) Prediction - Correct (d) Prediction - False

Fig. 5.1: Set-associative cache access schemes based on Fig. 1.6 (Section 1.2)

this introduces an additional memory cycle, it is undesirable for performance sensitive systems. The energy estimates underlying the previous chapter are based on a combination of Fig. 5.1a and 5.1b. In particular, while all tag- and data-arrays are activated in parallel, broadcasts within data-array internal h-tree structures are delayed until the confirmation from the corresponding tag-array is received. Fig. 5.1c and (d) illustrate the idea of using way prediction to realize more energy efficient cache accesses. Correct predictions only require one tag- and data-array to be accessed. However, false predictions entail a second cycle and the activation of all remaining tag- and data-arrays. Consequently it is desirable to either minimize the number of false predictions or avoid them completely.

The way prediction schemes introduced in Section 2.1.3 were primarily designed with single access caches in mind and are not necessarily suitable for MALEC. In particular, while "way prediction" techniques based on MRU statistics are simple to implement, false predictions require a second cache access to find the desired datum within the previously discarded ways. This not only increases the latency of the falsely predicted accesses, but also reduces the number of cache banks available to subsequent accesses within the next cycle. Depending on the actual MALEC implementation, these subsequent accesses might furthermore be limited to the same vPageID as used by the falsely predicted accesses. Significantly better suited for MALEC are "way estimation" techniques. As they do not require subsequent cache accesses to recover from false predictions, their performance impact is limited to additional comparator latencies within the critical path. However, operations based on a set of potential ways still consume energy for unnecessarily activated tag- and data-arrays. The Way Determination Unit (WDU) proposed by Nicolaescu et al. avoids this problem by associating each cache line with exactly one way and guaranteeing it to be either found there or not to be present in the cache at all. Primary concern for the adaption of this technique for MALEC is its scalability. To achieve n parallel WDU predictions per cycle, n WDU ports are required. As the WDU uses fully-associative tag lookups, these ports impose a significant energy and latency overhead. To reduce the number of required ports, WDU accesses may be performed subsequent to address comparison within the Arbitration

Unit. Hence, n would be limited by the maximum number of parallel accessible cache banks instead of the number of elements inside the Input Buffer. A second drawback of WDUs is their limited coverage. As MALEC already implements a merge buffer and the Arbitration Unit to exploit intra cache line locality, the effective number of accesses covered by the WDU is significantly reduced. The Page-Based Way Determination scheme proposes in the following sections was designed with the following parameters in mind:

- Avoid subsequent cache accesses as a consequence of false predictions
- Avoid unnecessary tag- and data-array activations (predict exactly one way)
- Avoid tag-array accesses to verify prediction results
- High scalability in terms of parallel accesses per cycle
- High coverage even for limited memory access locality (temporal and spatial)
- Limited latency introduced into the critical path

5.3 Proposed Configuration

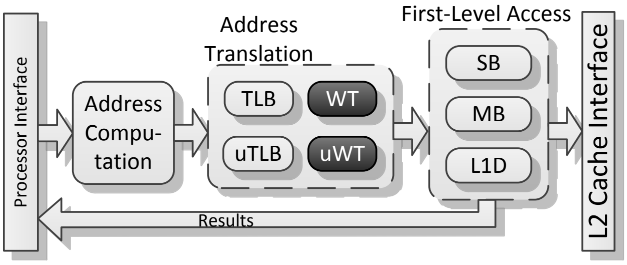


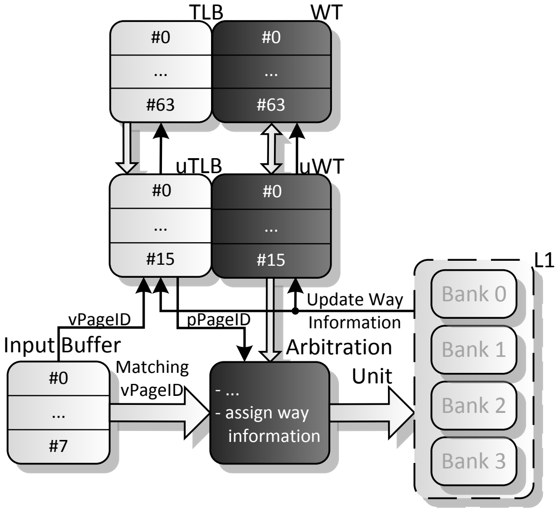
Fig. 5.2: Extended processor-L1D interface (Fig. 4.1a)

The Page-Based Way Determination scheme introduced in this chapter extends the L1 cache interface underlying MALEC with so called Way Tables (Fig. 5.2). Each table corresponds to a specific TLB level; i.e. TLB - WT (Way Tables) and uTLB - uWT (Micro Way Table). While the actual number of way tables is an implementation dependent parameter, simulation results suggest the use of at least two levels. The following sections give detailed information on each WT level and their interactions. Further exploiting the idea of Page-Based Memory Access Grouping, each WT entry includes way information corresponding to all cache lines within a given page. Besides enabling the WT to simultaneously service all loads and stores accessing one particular page, this allows the use of a simplified lookup structure that re-uses address comparisons required for TLB lookups to reduce the WT's energy consumption and latency. The scheme uses validity information to achieve 100% accuracy, allowing the majority of cache accesses (94%, Section 5.4.3) to bypass the cache's tag-array completely. This distinguishes it from alternative schemes (previous section) that need to verify their results with at least one tag comparison. MALEC uses this feature to increase its energy efficiency by supporting two different cache access modes:

- Conventional cache access (way unknown):
  - Parallel access to all tag- and data-arrays
  - Select data associated with matching tag

- Reduced cache access (way known and valid):
  - Tag-arrays bypassed
  - Access to one specific data-array only

5.3.1 General Operation



- Input Buffer sends vPageID for address translation
- uTLB hit: return pPageID and index uWT
- uTLB miss: forward pPageID to TLB
- TLB hit: return pPageID and index WT
- TLB miss: forward pPageID to lower level TLB or page table
- Evaluate uWT/WT entry in Arbitration Unit
- Valid way information: perform reduced cache access
- Else: perform conventional cache access

Fig. 5.3: Extract of MALEC (Fig. 4.2) and general operation of way tables

Fig. 5.3 shows an extract of the MALEC implementation introduced in Section 4.3. Each cycle, the Input Buffer sends one vPageID to the uTLB to be translated. In case of a uTLB miss, the vPageID is forwarded to the TLB and if necessary to lower TLB levels or page tables (Section 1.1.2). While the parallel lookup of uTLB and TLB may provide higher performance, the following sections assume more energy efficient sequential lookups. In case of a uTLB/TLB hit, a pPageID is returned to the Arbitration Unit and the location of the corresponding uTLB/TLB entry is used to index a WT/uWT entry; e.g. should a vPageID hit the third uTLB entry, the third uWT entry is activated. Consequently, the energy per page-wide address lookup is split over both structures (20-bit for 4 KByte pages and a 32-bit address space). As this process requires address translations to be performed prior to L1 accesses, it is not suitable for performance optimized systems that parallelize TLB and L1 lookups (Section 4.2). Each WT entry contains way and validity information for all cache lines mapping to the page described by its corresponding TLB entry. The evaluation of WT entries received is performed by the Arbitration Unit (Section 4.3.3). It assigns ways to groups of memory references (loads or MBEs) accessing the same cache line, and forwards this information to the corresponding cache banks. The maximum number of ways read from a single WT entry is thereby equal to the number of available cache banks (four for the system in Fig. 4.2). Consequently, the energy required to evaluate WT entries is independent of the number of memory references to be serviced in parallel, which makes page-based way determination highly scalable. Finally, the obtained way information is used to perform reduced or conventional cache accesses for valid or invalid ways, respectively.

5.3.2 Way Tables and Update Mechanisms

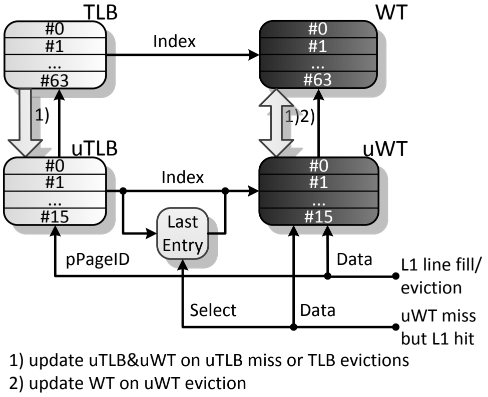


Fig. 5.4: Overview of interactions between TLBs, WTs and the L1D based on Fig. 5.3

Fig. 5.4 gives a more detailed overview of the interactions between TLBs, WTs and the L1D based on Fig. 5.3. Way tables are closely coupled to their respective address translation component, similar to a cache's tag- and data-array. The number of entries within a WT, and therefore the number of pages covered by it, equals the number of entries within the corresponding TLB (64 TLB and 16 uTLB entries in Fig. 5.3). Assuming 4 KByte pages and 64 Byte cache lines, each WT entry holds information on 4096/64 = 64 lines (Section 5.3.3). Similar to an uTLB, an uWT holds only a fraction of the corresponding WT entries, but is able to service the majority of memory accesses. In fact, the more accesses can be serviced by the small uWT instead of the WT, the more energy is saved compared to a conventional cache access.

A prerequisite for reduced cache accesses is the accuracy of way information. Consequently, validity bits corresponding to way information of specific lines are set and reset on cache line fills and evictions. To avoid energy consuming redundant WT accesses, a write-back policy is implemented. Hence, the WT is only updated if no corresponding uWT entry was found and the synchronization of uWT and WT itself is based on full entries transferred during uTLB updates. An alternative design might implement uWT and WT exclusive rather than inclusive (i.e. TLB/WT contain no uTLB/uWT data). However, this would incur additional latencies for transfers from the uWT to the WT, which are undesired in performance sensitive systems. To reduce the number of uWT to WT transfers, the uTLB might implement an advanced replacement policy. Considering the low number of uTLB entries (16, Fig. 5.4), the additional overhead imposed by most policies is modest compared to random replacement; the simulation environment in Section 5.4.1 therefore applies the second chance algorithm [10]. Because the cache performs line fills and evictions based on physical tags, the uTLB and TLB need to be modified to allow lookups based on physical, in addition to virtual PageIDs. Furthermore, the finite number of TLB entries (64, Fig. 5.4) might require the eviction of a page that still has corresponding lines within the cache. If the page is re-accessed later on, a new WT entry is allocated and all way information invalidated. Hence, invalid way information stored within a WT entry cannot be used to "predict" L1 misses, it simply indicates that the WT is unable to tell where the desired line is held. A way determination returning way unknown requires a conventional cache access, including a tag-array lookup and access to all available ways. Section 5.5 discusses the idea of a mechanism to temporarily store evicted WT entries, and therefore effectively eliminate the need for conventional cache access and in case of uniprocessor systems the tag-array itself.

To compensate for the loss of information due to page evictions, the uWT is updated subsequent to conventional cache accesses hitting the L1D; i.e. if the uWT returns way unknown, but a conventional cache access hits, the corresponding uWT entry is updated. The difficulty of this update mechanism is the delay between a uWT access and the feedback (hit/miss) from the L1 cache. Assuming a single-cycle L1 accesses latency, the feedback information would correspond to the preceding and not the current uWT access; hence, a uTLB lookup would be required to identify the uWT entry to be updated. To avoid this additional uTLB lookup, a register holding the ID of the last read uWT entry is introduced (Fig. 5.4). In consequence, feedback of conventional cache accesses hitting the cache can bypass the uTLB and simply accesses the uWT entry indicated by this register. Hence, the feedback information only includes a way ID and the line address to identify the corresponding bit field within the uWT entry (Section 5.3.3). In case of a multi-cycle L1 access latency, the register may be replaced by a FIFO. Depending on the chosen uTLB replacement policy this may require FIFO entries to be invalidated in consequence of corresponding uTLB/uWT entries being evicted. Such mechanism are unnecessary for the second chance replacement policy employed by the following evaluation, because it operates in a FIFO like manner and does not evict recently accessed uTLB entries. Simulations based on the environment of Section 5.4 show that the update of way information in consequence of conventional cache accesses instead of just on cache line fills and evictions allows page-based way determination to cover the vast majority of L1D accesses (94% instead of approximately 75%).

5.3.3 Composition of Way Table Entries

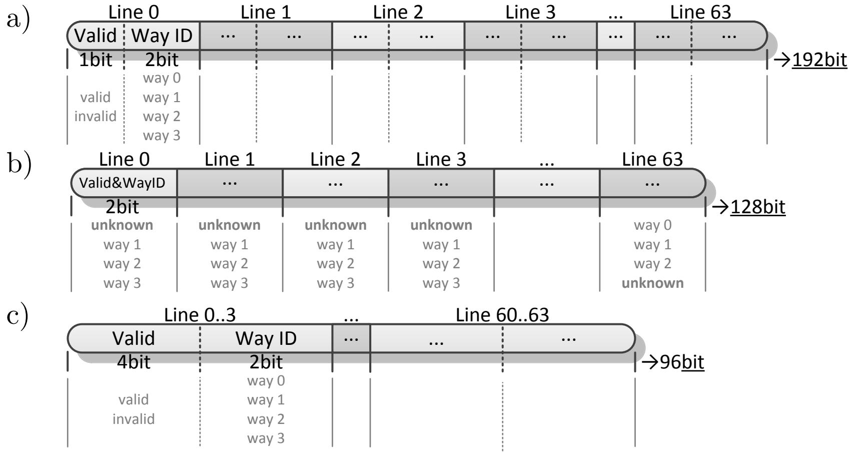


Fig. 5.5: Examples of possible way table entry implementations

How way and validity information is arranged within WT entries is not relevant for their function. Fig. 5.5 illustrates a selection of three possible arrangements for a system with 4 KByte pages, 64 Byte cache lines and a 4-way set-associative cache. For all three, the position of a particular bit field within the WT entry is associated with the corresponding line inside a page. This allows the use of simple decoders instead of complex lookup structures to index specific lines. The naive approach shown in Fig. 5.5a uses separate validity and way fields, resulting in an overall storage requirement of 192 bits per WT entry.

Considering that for the given system each line supports only five possible states (i.e. way0, way1, way2, way3 and unknown), the encoding of 8 states within 3 bits is inefficient. Fig. 5.5b combines way and validity information within 2 bits, resulting in an area and leakage power reduction of 1/3. Given the configuration of Fig. 4.2, a cache consisting of four banks may allocate lines 0..3 to separate banks and lines 0, 4, 8, .., 60 to the same bank. Deeming way 0 invalid for lines 0..3, way 1 invalid for lines 4..7 and so on limits the number of available ways for a particular cache line to 3, which allows 2 bits to encode way and validity information. Note that while a single line is limited to three cache ways, the working set of a specific application may still use all four ways of the 4-way set-associative cache. In fact, simulations based on the configuration described in Section 5.4 show no measurable increase of the L1 miss rate from Fig. 5.5a to Fig. 5.5b.

An even more restrictive approach is illustrated in Fig. 5.5c. It combines way information of four adjacent lines (e.g. lines 0..3 or 4..7) by directing the L1's replacement policy to place them within the same way. As these lines are located in different banks, the total number of available cache ways is unaffected. Although this still requires one validity bit per line, the size of a WT entry is reduced to 96 bit. However, this approach has several disadvantages and is therefore not further investigated in subsequent sections. First, it limits the freedom of the cache replacement logic, which can lead to increased miss rates. Second, each cache replacement requires a Way Table access to evaluate potential pre-existing way information (might be combined with usual WT updates on line fills/evictions). Third, in case of a page being evicted and later on reloaded into the TLB, the information within the corresponding Way Table entry is lost. Consequently, the cache replacement policy would allocate a new cache line to an arbitrary way. This way may differ from the way an older line that still resides in the cache is located. Hence, updates of Way Table entries with information on re-discovered cache lines would be more complicated. Note that without these updates the coverage of WTs, and therefore their potential energy savings, is reduced from 95% to 80%.

5.4 Evaluation

5.4.1 Methodology

The methodology underlying this chapter is based on the analysis of MALEC in Section 4.4.1. Both way tables implement 128-bit entries and are modeled in conjunction with their corresponding TLB; i.e. uTLB and TLB are treated as tag-arrays for their respective uWT and WT data-array (Fig. 5.5b). Furthermore, to account for the impact of reverse lookups, i.e. uTLB/TLB lookups based on physical in addition to virtual PageIDs, both, uTLB and TLB are treated as two separate fully associative tag-arrays. While conventional address translations access the virtual and the physical tag-array, simple uWT/WT updates in consequence of cache line fills and evictions only access the physical tag-array. The replacement policies for the uTLB and TLB are second chance and random, respectively (Section 5.3.2).

Preliminary simulations showed that port conflicts due to competing address translations and uTLB/TLB updates are rare. Hence, the number of uTLB/TLB ports for the analyzed MALEC implementation is the same as was used in Section 4.4.1. In particular, as the uTLB tends to handle the vast majority of page translations (>95%, Fig. 5.9b) and updates due to cache line fills, actual TLB-WT accesses are primarily limited to updates on cache line evictions. Furthermore, the last entry register (Fig. 5.4) completely avoids uTLB lookups in consequence of conventional cache accesses hitting the cache. WT and uWT are implemented using a shared read/write port and separate read and write ports, respectively.

In addition to the previously analyzed cache configurations stated in Tab. 4.2, Section 5.4.2 introduces *MALEC_3cycleL1* and *Base2ld1st_1cycleL1*. While the latencies introduced by Page-Based Way Determination are not expected to impose additional CPU cycles, *MALEC_3cycleL1* illustrates a MALEC implementation that requires three instead of two cycles per L1 access. Similarly, *Base2ld1st_1cycleL1* represents a VIPT implementation of *Base2ld1st* that performs L1 accesses within a single cycle. Note that the energy values given for the VIPT implementation in Section 5.4.3 are based on the same (slow, but low energy) transistors used for all other configurations and do not account for additional circuitry required for VIPT. Hence, they present the best case scenario for VIPT. To allow the comparison of Page-Based Way Determination against other way prediction schemes, Section 5.4.3 analyses an implementation of Nicolaescu et al.'s [49] Way Determination Unit (Section 2.1.3). To allow fair comparisons, the scheme was modified to benefit from reduced cache accesses. A comparison to actual "way prediction" techniques was discarded, due to the undesired effects of latency and energy consumption imposed on false predictions. Although "way estimation" techniques do not exhibit this problem, they tend to deliver multiple results (i.e. potential cache ways) per access, which consumes energy for unnecessarily activated tag- and data-arrays.

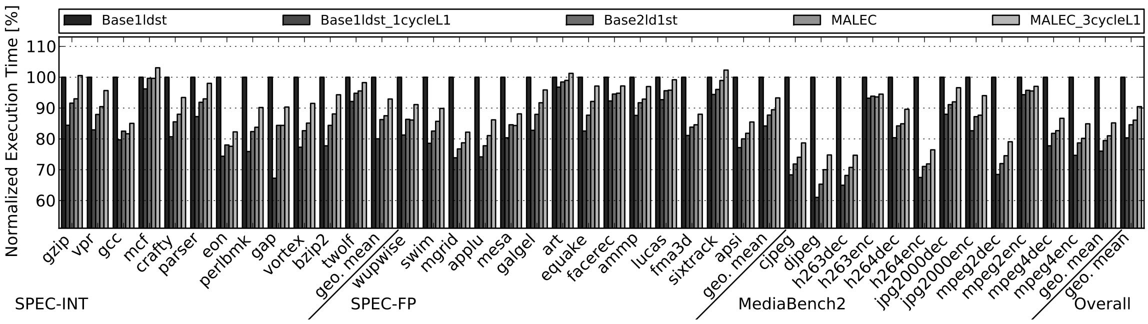


Fig. 5.6: Normalized performance of - in order left to right - *Base1ldst*, *Base2ld1st_-1cycleL1*, *Base2ld1st*, MALEC and *MALEC_3cycleL1*

5.4.2 Performance

Similarly to Fig. 4.6 in Section 4.4.2, Fig. 5.6 illustrates the performance of the analyzed configurations normalized to *Base1ldst*. As the latencies introduced by Page-Based Way Determination are considered small enough not to require an additional CPU cycle per L1D access, MALEC's performance is unchanged. The introduction of an additional cycle in *MALEC_3cycleL1* reduces its average performance benefit from 14% to 10%, and even results in a performance degradation of up to 4% relative to *Base1ldst*. In contrast, the single cycle implementation of *Base2ld1st* exhibits an average performance improvement from 15% to 20%. An explanation for the exceptional high improvement (≈17%) for gap is the combination of a high proportion of loads (≈37%; Fig. 3.1) on the overall instruction count and the frequent execution of instruction sequences that exhibit dependencies that prevent re-ordering; i.e. load latencies cannot be overlapped with the execution of other instructions. Comparing the three benchmarks suites against each other indicates that SPEC-Int is more sensitive to L1 access latency variations. This is primarily due to less regular execution patterns and the higher proportion of loads and stores within this suite; i.e. 45% compared to 40% and 38% for SPEC-FP and MB2, respectively (Fig. 3.1).

5.4.3 Energy

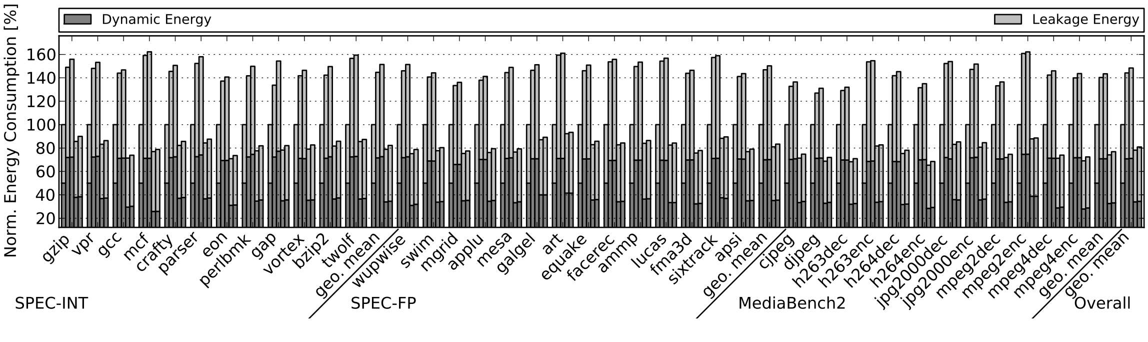


Fig. 5.7: Normalized energy consumption of - in order left to right - *Base1ldst*, *Base2ld1st_-1cycleL1*, *Base2ld1st* MALEC and *MALEC_3cycleL1*

The dynamic and overall energy consumption of the configurations analyzed in the previous section are illustrated in Fig. 5.7. As mentioned in Section 5.4.1, these results do not account for additional circuitry required for VIPT or the effects of transistor scaling (faster

⇒ more energy); hence, they represents worst- and best-case scenarios for *MALEC_-3cycleL1* and *Base2ld1st_1cycleL1*, respectively. The dynamic energy consumption of both adaptations (±1 cycle) is almost equivalent to their corresponding baseline configurations. While *MALEC_3cycleL1* imposes less than 1% additional dynamic energy consumption for all analyzed benchmarks, *Base2ld1st_1cycleL1* cuts gap's consumption by approximately 5%. The latter indicates a reduced number of memory accesses. One explanation for this is the faster computation of branch outcomes and targets, due to shorter load latencies, which reduces the number of speculatively executed instructions. In particular, while the number of committed loads is unaffected by *Base2ld1st_1cycleL1*, the number of speculatively executed loads is ≈ 9% lower than for *Base2ld1st*.

The minor impact of both adaptations on leakage originates from their small performance influence (Section 5.4.2). MALEC's overall energy consumption increases by less than 2%. Except for gap, *Base2ld1st_1cycleL1* reduces this value by no more than 5%. However, this effect is expected to be mitigated by the additional energy required for the implementation of faster transistors and additional circuitry to realize VIPT.

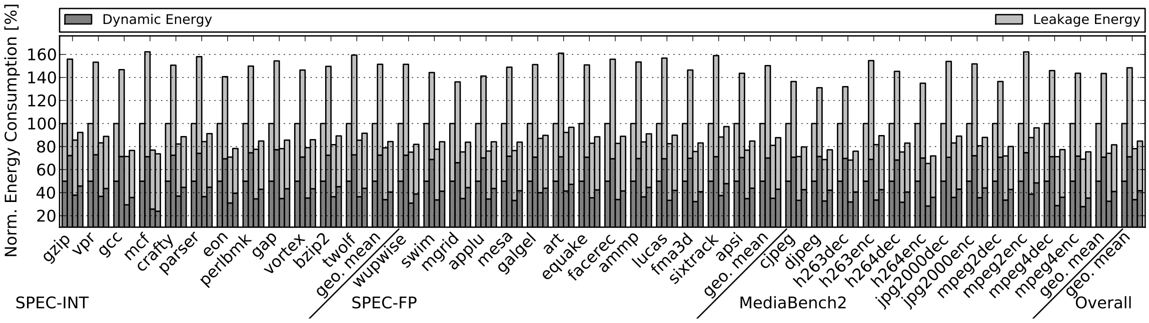
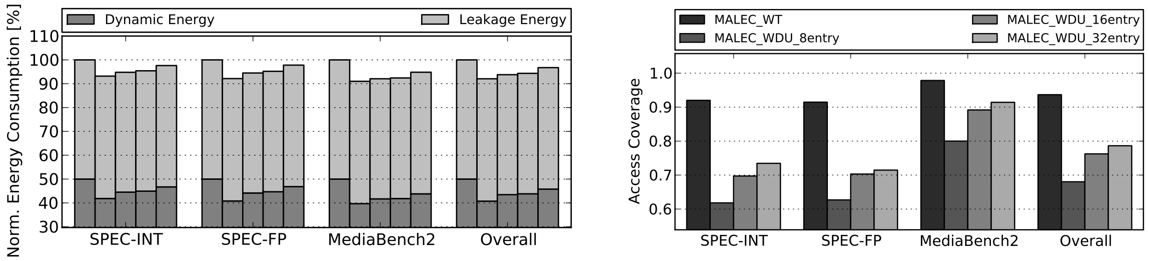


Fig. 5.8: Normalized energy consumption of - in order left to right - *Base1ldst*, *Base2ld1st*, MALEC with WTs and MALEC without WTs

To estimate the impact of Page-Based Way Determination on MALEC's energy consumption, Fig. 5.8 directly compares MALEC configurations with and without WTs. It can be observed that the implementation of WTs reduces MALEC's dynamic and overall energy consumption by ≈ 8%. A major exception is mcf, which consumes approximately 3% more energy. As previously discussed in Sections 3.3 and 4.4.3, mcf exhibits memory access patterns typical for streaming applications. High miss-rates in consequence of very low temporal and spatial localities drastically impair the efficiency of WTs. Note that orthogonal to MALEC are proposals that identify applications, threads, or program phases with low temporal locality and encourage allocation policies to bypass high level caches for accesses to specific memory regions (e.g. on page or instruction granularity). Examples would be compiler inserted hints or hardware monitors. It can be expected that those proposals would further improve MALECs energy efficiency.

The leakage energy required by the high number of transistors employed to store way and validity information in uWT and WT partially mitigates the dynamic energy savings induced by Page-Based Way Determination. Consequently, alternative way determination schemes may be considered. Nicolaescu et al.'s [49] Way Determination Unit (Section 2.1.3)



(a) Normalized Energy Consumption of - left to right - MALEC without Way Determination, MALEC with WTs, MALEC with WDU 8, 16 and 32 entries

(b) Ratio of L1 Accesses Covered by Way Determination to Overall Access Count

Fig. 5.9: Comparison of Page-Based Way Determination and various WDUs sizes

was originally designed for energy oriented processors similar to *Base1ldst*. It holds way information of recently accessed cache lines in a small buffer structure called WDU. The scheme was adapted to the analyzed MALEC configuration by extending the WDU with two additional ports and validity bits allowing up to three parallel fully associative lookups and guaranteeing the 100% accuracy required for reduced cache accesses (Section 5.3). Fig. 5.9a compares MALEC configurations without way determination, with WTs and with WDUs holding 8, 16 and 32 entries. The average dynamic energy consumption of the best WDU setup (8 entries) is comparable to the proposed Page-Based Way Determination scheme (approximately 3% higher). There are two reasons for this. First, in contrast to the single ported WTs, WDUs require three ports to service the up to three requests handled in parallel by the analyzed MALEC configuration. As it performs fully associative lookups of tag-sized address fields, the energy per WDU access is similar to an uWT access (approximately 30% less for 8 WDU versus 16 uWT entries). Second, with average values of less than 80% rather than 94% (Fig. 5.9b), all WDU configurations cover significantly fewer memory accesses than WTs, and therefore suffer the energy penalty of an increased number of conventional instead of reduced cache accesses. In contrast, accounting for the low leakage power of the scheme, originating from its small size, its overall energy consumption approaches the default MALEC configuration closely (approximately 2%, Fig. 5.9a). Generally speaking, due to their small requirements in terms of storage capacity, WDUs are suitable for processor configurations similar to *Base1ldst* that are designed for a low number of parallel memory accesses per cycle. In contrast, as the energy consumption of page-based way determination is independent of the number of memory references to be serviced in parallel, its energy efficiency increases rather than decreases on high performance processors.

Another approach to improve MALEC's leakage power consumption is to reduce the number of uTLB entries and therefore the uWT size. However, this effectively reduces the uWT's coverage, which increases the dynamic energy consumed due to more WT accesses. Simulations show that for the analyzed MALEC configuration this trade-off leads to overall energy consumption widely independent of the uTLB size; i.e. the energy difference between 4, 8, 16 and 32 entries is no more than 1%. Abandoning the WT completely in favor of lower leakage significantly reduces the number of L1 accesses covered by way determination. A simulation using a 16 entry uWT and no WT achieves an average coverage of approximately 70%, increasing MALEC's overall energy consumption by 5% in average. Similar effects can be observed for an implementation that does not perform uWT updates in consequence of conventional cache accesses hitting the cache (average coverage reduced to 80%). In particular, mcf would actually consume 5% more instead of 51% less dynamic energy due to a high number of invalid way predictions.

5.4.4 Sensitivity Analysis

Suitability for SMT and CMPs

Similar to MALEC, Page-Based Way Determination was developed in the context of uni-core processors, but could be adapted to multi-core environments. A key concern for this case would be the ability of reduced cache access to bypass tag-arrays. As those accesses are often used to enforce coherency between L1Ds corresponding to different cores, either the way determination scheme or the coherency protocol would need to be adapted. The former could simply perform conventional tag-array accesses even for valid way predictions. However, this would entail increased energy consumption. The latter might separate status bits used by the coherency protocol from the tag-array and only active those bits during reduced cache accesses. While this approach would be more energy efficient, its complexity might prove it infeasible.

Due to the use of physical addresses as prediction source, the general operation of WTs is not effected by simultaneous multithreading. To allow simultaneous way predictions for two or more threads, additional uWT and potentially WT ports might be required.

Latency Overhead

As uTLB lookups are faster than TLB lookups, uTLB+uWT accesses can be implemented without introducing additional latency. Although the latency of WT accesses are relatively small (single ported, lookup already done during TLB access), they cannot be hidden in this manner. For the implementation described in previous sections (at 1GHz) this latency was small enough not to introduce an additional cycle. Should this not be the case, it might be considered to execute WT accesses within an optional cycle (i.e. uTLB miss but TLB hit) between address translation and cache access. However, this would further increase variability in load latency as discussed in Section 4.5. The performance and energy impact of such an implementation is discussed in Section 5.4.

Scalability

A key feature of Page-Based Way Determination is its scalability in terms of parallel L1D accesses. In fact, its efficiency is proportional to the number of memory request simultaneously issued by the processor. The reason for this is the increased probability for multiple accesses to the same page, which are able to share way information originating from a single WT entry. The maximum number of ways read from a single WT entry is equal to the number of available cache banks (four for the system in Fig. 4.2). However, as the WT entry arrangements proposed in Section 5.3.3 allow the use of simple decoders instead of complex lookup structures to index specific lines, the energy impact of additional cache banks on Page-Based Way Determination is considered negligible.

Cache/Memory Parameter Dependencies

The relevance of cache parameters to Page-Based Way Determination is similar to those described for MALEC in Section 4.5. However, while wider cache lines effectively reduce the number of bits required per WT entry (Section 5.3.3), they do not affect the address space (i.e. one page) covered per uWT/WT access. Hence, wider cache lines increase the efficiency of page-based way determination. Although higher cache associativities require wider WT entries, the additional energy saved due to reduced cache accesses actually increases the efficiency of page-based way determination. Memory systems that use wider address spaces (e.g. 40 or 48-bit) exhibit larger tag-arrays and an increased number of bits within virtual and physical page IDs. Hence, the efficiency of reduced cache accesses and the energy savings due to the re-use of TLB lookups to index WTs are further improved.

Finally, as larger pages (e.g. 16K or 64K) significantly increase the number of lines held per WT entry, the Page-Based Way Determination requires TLB entries to be quantized in 4 KByte segments when entering the uTLB. The WT itself can be segmented into a number of chunks, each representing data corresponding to a 4 KByte address region. By allocating and replacing chunks in a FIFO or LRU manner, their number can be smaller than required to represent full pages. Note that an alternative is the implementation of a speculative address translation scheme that mitigates penalties due to TLB misses. It essentially allows the use of small pages to achieve fine-grained allocation and protection, while avoiding the associated latency penalties of small pages [86].

Exception Handling

All information held by WTs is non-speculative and based on physical addresses. Hence, it is unaffected by miss-speculations and therefore does not impact any recovery mechanism.

5.5 Miss-Prediction to Completely Avoid Tag-Array Accesses

Another concept in the context of WTs is referred to as miss prediction. It assumes that all WT entries are in perfect sync with the tag-array. Consequently, an access to an invalid line within a WT entry can be interpreted as an L1 miss and treated as such. The otherwise blocked tag- and data-arrays can therefore be accessed by other memory references or left idle to save dynamic energy. The primary problem of this concept is way information lost between TLB evictions and later reallocations. A possible solution would be the storage of evicted WT entries within a separate structure or a lower cache level. However, besides requiring a considerable amount of circuitry, this would raise additional questions; e.g. on how to invalidate evicted cache lines within those WT entries. A simpler method might store only the physical page IDs of pages that have been evicted by the current thread. Ends the thread or is the cache flushed, this data is invalidated. The stored physical page IDs are then checked against each newly allocated TLB entry. WT entries corresponding to previously known pages are flagged as “No Miss Prediction” using a dedicated bit. Hence, while accesses to invalid lines of flagged WT entries would be treated as “unknown” and perform conventional cache accesses, others could be treated as miss predictions. In conclusion, although miss prediction represents a novel and potentially beneficial concept, its practical use is questionable. As most L1 caches exhibit very low miss rates (Fig. A.2), it is hard to justify the additional complexity and energy consumption of the required circuitry.

5.6 Concluding Remarks

This chapter introduced a novel way determination scheme designed in conjunction with MALEC to further reduce its energy consumption. The scheme exploits MALEC’s restriction to access only one page per cycle, by re-using TLB lookups to index small storage structures (WTs). As each WT entry can provide way information for all cache lines mapping to the corresponding page, Page-Based Way Determination is highly scalable in terms of parallel accesses per cycle. The performance penalty induced by the scheme is relatively low, because the associated components are simple (single-ported, lookup free) and in most cases able to hide their latencies behind other circuitry. Based on the simulation environment underlying the previous chapter, the scheme achieves an access coverage of $\approx 94\%$ and reduces MALEC’s energy consumption by an average of 8%.

The engineering contributions underlying this chapter are described in Section 5.4.1. In addition to the modifications discussed in the previous chapter, gem5 was extended with the support for way tables, advanced replacement policies for TLBs, and mechanisms to index, read, write and evaluate various types of WT entries. Furthermore, detailed statistics and access counters were introduced into the cache model to provide way specific information. Finally, the impact of different WT entry types (Section 5.3.3) on miss rates was evaluated and mechanisms to implement miss-prediction (Section 5.5) tested.

6 | Vector Benchmarking

6.1 Motivation

The scalar benchmarks used for the initial analysis of Page-Based Memory Access Grouping in Chapter 4 show significant energy savings for moderate performance degradation. However, even executed on a high performance processor configuration (Tab. 4.1), SPEC2000 and MB2 do not exert sufficient memory pressure to fully utilize MALEC. The Vector Benchmark Suite (VBench) introduced here addresses this by implementing scalar and vectorized versions of several popular algorithms on a configurable simulation environment.

Individual algorithms were selected to represent computational patterns commonly observed in general purpose code. Those patterns do not exhibit sufficient regularity to be efficiently processed with ARM's current vector ISA extension (NEON), which was originally designed for multimedia and signal processing algorithms. To increase the vectorizability of VBench and consequently the pressure on MALEC, Section 6.2 introduces ARGON. This NEON based vector extension supports several advanced SIMD features that were previously exclusive to HPCs and only recently considered for commercial microprocessors [24]. However, wider data types, registers and datapaths, as well as per-lane predication and indexed memory accesses impose considerable costs in terms of silicon area, design complexity and power consumption. Moreover, as automatic vectorization techniques are heavily constrained by the presence of dynamic data dependencies [87], advanced SIMD features have been commonly confined to hand-optimized kernels and libraries. ARGON and the development toolchain based on it are designed to allow the analysis of trade-offs between imposed complexity and achievable benefits over a range of distinct design points.

ARGON and VBench were developed in cooperation with ARM Holdings in Cambridge with key contributions by: Alastair Reid (ARGON ISA), Dr. Giacomo Gabrielli (gem5 framework), Dr. Mbou Eyole (benchmark selection and handling of segmented scans), Harsh Kumar (assembler/disassembler framework), and Wojciech Meyer (compiler support for ARGON). The following sections give a brief overview of ARGON (Section 6.2), the analyzed algorithms (Section 6.3), the toolchain used to develop and evaluate VBench (Section 6.4), and a quantitative analysis of scalar, NEON and ARGON implementations of each algorithm (Section 6.5). The chapter concludes with a list of recommendations for future SIMD architecture extensions to increase the vectorizability of workloads and the utilization of the underlying datapath and memory system (Section 6.6).

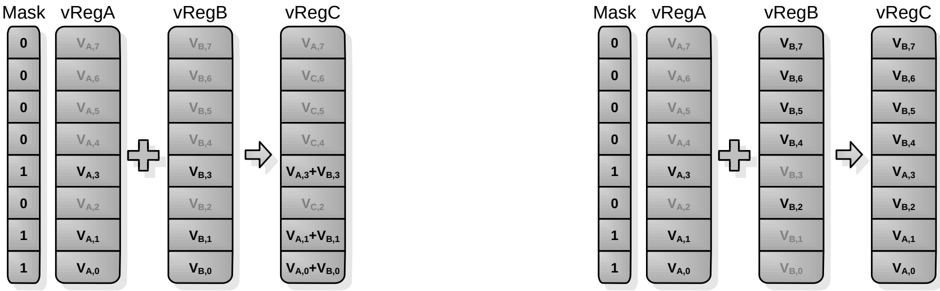
6.2 ARGON: Extending NEON to Handle Irregular Computation Patterns

Key challenges for the vectorization of general purpose code are irregular computation patterns, data dependencies and the need for balanced computation paths. ARGON is a superset of ARMv7 NEON that incorporates the same data-processing instructions, but was extended to support the features listed below to increase its applicability. To avoid negative impacts on scalar performance, ARGON limits interactions between the scalar and vector register file to dedicated directives such as “get/set lane” and “fill vector with scalar value”. Most vector instructions exclusively operate on vector registers.

6.2.1 Per-Lane Predication

```
for (i = 0; i < VLmax; i++) {
  if (A[i] > 0) {
    C[i] = A[i] + B[i];
  }
}
```

```
for (i = 0; i < VLmax; i++) {
    C[i] = (A[i]>B[i]) ? A[i] : B[i];
}
```


Mask
vRegA
vRegB
vRegC
0 0 0 0 1 0 1 1
$V_{A,7}$ $V_{A,6}$ $V_{A,5}$ $V_{A,4}$ $V_{A,3}$ $V_{A,2}$ $V_{A,1}$ $V_{A,0}$
$V_{B,7}$ $V_{B,6}$ $V_{B,5}$ $V_{B,4}$ $V_{B,3}$ $V_{B,2}$ $V_{B,1}$ $V_{B,0}$
$V_{A,7}$ $V_{C,6}$ $V_{C,5}$ $V_{C,4}$ $V_{A,3}+V_{B,3}$ $V_{C,2}$ $V_{A,1}+V_{B,1}$ $V_{A,0}+V_{B,0}$
Mask
vRegA
vRegB
vRegC
0 0 0 0 1 0 1 1
$V_{A,7}$ $V_{A,6}$ $V_{A,5}$ $V_{A,4}$ $V_{A,3}$ $V_{A,2}$ $V_{A,1}$ $V_{A,0}$
$V_{B,7}$ $V_{B,6}$ $V_{B,5}$ $V_{B,4}$ $V_{B,3}$ $V_{B,2}$ $V_{B,1}$ $V_{B,0}$
$V_{B,7}$ $V_{B,6}$ $V_{B,5}$ $V_{B,4}$ $V_{A,3}$ $V_{B,2}$ $V_{A,1}$ $V_{A,0}$

(a) Element Selection

(b) Permutations

Fig. 6.1: Potential use cases for mask registers (vReg... vector registers)

Similarly to the conditional execution of instructions within branch statements discussed in Section 3.4, so called vector masks (VMs) may be used to selectively operate on individual vector elements. Fig. 6.1a illustrates this based on a conditional addition inside a loop statement. A second use case for VM is the permutation of elements as exemplified in Fig. 6.1b.

```
rest  = nElem & (VLmax - 1);
vElem = nElem - rest;

for (i = 0; i < vElem; i += VLmax) {
  ... loop body (vectorized) ...
}
for (i = vElem; i < nElem; ++i) {
  ... loop body (scalar) ...
}
```

```
rest  = nElem & (VLmax - 1);
vElem = nElem - rest;

vsetVL(VLmax);
for (i = 0; i < vElem; i += VLmax) {
  ... loop body (vectorized) ...
}
vsetVL(rest);
... loop body (vectorized) ...
```

Fig. 6.2: Using VL to avoid scalar fix-up operations following vectorized loops

A second concept commonly used to limit the number of active elements is based on the vector length. The maximum vector length (VL_{max}) is defined as the maximum number of elements to be held by a vector register. As ARGON operates on packed vector registers, VL_{max} for a specific data type results to $SIMD_{width}/element_{width}$ (Section 1.4.3). A vector length register (VL) enables the vectorization of loops that exhibit iteration counts unequal to a multiple of VL_{max}, without the need for scalar fix-up operations. Note that the example given in Fig. 6.2 may be rewritten to adapt VL for the last iteration of the loop. This would improve code density by avoiding the replication of the vectorized loop body; however, the increased instruction count and the additional control dependency inside the loop usually result in a performance penalty over the presented version.

ARGON was design to incorporate exactly one VM and one VL, which are implicitly used by all applicable vector instructions. This allows ARGON's implementation without the need for additional opcode bits to specify masks and/or vector lengths with each instruction, and simplifies the underlying hardware. However, later evaluations revealed that multiple VM&VL or at least the ability to discard them would be desirable; e.g. to handle sequences that alternate between instructions that do / do not use VM&VL, or require multiple masks. In particular, segmented scans require a second mask to allow predication, because the contents of VM are currently used to delineate segments (Section 6.2.3).

6.2.2 Indexed Memory Accesses (Gather/Scatter)

Indexed memory accesses allow the vectorization of data dependent-memory accesses, e.g. pointer chasing when traversing linked lists. Further details and an example can be found in Section 1.4.3. The implementation chosen for ARGON uses a scalar base address and a vector of 32-bit offsets. Applying 32-bit offsets independent of the underlying data type, i.e. 8-, 16-, 32- or 64-bit, may require multiple instructions to service all elements of a vector register. However, using 8-bit offsets to access 8-bit elements would limit the addressable memory region to 256 bytes.

6.2.3 Scans

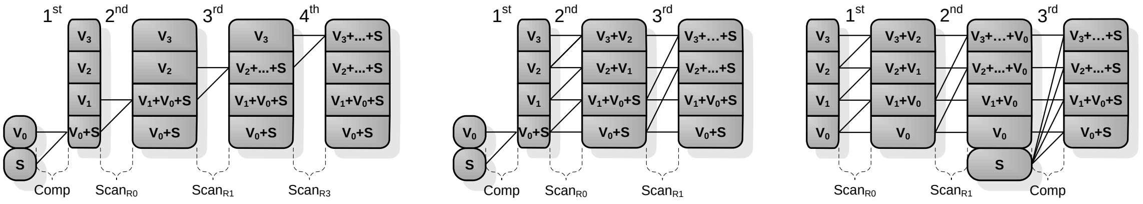
```
sum = 0;
for(i = 0; i < n*VLmax; ++i){
  sum += data[i];
}
```

Fig. 6.3: Scalar scan add

```
sum = 0;
for(i = 0; i < n*VLmax; i+=VLmax){
  vData = vld1(data+i);
  {sum, vData} = viscanadd(sum, vData);
}
```

Fig. 6.4: Vectorized scan add

Scans - also called prefix-sums - are primitives that perform cumulative operations over sequences of vector elements. They may benefit algorithms that need to identify specific elements within a data set (e.g. the minimum), or reduce multiple elements to a single outcome (e.g. sum). Fig. 6.3 and Fig. 6.4 illustrate scalar and vectorized variants of a scan add. The latter operates on a scalar and a vector register to allow a scalar (carry) to be propagated between multiple iterations. For this example the number of elements to be added was chosen to be a multiple of $\mathrm{VL}_{\mathrm{max}}$. Other values would require the use of VL as described in Section 6.2.1.



(a) Serialized (b) Add Scalar first (c) Add Scalar last

Fig. 6.5: Implementation examples for 4-element vector scan add

Fig. 6.5 illustrates three implementation variants for a vector scan add. The serialized variant in (a) requires N stages and N operations; with N being the number of elements per vector. It might be found in low-end systems that shy away from the hardware costs associated with scans. It exhibits a low operation count, but does not exploit potential performance gains from parallelization. The other two variants perform the operations used to process the vector elements in only $\log_2$N stages (Scan_{R0}, Scan_{R1}, ...). While (b) processes the scalar input / carry first, (c) delays this computation until the end (Comp). Both variants require $\log_2$N+1 stages; however, the former uses $1 + \sum_{i=0}^{log_2 N-1}(N - 2^i)$ and the later $N + \sum_{i=0}^{log_2 N-1}(N - 2^i)$ operations (e.g. 6 and 9 for the given example). The lower operation count indicates a higher efficiency for the former variant, but the need to employ a scalar arithmetic logic unit (ALU) for the initial stage increases its complexity and potentially impacts the performance of scalar computations. Variant (b) is also not compatible with Scan Partitioning as proposed in Section 7.3 and used for analyses related to multi-register operations (Section 7.7).

To obtain the same results from all three variants of Fig. 6.5, the underlying operation (add) is assumed to be associative. This is not the case for certain floating point operations. Hence, the accuracy of vector instructions involving FP values over a wide range of exponents may be affected. This is not unique to SIMD engines, but also known on GPUs and multi-core processors. Algorithms sensitive to FP associativity generally do not lend themselves to parallelization. A system targeting them may provide a flag or opcode bit to request the serialized execution of FP scans. To investigate the performance impact of serialized scans, the timing profiles introduced in Section 6.4.2 are based on (a) and (c).

Note that all variants given in Fig. 6.5 return intermediate results in form of a vector register. This distinguishes scans from reductions, which only return a single scalar result equivalent to a scan's carry output. While reductions may require fewer operations [88], systems usually implement the more generic scans. Besides the variants presented here, several alternatives have been proposed in the context of high performance adders and parallelization guidelines for recurrence equations [89]. However, given the restriction to operate on a maximum of two elements within each individual FU (e.g. ALU, multiplier), the number of stages required to compute N elements is usually no less than $\log_2$N.

6.2.4 Segmented Scans

```
for(i = 0; i < x; ++i){
  for(j = 0; j < y[i]; ++j){
    ... loop body (scalar) ...
  }
}
```

(a) Scalar

```
for(i = 0; i < x; ++i){
  rest  = y[i] & (VLmax - 1);
  vElem = y[i] - rest;
  vsetVL(VLmax);
  for(j = 0; j < vElem; ++j){
    ... loop body (vectorized) ...
  }
  vsetVL(rest);
  ... loop body (vectorized) ...
}
```

(b) Vectorized

```
/* (nElem = Σ_{i=0}^{x} y[i]) */
rest  = nElem & (VLmax - 1);
vElem = nElem - rest;
vsetVL(VLmax);
for(i = 0; i < vElem; ++i){
  ... loop body (segmented) ...
}
vsetVL(rest);
... loop body (segmented) ...
```

(c) Segmented

Fig. 6.6: Examples for the computation of a nested loop

Segmented scans are a derivative of scans that allow arbitrary length segments within vectors to be processed in parallel; i.e. perform multiple independent scans within a single vector. Fig. 6.6 illustrates how conventional scans in combination with VL may be use to vectorize the innermost of two nested loops. While this is sufficient in many cases, certain algorithms or working sets may not provide enough elements within the inner loop to fully utilize available SIMD units. Segmented scans circumvent this issue by collapsing the two loops into one and computing $\mathrm{VL}_{\mathrm{max}}$ elements in each iteration.

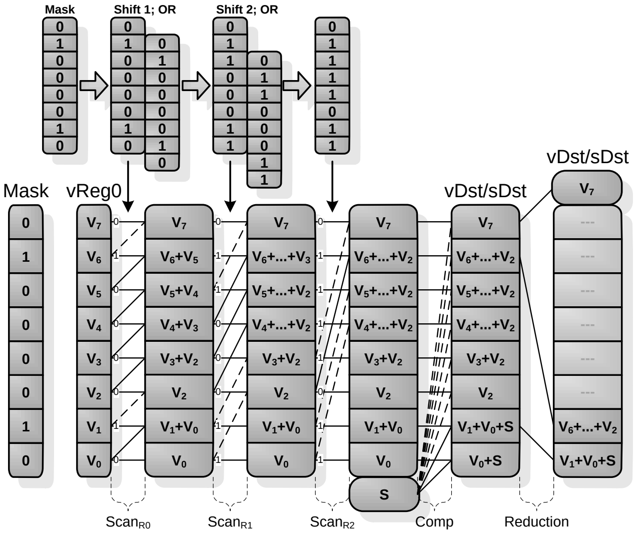


Fig. 6.7: Example for segmented scan / reduction (Add)

Fig. 6.7 depicts an implementation variant based on Fig. 6.5c allows the re-use of corresponding FUs. A “1” within the mask indicates the end of a segment and - depending on the current stage - determines which operations are substituted with moves (dashed lines in Fig. 6.7). Details on the compute descriptor instruction used to generate the mask and on how those substitutions may be implemented efficiently can be found in Sections 7.2 and 7.4, respectively. The final stage of the example, illustrates a segmented reduction. For the timing profiles introduced in Section 6.4.2, it is assumed that this step may be performed by the same multiplexer network already employed to align results for their storage in packed register format. Hence, this step does not impose an additional latency. Alternative implementation variants may process individual segments in separate FUs to speed up the computation of multiple short segments; however, their additional hardware complexity is considered too costly.

6.2.5 Datapath Width and Double-Length Register Operations

Besides investigating the performance impact of 128-, 256- and 512-bit wide datapaths, the following sections also investigate the impact of double length (L-type) instructions based on the proposal in Section 7.7. These allow operands to reference two physical registers in form of a single logical register. For example, $vaddL(VW_0, VW_1)$ would be executed by pipelining $V_0 + V_1$ and $V_{16} + V_{17}$. The timing profiles introduced in Section 6.4.2 furthermore investigate datapath widths equivalent to a) half the available register width and b) one element. The former might be employed by mid- and high-end systems to trade off the hardware costs associated with specific vector FUs against achievable speed-ups. The later may be used by low-end systems for similar purposes or generally to allow fully associative FP scans (Section 6.2.3).

6.3 Algorithm Characterization

The list of algorithms presented in Tab. 6.1 was inspired by the Berkeley Dwarfs [90]; i.e. a set of numerical methods believed to mirror the key challenges that conventional architectures face in exploiting parallelism. VBench deliberately attempts to widen the scope of vector processing from multimedia and signal processing algorithms, as these can often be more efficiently computed on high throughput engines such as GPUs, DSPs and customized FPGAs. Tab. 6.1 labels individual implementations as follows:

- **Scalar:** reference based on existing libraries or benchmarks
- **NEON:** ARMv7 NEON variant. Not always considered worthwhile; i.e. results from insufficiently vectorizable algorithms would primarily represent scalar fix-up statements.
- **ARGON:** ARGON variant without segmented scans
- **ARGON SegScan:** ARGON variant with segmented scans. Code varies substantially from variants relying on single-segment scans. Initialization phase and restructured data arrays can exhibit significant performance impact.
- **ARGON SegScan Min1:** SpMV implementation that assumes at least one non-zero element per row. Hence, scalar fix-up statements are not required.
- **ARGON Hybrid:** PathFind implementation that vectorizes extraction of minimum from d-heap, but decreases keys in a serialize manner. Analyses beyond those presented in Section 6.5 show benefits in the context of small workloads with heaps too small to efficiently utilize vector FUs.

Additional algorithms to those listed in Tab. 6.1 were investigated, but are not further discussed due to behaviors very similar to those presented. For instance, FIR filter and k-means clustering algorithms rely on nested loops, whereby the iteration count of the innermost loop is workload dependent and/or varies at runtime. Both lend themselves to segmented scans, but are considered too similar in nature to BackProp and SpMV to justify separate analyses. The results presented in Section 6.5 are based on the fastest of the developed implementation variants. For instance, multiple ways to perform the indexed memory access of 16-bit values based on 16-bit offsets required by BitAlloc ARGON have been investigated; e.g. pro-/demote offsets and results, limit the number of utilized 16-bit elements to $1/2$ VL_{max}, use a scalar fix-up instead, ect. However, all variants are functionally equivalent and their behaviors considered too similar to be individually presented.

While gcc with the highest optimization level was employed to compile all analyzed implementations, the vectorized portions were hand-coded using intrinsics to leverage features introduce by ARGON. Library code was substituted with optimized inline functions, when it would have resulted in discrepancies between scalar and vectorized code. For example, calls to C math reduce BackProp Scalar's performance by a factor of 3. Similar attention was paid to system calls to minimize OS interactions and other system-level effects. Assembly level optimizations such as processor aware instruction scheduling and manually placed pre-fetch instructions were considered too hardware specific and therefore avoided.

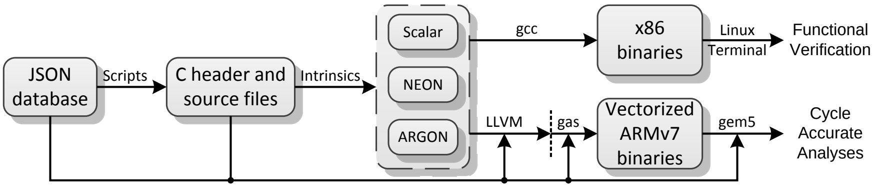
	Benchmark Name and Description	Working Set Params.	Implementation Specific Considerations		Utilized Features
AESEnc	Advanced Encryption Standard as established by NTIS in 2001 [91]; very computation-intensive; primarily focused on combinational logic; exhibits nonlinearly dispersed memory accesses	CTR (counter) mode; 128-bit keys	Scalar & ARGON	32-bit version combining multiple round transformation steps in a single set of table lookups	indexed mem. acc.; L-type operations; VL (to a minor degree)
			NEON	Lack of support for indexed loads favors 8-bit SIMD version	
BackProp	Backpropagation; used to train neural networks [92]; Parameters outside the default working set, do not significantly affect the observed performances, except for cases where the number of elements per layer exactly matches the vector width; these improve the results observed for the ARGON variant	3 input and 2 output nodes; 1 hidden layer including 5 neurons; 8 training sets; randomly initialized weights	Scalar	—	indexed mem. acc.; VL; VM; scans / segmented scans
			NEON	Lack of advanced features makes it not worthwhile[1]	
			ARGON	Based on scans and VL; processes one neuron at a time	
			ARGON SegScan	Segmented scans to process multiple neurons in parallel; initialization phase to precompute indexes and segmentation masks (descriptors)	
BitAlloc	Bit Allocation; elemental step in compression and optimization algorithms; analyzed variant models the allocation of bits to carriers within a DSL transmission channel based on their SNR (16-bit integers)	256 carriers equivalent to working set provided by EEMBC TeleBench v1.1 [93]	Scalar	—	L-type instructions; VM; scans; VL (to a minor degree)
			NEON	Lack of VM allows only partial vectorization	
			ARGON	Employs VM to remove control dependencies; almost fully vectorized	
PathFind	Dijkstra's Algorithm; fastest single-source shortest path algorithm for arbitrary directed graphs with unbounded non-negative weights [94]; inherently scalar nature restricts vectorization to two primary functions, extractMin() and decreaseKey(); performance of vectorized variants is proportional to heap size	New York City road map provided by the 9th DIMACS Implementation Challenge [95]	Scalar	Binary heap stored as consecutive array to simplify address calculations and speed up memory accesses	indexed mem. acc.; VL; VM; scans
			NEON	Lack of advanced features makes it not worthwhile[2]	
			ARGON	d-heap; d equal to number of 32-bit elements per vector	
			ARGON Hybrid	Based on ARGON variant; serialized execution of decreaseKey(); analyses on smaller working sets show benefit over ARGON variant	
SpMV	Sparse matrix-vector multiplication; integral building block for a variety of science and engineering applications; generally involves matrices with a large number of zero elements stored in a compressed format; analyses varying the element density while keeping the matrix size constant show similar trends to those presented in Sec. 6.5.5	randomly generated matrices in Yale format; sizes ranging from 2*2 to 1000*1000 (default 100*100) elements; default density of 15%	Scalar	—	indexed mem. acc.; VL; VM; scans / segmented scans
			NEON	Mimics indexed mem. accs. by using scalar loads	
			ARGON	Based on scans and VL; processes one row at a time	
			ARGON SegScan	Segmented scans to process multiple rows in parallel; segmentation masks computed within every iteration	
			ARGON SegScan M1	Assumes at least one non-zero element per row; avoids conditional branch used for scalar fix up	

[1] Without VL, NEON is limited to network layers with at least $\mathrm{VL}_{\max}$ neurons. Results obtained for smaller layers only represent scalar fix-up statements. Other components lacking are scan adds and non-unit strides. NEON possesses the latter, but does not allow the stride distance to be specified as an operand; i.e. workload dependent. The padding of values in memory can mitigate some of these issue, but not sufficiently to consider a NEON implementation worth while.

[2] Operations on a vectorized d-heap require scans to identify the smallest child of a specific node. VM is used to promote, demote and insert children. Indexed memory accesses allow nodes of different heap levels to be gathered/scattered. Without either of those features a NEON implementation would be mostly scalar.

Tab. 6.1: Analyzed benchmarks, corresponding working sets and implementations

6.4 Development Toolchain



- **gas:** GNU Assembler
- **gcc:** GNU Compiler Collection
- **gem5:** gem5 Simulator System
- **JSON:** JavaScript Object Notation
- **LLVM:** Low Level Virtual Machine

Fig. 6.8: Overview - VBench development toolchain

The environment used to evaluate ARGON was designed with two key requirements in mind: 1) *automation*, as the ISA is intended as a testbed for additional features, and 2) *configurability*, to allow evaluation of different points of the design space. The resulting toolchain illustrated in Fig. 6.8 comprises:

a) **JSON database:** unified representation of ARGON instructions; e.g. opcodes, operand specifications, pipelines accessed, flags read/written, ...

b) **Source files generated from a):** ARGON intrinsics and equivalent C functions

c) **Modified LLVM front-end:** supports ARGON intrinsics

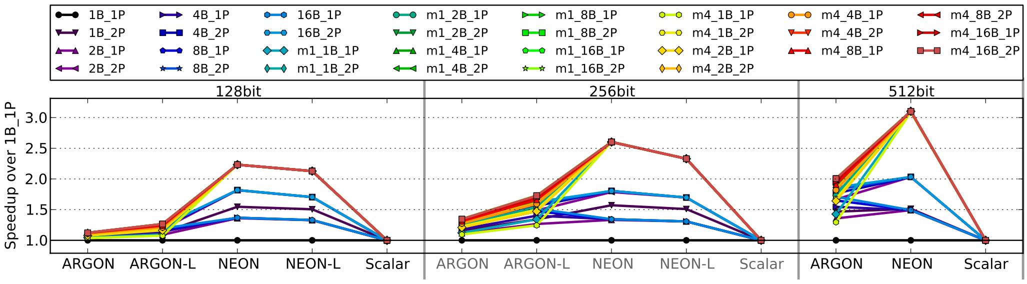
d) **Modified gas:** supports ARGON instructions

e) **Modified gem5:** supports ARGON instructions and hardware features

The micro-architectural models of gem5 were expanded to support a reconfigurable vector register file (128-, 256-, and 512-bit wide), VM, VL, and other resources. Its modified memory model facilitates cache banks, ports and sub-banks, and accurately models contentions (Section 6.4.1). The latencies of vector FUs are parameterizable based on a set of timing profiles (Section 6.4.2). Note that NEON implementations employ the same framework as ARGON variants to allow fair comparisons over a range of register widths, memory configurations and timing profiles. While all implementations were compiled using the highest optimization level, those compiler components responsible for scalar code segments are most mature, leading to a slight performance bias towards scalar implementations.

Most toolchain components are automatically generated from the shared database to reduce turnover times and potential error sources. Individual benchmarks are functionally verified using an x86 based workflow that employs C functions instead of ARGON intrinsics. Moreover, equivalence checks are performed on disassembled binaries that include sequences of all supported ARGON instructions with randomly generated operands. The gem5 framework itself was tested using trace driven analyses, an extensive set of micro benchmarks and a pipeline viewer (Section 6.4.2). Note that the developed toolchain is currently limited to cycle accurate performance analyses. As RTL simulations are not flexible enough to accurately model the datapath components introduced for ARGON for all configurations of interest, energy estimates are as of now considered beyond the scope of the framework.

6.4.1 Cache Configurations



- **Implementations:** Scalar, NEON and ARGON, "-L" indicates use of double-length registers
- **Datapath:** 128-, 256- and 512-bit wide
- **Number of Banks:** 1 to 16 (1B to 16B)
- **Ports per Bank:** single read/write (1P), or 1 read/write & 1 rd (2P)
- **Merging of Accesses to same Line:** disabled (default), enabled on configuration with 1 sub-bank (m1), or limited to max. two consecutive sub-banks of configuration including total of 4 sub-banks (m4)

Fig. 6.9: Performance of AESEnc relative to underlying L1D configuration

As vectorized algorithms exhibit a high number of concurrent memory references, their performance is very sensitive to the underlying memory system. In order to select a subset of relevant cache configurations for further investigation in Section 6.5, all benchmarks were analyzed on 30 different memory interfaces. Based on AESEnc, Fig. 6.9 illustrates speedups achieved over an L1D with 1 bank and a single read/write port. It can be seen that the scalar implementation operates independently from the underlying cache model. The reason for this is its low number of parallel memory accesses. This is due to the fact that the energy optimized baseline configuration used here (Tab. 6.3) relies on vectorization to achieve a high peak performance. This is contrary to the system analyzed in Chapter 4 (Tab. 4.1) that focuses on scalar throughput.

AESEnc was chosen for Fig. 6.9, because the lack of indexed loads favors an 8-bit based NEON implementation relying on unit strides. Compared to the 32-bit based ARGON variant, this implementation is highly memory dependent, which simplifies the process of identifying groups of cache configurations that exhibit similar behavior. While a detailed analysis of Fig. 6.9 is beyond the scope of this section, it can be observed that: a) wider datapaths generally increase the susceptibility to fast memories, b) a higher number of ports is slightly faster than multiple banks, and c) merging[1] is the key performance parameter. The number of sub-blocks considered for merging predominantly affects ARGON variants with a high $\mathrm{VL}_{\mathrm{max}}$. Considering this, and the fact that more than 2 banks and/or ports do not exhibit significant benefits, Section 6.5 focuses on:

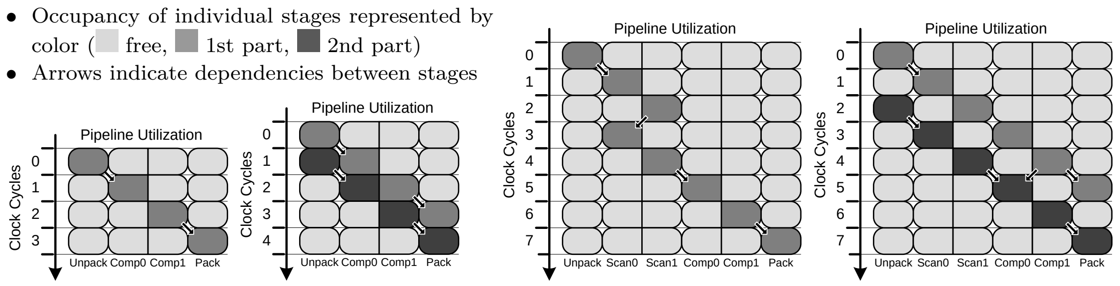
- **1B_1P:** Slowest, simplest and most energy efficient model
- **2B_1P:** Two banks reduce dynamic power consumed per access and allow two simultaneous accesses to different banks

[1] The merging scheme described here differs from MALEC. To allow even low- and mid-range systems to benefit from merging, it is simplified and less intrusive. Operating on address offsets of elements within a single vector in parallel to the address computation phase, it is limited to intra-vector merging. In contrast, an Arbitration Unit handles accesses from scalar reference, elements within the same as well as from different vectors (Section 4.3.3).

- **1B_2P:** Two ports increase dynamic and leakage energy, but allow two simultaneous accesses to different lines in same bank
- **2B_2P:** Two banks with two ports each (1 read/write and 1 read only)
- **m_2B_1P (m4_2B_1P):** Energy benefits of 2B_1P; performance increased by merging accesses to same two subsequent sub-blocks of 4 sub-block cache lines

6.4.2 Timing Profiles

The gem5 model developed for ARGON characterizes FUs by their operation and issue latency; i.e. time until an instruction is ready to commit and time until a subsequent instruction may issue to the same FU. The sharing of resources such as issue queues and bus structures allows multiple logical execution pipelines to be grouped into one physical pipeline. For instance, arranging an integer ALU and a permutation unit into one physical pipeline, limits the number of instructions that may be issued to / committed by either of those FUs. The physical pipelines underlying all subsequent analyses are: i) one scalar pipeline to handle VL&VM updates and data movements between the scalar and vector register files, ii) one vector load and one vector store pipeline, and iii) two vector execution pipelines for all NEON and ARGON instructions. The following paragraphs describe the timing profiles used to parameterize FUs for specific datapath configurations.



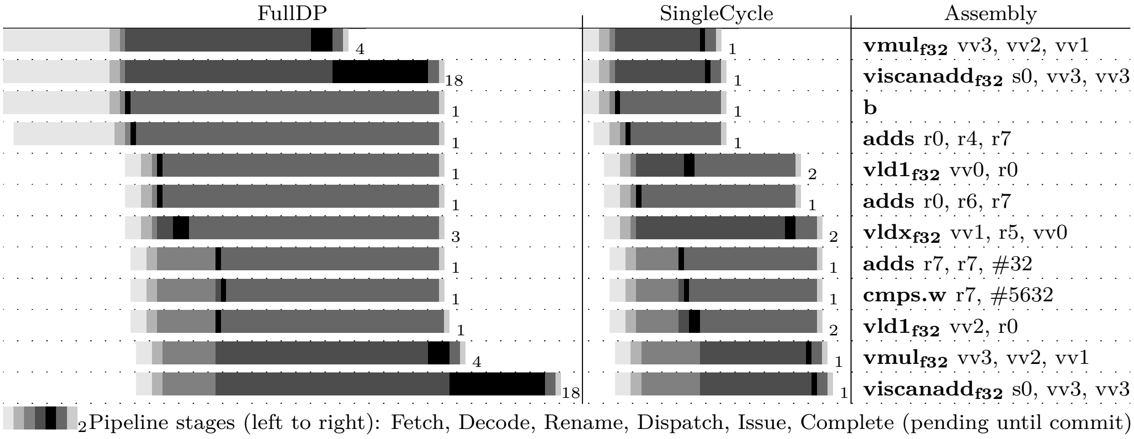
- Occupancy of individual stages represented by color (□ free, ▩ 1st part, ■ 2nd part)
- Arrows indicate dependencies between stages

(a) Single Part ALU (b) Two Part ALU (c) Single Part Scan (d) Two Part Scan

Fig. 6.10: Pipeline utilization underlying FullDP and HalfDP timing profiles

FullDP is based on a high performance ARM A-class core scaled to 128-, 256- and 512-bit wide registers. The scaling primarily affects the execution time of scans, which require $log_2(VL_{\text{max}}) + 1$ stages. **HalfDP** is derived from FullDP to estimate the impact of a datapath width of $^1/_2$ VL$_{\text{max}}$. It assumes that instructions are split into two parts. Fig. 6.10a and (b) show that pipelining both parts increases the operation and issue latency of the instruction by only one cycle. Dependencies between individual elements of scans require them to be executed in sequence (Fig. 6.5c). Fig. 6.10c depicts a scan executing in $log_2(4) = 2$ scan and 1 computation stages. The equivalent HalfDP variant in (d) requires $log_2(^4/_2) = 1$ scan and 1 computation stage per part. Both variants exhibit so called pipeline bubbles that lead to an underutilized datapath. Note that the scan and computation stages in Fig. 6.10 have been chosen to take two cycles to highlight the effect of dependencies. The single cycle latencies actually used for the FullDP and HalfDP profiles do not exhibit pipeline bubbles. Further details on the pipeline utilization of scans can be found in Section 7.3, which investigates Scan Partitioning as a method to trade of datapath width against execution speed.

Unpacked is used to estimate the impact of CPU cycles required at the begin and end of most vector instruction to route elements between their packed representation inside registers to individual datapath lanes (Section 1.4.3). It is based on FullDP, but does not incorporate the "Unpack" and "Pack" stages illustrated in Fig. 6.10. **SerialScan** considers the requirement of certain algorithms to perform FP operations in-order to avoid inaccuracies imposed by a lack of associativity. By relying fully serialized scans, it effectively represents the lower performance bound for the execution of scans. In contrast, **SingleCycle** represents the upper bound for performance gains achievable by vectorization. It idealizes operation and issue latencies for all vector instructions to one cycle.



FullDP	SingleCycle	Assembly
4	1	**vmul$_{\mathbf{f32}}$** vv3, vv2, vv1
18	1	**viscanadd$_{\mathbf{f32}}$** s0, vv3, vv3
1	1	**b**
1	1	**adds** r0, r4, r7
1	2	**vld1$_{\mathbf{f32}}$** vv0, r0
1	1	**adds** r0, r6, r7
3	2	**vldx$_{\mathbf{f32}}$** vv1, r5, vv0
1	1	**adds** r7, r7, #32
1	1	**cmps.w** r7, #5632
1	2	**vld1$_{\mathbf{f32}}$** vv2, r0
4	1	**vmul$_{\mathbf{f32}}$** vv3, vv2, vv1
18	1	**viscanadd$_{\mathbf{f32}}$** s0, vv3, vv3

2 Pipeline stages (left to right): Fetch, Decode, Rename, Dispatch, Issue, Complete (pending until commit), Commit; Index specifies issue latency

Tab. 6.2: Comparison of O3 pipeline viewer outputs over 1 iteration of SpMV-based loop for FullDP and SingleCycle timing profiles

Tab. 6.2 compares pipeline utilizations based on a loop iteration extracted from SpMV. The FullDP variant requires 105 cycles and is computation bound; i.e. the scan waits for the multiplication to complete and vice versa[2]. In contrast, the SingleCycle variant only requires 43 cycles and is memory bound; i.e. the multiplication waits for the indexed load (vldx$_{f32}$). Note that memory latencies are primarily determined by cache models rather than timing profiles. The load inside this particular iteration experiences several L1D misses. Other iterations of the same loop may exhibit fewer misses and therefore execute faster. The O3 Pipeline Viewer used here to verify timing profiles, was also employed to test the gem5 framework and identify performance bottlenecks for the algorithms analyzed in Section 6.5. It has since been released to the public gem5 repository.

[2] The dependency of multiplications on preceding scans in Tab. 6.2 is based on the merging behavior of its output register; i.e. the use of VL/VM requires the contents of vv3 to be ready prior to issue. This dependency could be mitigated by partially unrolling the underlying loop and using a different destination registers for each iteration. Alternatively, ARGON might be extended with support for zeroing; i.e. inactive elements are set to 0.

6.4.3 Baseline Configuration

Component	Parameter
Processor	single-core, out-of-order, 1 GHz clock, 40 ROB entries, 3 elem. fetch/decode and rename, 6 elem. dispatch, 8 elem. issue, FullDP timing profile, 256-bit wide regs.
L1 interface	64 TLB entries, 64 LQ entries, 64 SB entries, 32-bit address space, 4 KByte pages
L1D	32 KByte, 2 cycle latency, 64 byte lines, 2-way set-assoc., 128-bit sub-blocks per line, 2 banks, 1 rd/wt port per bank, merging enabled (m_2B_1P Section 6.4.1), physically indexed, physically tagged, 6 MSHRs (8 targets each)
L2 cache	1 MByte, 12 cycle latency, 16-way set-assoc.
DRAM	512 MByte, 30 cycle latency

Tab. 6.3: Simulation parameters of the baseline configuration

Tab. 6.3 lists key parameters of the configuration used as baseline for the analyses in Section 6.5. Whereas the evaluation of MALEC in Section 4.4.1 employs a high-performance setup optimized for scalar throughput, Tab. 6.3 describes a less aggressive configuration that relies on vector processing to accelerate specific workloads. Due to the clustering of memory references in form of vector loads/stores, this setup is even better suited to stress the L1D memory system; e.g. a single gather instruction of byte-sized elements generates up to 64 loads to fill one 512-bit wide register. Note that the current gem5 framework interprets those 64 loads as micro-ops; hence, the baseline configuration requires 64 LQ and 64 SB entries. While preliminary studies indicated no significant scalar performance impact due to this concession, the LQ&SB structures are fully associative and would therefore impose considerable energy consumptions. Furthermore, as RTL simulations are not flexible enough to accurately model the datapath components introduced for ARGON for all configurations of interest, energy estimates are considered beyond the scope of the framework described in this chapter.

6.5 Evaluation

The following sections perform quantitative analyses of scalar, NEON and ARGON implementations of VBench (Section 6.3) using the development toolchain introduces in Section 6.4. After a series of analyses employing the baseline configuration (Section 6.4.3), dependencies on the datapath configuration, functional unit timings, the memory model and input sets are investigated.

6.5.1 Baseline Configuration

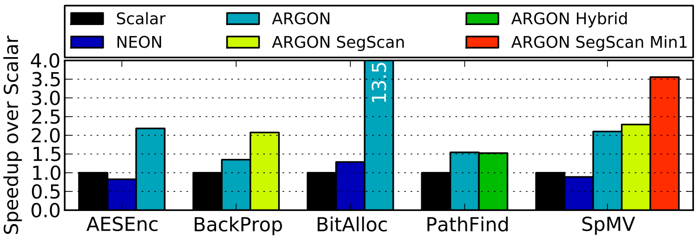


Fig. 6.11: Speedup over scalar implementations for baseline configuration

Fig. 6.11 shows speedups for the baseline configurations of the implementations introduced in Tab. 6.1 over their scalar counterparts. Benchmarks are identified by labels along the x-axis and implementations are distinguished by color. The NEON variants of AESEnc, SpMV and BitAlloc exhibit speedups of 0.8x, 0.9x and 1.3x, respectively. The limiting factor in case of AESEnc is the lack of indexed memory accesses favoring an 8- instead of a potentially faster 32-bit variant of the algorithm (Section 6.3). SpMV suffers performance penalties from having to emulate indexed memory accesses and scans using scalar instructions and fall back to scalar fix-up statements for vectors with less than VL_{max} elements (at the end of each row). While BitAlloc's NEON implementation achieves a speedup of 1.3x, it still falls significantly behind the 13.5x of the corresponding ARGON variant. This originates from the absence of per-lane predication, which forces the fall back to scalar computations to handle data-dependent operations.

	AESEnc			BackProp			BitAlloc			PathFind			SpMV				
	Scalar	*NEON*	*ARGON*	*Scalar*	*ARGON*	*SegScan*	*Scalar*	*NEON*	*ARGON*	*Scalar*	*ARGON*	*Hybrid*	*Scalar*	*NEON*	*ARGON*	*SegScan*	*SegScan Min1*
IntALU	56.9	49.4	8.5	46.2	40.6	32.3	76.5	75.8	37.5	54.7	43.9	46.2	52.8	48.6	56.2	44.0	39.6
IntMul	—	—	—	0.3	5.1	1.2	—	—	0.1	—	—	—	—	—	—	—	—
FloatALU	—	—	—	15.0	2.8	1.4	—	—	—	—	—	—	9.8	9.0	2.5	—	—
FloatMul	—	—	—	13.8	2.4	—	—	—	—	—	—	—	8.6	—	—	—	—
Load	37.0	8.7	6.0	20.9	23.2	21.7	18.6	6.1	0.9	31.1	33.1	34.6	28.2	24.1	4.7	12.3	0.3
Store	6.1	8.9	1.3	3.7	6.1	7.0	4.9	5.4	0.2	14.2	12.3	13.7	0.6	0.6	2.5	2.1	0.2
vScalar	—	—	—	—	3.9	7.3	—	10.4	16.3	—	2.8	1.5	—	15.9	5.6	14.9	21.4
vMov	—	1.2	4.1	—	2.8	4.9	—	—	9.9	—	2.1	0.9	—	—	—	3.0	4.3
vALU	—	21.3	65.5	—	0.2	1.0	—	1.9	24.5	—	2.7	0.9	—	—	—	3.0	4.3
vALUScan	—	—	—	—	—	—	—	—	2.2	—	0.9	0.9	—	—	—	—	—
vMul	—	0.9	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
vFloatALU	—	—	—	—	2.0	2.8	—	—	—	—	—	—	—	—	—	—	—
vFloatALUScan	—	—	—	—	1.6	0.2	—	—	—	—	—	—	—	—	5.4	—	—
vFloatALUSegS	—	—	—	—	—	1.4	—	—	—	—	—	—	—	—	—	3.0	4.3
vFloatMul	—	—	—	—	4.7	8.1	—	—	—	—	—	—	—	0.9	5.4	3.0	4.3
vLoad	—	9.5	14.1	—	3.5	8.3	—	0.3	6.1	—	2.1	1.2	—	0.9	17.8	11.9	17.1
vStore	—	0.1	0.5	—	1.1	2.6	—	—	2.2	—	0.2	—	—	—	—	3.0	4.3

- All ratios given in percent; "—" indicates $x = 0\%$
- Cell colors indicate [ ] $x < 1\%$, [░] $1\% \leq x < 5\%$, [▒] $5\% \leq x < 10\%$ and [▓] $10\% \leq x$

Tab. 6.4: Ratio of instruction types executed by baseline configurations

Tab. 6.4 lists the contribution of specific instruction types relative to the overall number of instructions executed by each implementation. It allows interpretations regarding the ratio between load, store and computation instructions - similarly to Fig. 3.1 - and the proportion of vectorized to scalar code. For instance, the table confirms a relatively low degree of vectorization for BitAlloc's NEON variant; i.e. less than 13% of all instructions are vectorized compared to more than 61% for the corresponding ARGON implementation. By being highly vectorized and utilizing VL_{max} for the majority of operations (Fig. 6.12b), the 13.5x speedup of the latter approaches the theoretical maximum of 16x for 16-bit operations on a 256-bit datapath. Although AESEnc exhibits an even higher degree of vectorization (≈84%), it is held back by latencies imposed by its nonlinearly dispersed memory accesses[3]. Another reason for the relatively low speedup of 2.2x is that instructions such as element wise rotations are currently considered too algorithm specific for ARGON; hence, the implementation relies on a slower combination of two shifts and one OR instead.

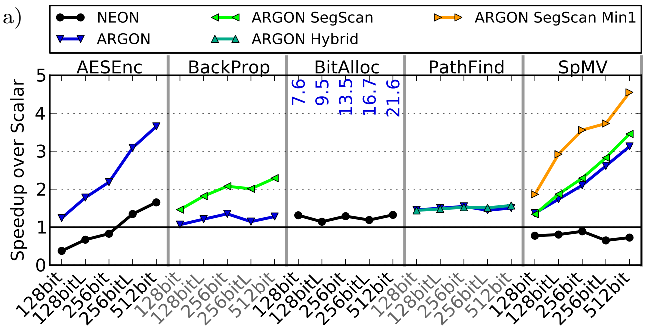
[3] Approximately 50% of read requests issued by the ARGON variant of AESEnc are stalled due to an insufficient number of available cache ports/banks. Fig. 6.9 and Fig. 6.14 illustrate the sensitivity of this implementation to the underlying L1D configuration.

The ARGON variants of BackProp and SpMV exhibit a lower degree of vectorization (20% and 34%, respectively). They also suffer from a lack of elements to be processed during each iteration of their innermost loop, which leads to an underutilization of the available datapath. As the highlighted column in Fig. 6.12b shows, the average number of elements active within those implementations is only 3.5 and 6.0, respectively. Employing segmented scans (SegScan) to effectively collapse this innermost loop, increases the achieved speedups from 1.3x to 2.1x and from 2.1x to 2.3x, respectively. Moreover, assuming a minimum of one element in each row of SpMV (Min1), allows the elimination of a conditional branch - usually required to handle empty rows -, yielding a speedup of 3.6x.

A major limiting factor for the performance of PathFind is its inherently sequential nature (<11% of instructions vectorized). The algorithm operates on a node to node basis; i.e. extract minimum from the heap, then evaluate corresponding edges one by one. Depending on the size of the heap, this might also lead to an underutilization of the datapath. For example, to allow operations on a complete 8 element vector within the decreaseKey() function, the heap would have to include $\sum_{i=0}^{7}(8^i) = 7,907,396$ nodes (just $1 + 8 = 9$ for extractMin()). This is confirmed by Fig. 6.12b, which reveals that not vecotrizing decreaseKey() for the hybrid version increases the average number of active elements from 7.4 to the theoretical maximum of 8.0. However, Fig. 6.11 shows that the vectorized version of this function still increases the overall performance of the algorithm slightly.

6.5.2 Dependency on the Datapath Configuration

a)



b)

		128bit	128bitL	256bit	256bitL	512bit
AESEnc	NEON	12.6	15.5	22.3	30.4	39.2
	ARGON	4.0	8.0	8.0	16.0	16.0
BackProp	ARGON	2.4	3.5	3.5	4.0	4.0
	SegScan	2.8	4.6	4.6	6.3	6.3
BitAlloc	NEON	8.0	16.0	16.0	32.0	32.0
	ARGON	7.1	11.0	13.9	18.5	23.5
PathFind	ARGON	3.9	7.4	7.4	14.2	14.2
	Hybrid	4.0	8.0	8.0	15.9	15.9
SpMV	NEON	4.0	8.0	8.0	16.0	16.0
	ARGON	3.4	6.0	6.0	10.4	10.4
	SegScan	3.6	7.3	7.3	14.5	14.5
	SegScan M1	3.6	7.3	7.3	14.5	14.5

Fig. 6.12: Speedup over scalar implementations (a) and number of active elements utilized (b) for different datapath configurations

To estimate the impact of the underlying datapath on the observed speedups, Fig. 6.12 shows simulation results obtained for a series of different widths and corresponding L-type variants (Section 6.2.5). The adjacent table lists the average number of active elements utilized by specific implementations with respect to the datapath. Note that a 256-bit L-type instruction implicitly refers to the same number of elements as a traditional 512-bit instruction. However, it has to split those vectors into two smaller parts and pipeline them to account for the narrower datapath. Due to the lack of VL and VM, NEON variants are limited to instructions using all available vector elements. The rational numbers given for AESEnc NEON in Fig. 6.12b, result from a mix of 8- and 32-bit based operations.

The different values for 128bitL & 256bit and 256bitL & 512bit configurations of this implementation originate from non L-type table lookups. As L-type variants of those instructions would operate on 10 registers, they are considered unfeasible with respect to NEON's 32 vector registers.

Two distinct behaviors can be observed in Fig. 6.12a; i.e. performances either independent of or proportional to the underlying datapath width. The former case indicates a low degree of vectorization, or a consistent underutilization of the datapath. In particular, due to the lack of nodes within the neural network, BackProp ARGON and ARGON SegScan saturate at 4 and 6.3 active elements, respectively (Fig. 6.12b). In contrast, other implementations exhibit linear gains from wider vectors. However, the energy cost and hardware complexity incurred by wider datapaths potentially outweigh the benefits of higher performance gains.

Less costly, because re-using narrow datapaths, the gains from L-type instructions for specific algorithms depend on their sensitivity to instruction latencies. In particular, AESEnc, BitAlloc and SpMV show significant improvements over their traditional counter parts, whereas the 256-bit L-type variant of BackProp yields reduced performance. Reason for this is a dependency on a particular FP scan. As the underlying timing profiles assume that pipelines are re-used by subsequent scan stages, they impose significant latencies to L-type operations. Furthermore, algorithms that require too many registers to be preserved between loop iterations, may suffer from long latencies introduced by vector fill/spill instructions. Note that replacing L-type instructions with two single register instructions leads to increased resource requirements and henceforth a less favorable performance/energy trade-off. In particular, the processor's fetch/decode and rename stages, as well as the associated queues, would need to be expanded to handle the increased number of instructions to be processed in parallel.

6.5.3 Dependency on Functional Unit Timings

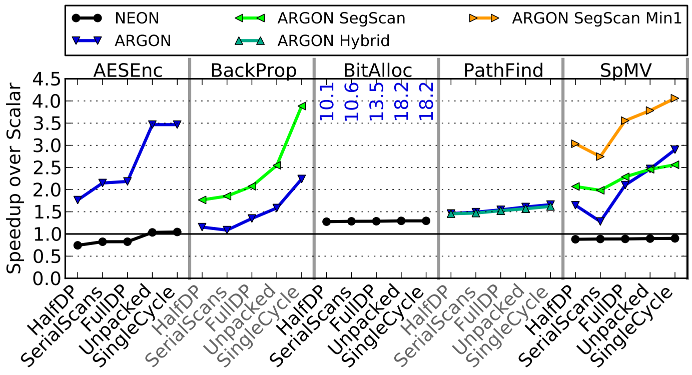


Fig. 6.13: Speedup over scalar implementations for different timing profiles

Fig. 6.13 illustrates the effects of the timing profiles introduced in Section 6.4.2 on the observed speedups. As would be expected, halving the available datapath reduces the performance of all implementations (comparing HalfDP to FullDP). However, similarly to L-type instructions, the performance does not scale linearly. Hence, ARGON might be implemented on a variety of different processors, providing a good trade-off between achievable performance and imposed hardware complexity / energy consumption.

FullDP processes scans in $log_2N + 1$ cycles (Section 6.2.3). However, certain algorithms may require FP operations to be executed in-order. SerialScan estimates the impact of the complete serialization of scans. Fig. 6.13 shows a significant performance drop particularly for SpMV, which iterates over a tight loop that is highly dependent on instruction latencies. Nevertheless, as all implementations still outperform their scalar counterparts, energy oriented processors might deliberately forfeit potential performance benefits in favor of reduced hardware complexity. High performance solutions might support a compatibility mode to control the parallelization/serialization of FP scans to address associativity concerns. Note that the current ARGON framework does not consider VL, when estimating the number of cycles required by scans. In particular, a processor may accelerate scans by avoiding those computation stages that operate exclusively on inactive elements; i.e. assuming VL=$^1/_2$VL$_{\text{max}}$, FullDP and SerialScan may save 1 and $^1/_2$VL$_{\text{max}}$-1 stages, respectively (Fig. 6.5c).

The timing profile labeled Unpacked estimates the impact of CPU cycles required at the begin and end of each vector instruction to route elements between their packed representation inside registers to individual datapath lanes. A comparison to the SingleCycle profile, which assumes one cycle latencies for all vector operations, shows that the integer based algorithms AESEnc and BitAlloc actually reach their theoretical peak performance. However, BackProp and SpMV are furthermore limited by long latency FP operations.

6.5.4 Dependency on the Memory Model

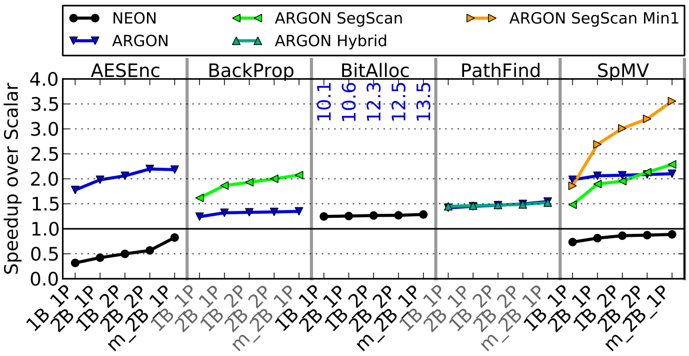
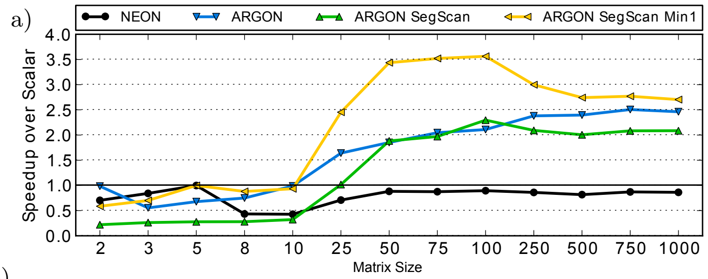


Fig. 6.14: Speedup over scalar impl. for different L1D configuration

Fig. 6.14 illustrates achievable speedups with respect to the memory models identified in Section 6.4.1. It can be observed that for the analyzed benchmarks multiple ports are generally more beneficial than multiple banks. In particular, the speedup of SpMV improves from 1.8x to 3.0x instead of 2.7x, when comparing 2B_1P against the 1B_2P setup. However, these gains have to be weighed against the consideration that multiple ports dramatically increase cache energy consumption and access latency, in contrast to banking which effectively reduces those parameters. In this context, a 2 bank - 1 port configuration serves as baseline for the analyses in this chapter. To better suit the needs of vector processing, it was extended with the ability to merge accesses to the same 128-bit sub-block. This is different from mechanisms employed by current processors supporting AVX, in so far that the merging of accesses is limited to sub-blocks rather than full cache lines [24]. At this point, the potential performance penalty of this approach (Fig. 6.9) is considered too marginal to justify the hardware complexity and energy consumption required by a multiplexer- and bus-network capable of accessing full 64-byte cache lines.

The performance impact of memory accesses can be observed when comparing the baseline configuration against 1B_1P. For instance, the ARGON variants of BackProp and SpMV exhibit a computation-bound behavior, whereas the corresponding SegScan implementations are memory-bound. This matches expectations set by the increased ratio of vector load/store to computation instructions observed in Tab. 6.4. For instance, in case of BackProp said ratio increased from 0.55_{ARGON} to 0.88_{SegScan}. Similarly, SpMV progresses from 1.66_{ARGON} to 2.51_{SegScan} and $2.51_{\mathrm{SegScan\ M1}}$. Furthermore, while latencies for vector loads/stores depend on the number of active elements, the latencies of computation instructions are vector length independent and are therefore not impacted by the increased datapath utilization exhibited by SegScan variants; i.e. 3.5 $\mathrm{elements}_{\mathrm{ARGON}}$ compared to 4.6 $\mathrm{elements}_{\mathrm{SegScan}}$ for BackProp and 6.0 $\mathrm{elements}_{\mathrm{ARGON}}$ compared to 7.3 $\mathrm{elements}_{\mathrm{SegScan}}$ and 7.3 $\mathrm{elements}_{\mathrm{SegScan\ Min1}}$ for SpMV (Fig. 6.15b). In case of low-end memory configurations, the combination of those factors allows SpMV's ARGON variant to outperform ARGON SegScan. It is noteworthy that even for the 1B_1P model all ARGON variants outperform their scalar counterparts. This implies that gains due to an increased degree of vectorizability would even benefit low end systems, which might implement single-ported caches and serialize indexed memory accesses.

6.5.5 Dependency on Input Sets



b)

SpMV	2	3	5	8	10	25	50	75	100	250	500	750	1000
NEON	—	—	—	8.0	8.0	8.0	8.0	8.0	8.0	8.0	8.0	8.0	8.0
ARGON	0.5	0.3	0.8	1.3	1.3	3.6	5.1	5.5	6.0	7.1	7.5	7.7	7.8
SegScan	1.0	1.0	3.9	5.6	6.3	6.9	7.3	7.2	7.3	7.2	7.2	7.2	7.2
SegScan Min1	1.0	1.0	3.9	5.0	6.4	7.0	7.3	7.2	7.3	7.2	7.2	7.2	7.2

Fig. 6.15: Speedup over scalar implementations (a) and number of active elements utilized (b) for SpMV (matrix sizes from 2*2 to 1000*1000)

A key concern of vector processors is underutilization. Not only does it effect the achievable performance, but also implies energy dissipated by inactive lanes. Fig. 6.15a illustrates this issue by showing speedups of SpMV over a range of matrix sizes. It can be observed that matrices smaller than 10*10 do not process sufficient elements ($10*10*15\% = 15$) to justify the overhead imposed by vectorization. For sizes between 25*25 and 100*100, segmented scans (Min1) achieve considerable speedups over conventional scans by better utilizing the available SIMD width; i.e. consistently processing at least 7.0 instead of 3.6, 5.1, 5.5 and

6.0 elements (Fig. 6.15b). However, matrices above this range provide sufficient elements per row to allow ARGON to approach full utilization of the datapath, whereas ARGON SegScan remains at approximately 7 elements. Furthermore, the computational overhead required to support segmented scans effectively outweighs their performance benefits as soon as instructions operate on less than two segments. The speedup of ARGON SegScan Min1 over ARGON even for a 1000*1000 matrix is based on the underutilization exhibited by ARGON at the end of individual rows (in average: $(1000 * 15\%)\ mod\ 8 = 6$ elements).

A direct comparison of both SegScan variants in Fig. 6.15a clearly highlights the sensitivity of the algorithm to a conditional branch introduced to handle the fix-up of empty rows. Even though it is never taken, the control dependency introduced by it, greatly impedes performance. BackProp does not include such a branch, because the number of nodes within each layer of the neural network is by definition greater than zero. The workload analyzed in the previous sections includes 3 inputs, 2 outputs and 1 hidden layer with 5 neurons (Tab. 6.1). The performance differences between the ARGON and the ARGON SegScan variant originate from the number of nodes/neurons per layer; i.e. they benefit most from neurons$_{\text{current layer}}$ = VL$_{\text{max}}$ and neurons$_{\text{current layer}}$*neurons$_{\text{previous/next layer}}$ = VL$_{\text{max}}$, respectively.

6.6 Concluding Remarks

Focus	Per-Lane Predication	Indexed Mem. Acc.	Scans	Segmented Scans	Datapath	
					Registers	FUs
Scalar	implicit VL/VM	serialized	serialized	re-use scan FU	128-bit, packed	serialized, chaining
↕	en/disable VL/VM	intra vector merging	partitioned	re-use scan FU	256-bit, packed	halfDP, partitioning
Vector	multiple explicit VL/VM	MALEC (Chapter 4)	combination of log$_2$N+1 and partitioned	compute segments in parallel using reductions	256-bit+, hybrid packed / unpacked	fullDP (Section 6.4.2)

Tab. 6.5: Implementation guidelines for advanced SIMD features (Section 6.2) with respect to focus on scalar or vectorized computation

Tab. 6.5 lists guidelines derived from the development and evaluation of ARGON and VBench in this chapter. While all features introduced in Section 6.2 justify architectural support, their microarchitectural implementation may differ between systems focusing on scalar or vectorized computation. A special case is per-lane predication, which ARGON currently supports using a single set of implicitly addressed vector length and mask registers. This architectural implementation is favorable in terms of encoding space and hardware complexity, but adversely impacts code density and performance. In particular, tight loops that include multiple assignments of VL/VM suffer from latencies and dependencies introduced by frequent updates of said registers. A compromise between scalar and vector focus could provide an opcode bit to en-/disable the use VL/VM on an instruction granularity. A vector oriented architecture may support multiple explicitly addressable VMs. Taking into account that the functionality of VL can be emulated by VM, a set of 8 to 16 masks is desirable. Combined with the ability to transfer masks into the scalar domain, they can be preserved between function calls and easily operated upon (e.g. $mask_A$ &= $mask_B$, or $mask_A$ >>= i).

The primary benefit of indexed memory accesses is the increased vectorizability of general purpose code. Even a serialized variant enables significant speedups for all analyzed algorithms (1B_1P in Fig. 6.14). More complex systems may employ multiple address computation units and mechanisms to merge vector accesses to the same 128-bit sub-block. Finally, MALEC might be employed to further reduce energy consumption (Chapter 4). Similarly, scans might be serialized to benefit from increased vectorizability, while keeping hardware complexity low and avoiding issues regarding FP associativity (Section 6.5.3). More vector oriented systems may employ Scan Partitioning as proposed in Section 7.3 to achieve specific energy-performance trade-offs. High-end systems may even implement $\log_2 N+1$ stage scans to accelerate frequently used operations such as a scan add of 32-bit integers (used for address generation). The complexity of segmented scans can be limited by reusing FUs designed for conventional scans, reinterpreting predicates as segment descriptors and appending a reduction step following the scan operation (Section 6.2.4). However, as segmented reductions only return one result per segment, the computation of all intermediate values as done by conventional scans is unnecessary. Hence, a vector focused processor might implement one or more specialized FUs to efficiently compute one or more segments in parallel.

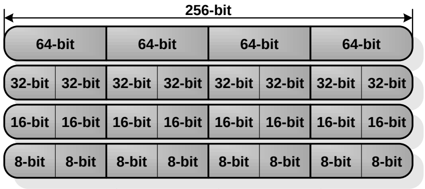


Fig. 6.16: Hybrid packing scheme (256-bit registers, 32-bit granularity)

As the energy consumption and complexity of vector registers and FUs depends on their width, scalar oriented designs are likely to be restricted to narrow registers and potentially serialized execution paths. Chaining may be employed to overlap latencies by forwarding intermediate results to subsequent instructions. Given a sufficiently high number of elements to operate on, wider registers and FUs yield substantial speedups over narrower designs (Fig. 6.12). By combining wide and narrow FUs, frequently used instructions may be sped up, while limiting the cost of more complex operations such as FP multiply and scans. Fig. 6.13 shows the significant impact of CPU cycles required to route elements between their packed representation inside registers to individual datapath lanes. As packing improves the capacity of registers and simplifies casts between different data type sizes (no padding required) it is particularly suited for processors with narrow registers. However, its negative performance impact, the additional routing network required and the increased complexity for instructions operating on multiple data types (e.g. indexed memory accesses) do not scale well with wider registers. Hence, processors may employ a hybrid packing scheme as shown in Fig. 6.16; i.e. arrange elements wider/narrower than 32-bit packed/unpacked. While this scheme does scale well, algorithms that exhibit sufficient data-level parallelism to utilize datapaths wider than 256-bit might justify the overheads associated with specialized throughput accelerators such as FPGAs or GPUs [59]. Alternatively, algorithms may incorporate multi-register operations to exploit data-level parallelism without the need for wider registers and FUs (Section 7.7).

The key considerations for vector ISA extension may be summarized to vectorizability and resource utilization. ARGON improves the former with the features introduced in Section 6.2. However, the introduction of per-lane predication, wider datapaths and L-type operations increases the probability of vector lanes being underutilized. Section 7.5 and Section 7.6 propose methods to accelerate scans on less than $\mathrm{VL}_{\mathrm{max}}$ and operations on 0 elements, respectively. Note that so called 0-length instructions are used to speed up algorithms by removing the need for conditional branches guarding code sequences with VM/VL=0. Another method to accelerate predicated vector instructions is zeroing. As discussed in the context of Tab. 6.2, the merging of inactive elements as performed by ARGON requires the destination register to be ready prior to issue. Zeroing removes this dependency without sacrificing determinism by assigning 0 to inactive elements.

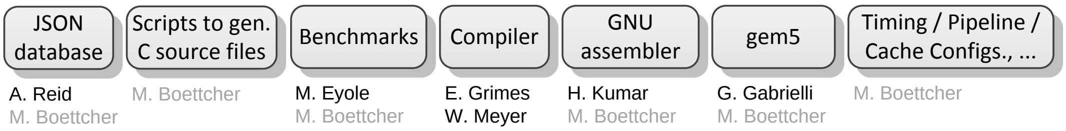


Fig. 6.17: Engineering contributions by author

The engineering contributions underlying this chapter are described in Section 6.4. To clarify the contributions made as part of this thesis, Fig. 6.17 includes a breakdown of the workloads involved in the development of the ARGON toolchain by author. The initial concept of ARGON was derived by Alastair Reid and consecutively refined and expanded for this thesis. Dr. Mbou Eyole selected the benchmarks to be investigated and provided the idea to implement some form of segmented scans. Edmund Grimes and Wojciech Meyer co-developed the compiler support for the proposed ISA. Harsh Kumar provided a basic assembler/disassembler framework for another ISA, that was expanded and ported to ARGON. Dr. Giacomo Gabrielli implemented the support for ARGON-style vector instructions into the majority of the pipeline stages of the gem5 OoO model. This model was refined and further extended for this thesis. This includes features similar to those described in the context of the cache model employed by previous sections, e.g. supporting banking, subbanking, ports, ect. A major addition are mechanisms to preserve memory references over multiple cycles in response to limited cache resources, and to perform intra-vector merging as described in Section 6.4.1. The timing profiles, pipeline and cache configurations were derived from existing high performance processors and the concepts described in the next chapter.

7 | Microarchitecture Optimizations for Energy Efficient SIMD Datapaths

7.1 Introduction / Overview

The previous chapter investigated the performance impact of advanced SIMD features that were previously exclusive to HPCs and only recently considered for commercial microprocessors [24]. This chapter addresses several aspects that arose during the implementation of those features into the ARGON simulator framework and the generation of the analyzed timing profiles. For instance, per-lane predication introduces the possibility of operations on arbitrary number of elements (Section 6.2.1). While this increases the vectorizability of general purpose code, it also opens up the potential of datapath underutilization due to operations on vectors partially or even exclusively composed of inactive elements. Similarly, segmented scans allows the efficient vectorization of nested loops with limited data-level parallelism, but require some form of translation mechanism between software and hardware comprehensible descriptor formats (Section 6.2.4). Finally, the hardware and energy costs associated with some of the analyzed features impose a significant hurdle for their applicability in the context of energy constrained systems. Hence, more flexible implementation variants that allow specific trade-offs between performance and energy consumption are desirable. In this context, the utilization of narrow datapaths in combination with wide registers allows low- to mid-range systems to exploit data-level parallelism for moderate hardware costs. The following list gives are brief overview of the microarchitecture optimizations described in the subsequent sections:

1) **The Compute Descriptor Instruction (Section 7.2):**
    - Segmented scans are a derivative of scans that allow arbitrary length segments within vectors to be processed in parallel. Algorithms commonly describe segments as arrays of tail-pointers.
    - The compute descriptor instruction facilitates means to translate those pointers into vector masks that may be used by FUs to control the interaction between individual lanes. It is designed to provide a high degree of flexibility in its application, while being simple enough to be realized with fast and energy efficient circuitry.
2) **Scan Partitioning (Section 7.3):**
    - Scan primitives perform cumulative operations over sequences of vector elements. The hardware complexity associated with them increases with the number of elements to be operated upon; i.e. higher number of FUs and more complex routing networks.
    - Scan Partitioning gives designers flexibility in how many resources may be allocated to scans to achieve a specific performance/energy trade-off. In particular, scans may be executed in two to N/2 parts to achieve the same results with a

decreasing number of operations and narrower datapaths. The performance of the described method dependents on the underlying pipeline structure; i.e. certain configuration can be faster or as fast as single-part implementations, while others trade performance against higher energy efficiency.

3) **Efficient Handling of Masks (Section 7.4):**
    - Per-lane predication introduces a requirement for circuitry to implement merging/zeroing behavior for inactive vector elements based on predicate flags. Similarly, segmented scans require mechanisms to forward elements between individual lanes in order to delineate segments.
    - Instead of implementing dedicated hardware to realize these functionalities, existing multiplexing circuitry - previously only used to select between two elements for comparison results or insert zeros for saturating subtractions - may be reused. The proposed method reduces area and energy consumption without imposing additional latencies into critical execution paths.
4) **Acceleration of Scans on incomplete Vectors (Section 7.5):**
    - While scans greatly increase the vectorizability of general purpose code, the data dependencies between individual operations imposes a need for multiple consecutive computation steps to implement them. Hence, they are often associated with long latencies. Per-lane predication further increases the applicability of scans to code segments with limited data-level parallelism, but introduces the possibility of datapath underutilization due to operations on a limited number of elements.
    - Conventional scan processing schemes allow the omission of a full computation step in case of VL $\leq$ 1/2 VL_{max}. The proposed scheme increases the probability of being able to accelerate incomplete vectors by improving this condition to VL $\leq$ VL_{max} - 1. It allows similar savings for segmented scans comprising two or more segments of length VL_{max} - 1 instead of specifically requiring a segment boundary separating the upper and lower half of the input vector.
5) **Efficient Handling of Zero Length Vector Micro-Ops (Section 7.6):**
    - Per-lane predication allows the number of elements operated upon to be determined data dependently at run time. However, certain algorithmic constructions introduce the possibility of instructions exclusively operation on inactive elements. For example, a vector mask may be determined based on a comparison between data elements and a threshold value. In case of all values falling outside the region of interest, the subsequent vector instructions are of length zero.
    - A common way to avoid the energy and performance impact associated with those instructions is their enclosure within a guarding branch; i.e. skip their execution if VL = 0. However, this approach relies on energy intensive branch predictors and the regularity of vectorized code segments to exploit potential performance benefits. The proposed optimization focuses on software transparent modifications to dispatch and or issue circuitry. In particular, it substitutes micro-ops depending on their types either with NOPs or vector moves. Meant to

be used in conjunction with guarding branches, it does not impose any overhead in form of additional instructions or miss-speculation penalties, while operation on a very fine granularity.

6) **Multi-Register Operations (Section 7.7):**

- While the implementation of wider datapaths allows further acceleration of vectorized code segments, it also increases the costs associated with the underutilization of the corresponding vector register file. In particular, the development of the Vector Benchmark Suite revealed that the analyzed benchmarks utilize less than half of the available vector register bank. When considering the size of this structure - e.g. 1 KByte for ARMv7 NEON, or 16 KByte for Intel's AVX-512 [68, 24] - this implies a significant amount of leakage energy and silicon area consumed by dormant registers.
- Multi-register operations attempt to mitigate this issue by allowing operations on logical registers comprised of several physical register. Corresponding instructions are then split into a number of micro-ops which are then executed in pipelined or – if redundant FUs are available – parallel fashion on the same datapath employed by conventional operations. Besides improved utilization, these operations offer performance benefits originating from higher code density and a reduced pressure on fetch and decode circuitry.

The following sections relate to the microarchitecture optimizations listed above (Sections 7.2 to 7.7). Individual sections are comprise a summary of the proposed optimization and the motivation behind it, a brief background on the surrounding subject matter, detailed explanations on the optimization itself and considerations regarding its interaction with other features introduced here or in the context of ARGON. Each section as well as this chapter concludes with a set of remarks. The remarks in Section 7.8 regard the influence of the proposed optimizations on ARGON and the current limitation of the employed simulation framework.

7.2 The Compute Descriptor Instruction: Improving the Utilization of SIMD Datapaths by Collapsing Nested Loops

7.2.1 Motivation and Overview

Segmented scans allow the parallel execution of multiple independent scans within a single vector. Each scan performs cumulative operations over a sequence of elements within its corresponding segment (Section 6.2.4). Chapter 6 investigated segmented scans in the context of the BackProp and SpMV algorithm, and showed that their application increased speedups achieved by vectorized over scalar implementations from 1.3x and 2.1x to 2.1x and 3.6x, respectively (Fig. 6.11). The reason for this improvement is the improved datapath utilization achieved by the simultaneous computation of multiple segments; i.e. the average number of active elements increased from 3.5 and 6.0 to 4.6 and 7.3 (Fig. 6.12b). The key challenges for the efficient computation of segmented operations are:

1) Gather inputs values corresponding to multiple segments into one vector register
2) Describe segment boundaries comprehensible to FUs
3) Perform segmented operations within one or more FUs
4) Scatter outputs

The data movement required to facilitate segmented operations relies on indexed memory accesses as introduced in Section 1.4.3, supported by ARGON (Section 6.2.2), and efficiently implemented by MALEC (Chapter 4). Common ways to describe segment boundaries in software are arrays of pointers indicating the start/end of segments within a sequence of elements, or arrays of segment lengths; e.g. the array [0,5,10,15] may represent head-pointers describing segments including elements 0-4, 5-9, 10-14 and 15 onwards. In contrast, a SIMD datapath interprets segments as controlled interactions between lanes. For instance, a FU may use a vector mask to isolate segments by preventing intermediate results from propagating between lanes during a scan operation (Fig. 7.2b, Section 7.4). The Compute Descriptor Instruction (CompD) - originally envisioned by Dr. Mbou Eyole (ARM) and refined, simplified and realized here - bridges the gap between software and hardware oriented descriptor representations by generating a mask from an arrays of pointers. It is capable of handling segments of arbitrary length; e.g. zero, less than $\mathrm{VL}_{\mathrm{max}}$ or more than $\mathrm{VL}_{\mathrm{max}}$ elements.

The following sections give background information on segmented operations (Section 7.2.2), introduce the CompD instruction - including underlying data formats and in-/outputs - (Section 7.2.3), propose a potential circuit design for it (Section 7.2.4), and demonstrate its operation in the context of the SpMV algorithm (Section 7.2.5), before concluding in Section 7.2.6.

7.2.2 Background

The concept of segmented scans was fist proposed by Schwartz [96] in the context of a theoretical many-core processor and later extended and generalized by Blelloch [97] for existing vector HPCs. Recent attempts at solving irregular computation problems focus on throughput accelerators such as GPUs [98]. While throughput-oriented processors excel at overlapping computation and memory access latencies, they struggle to efficiently balance irregular workloads between individual processing units. The spatial locality of SIMD lanes within modern microprocessors, mitigates issues associated with load balancing by simplifying interactions between lanes.

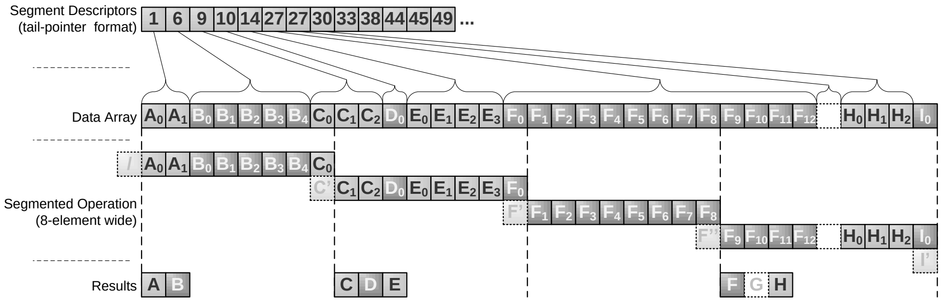
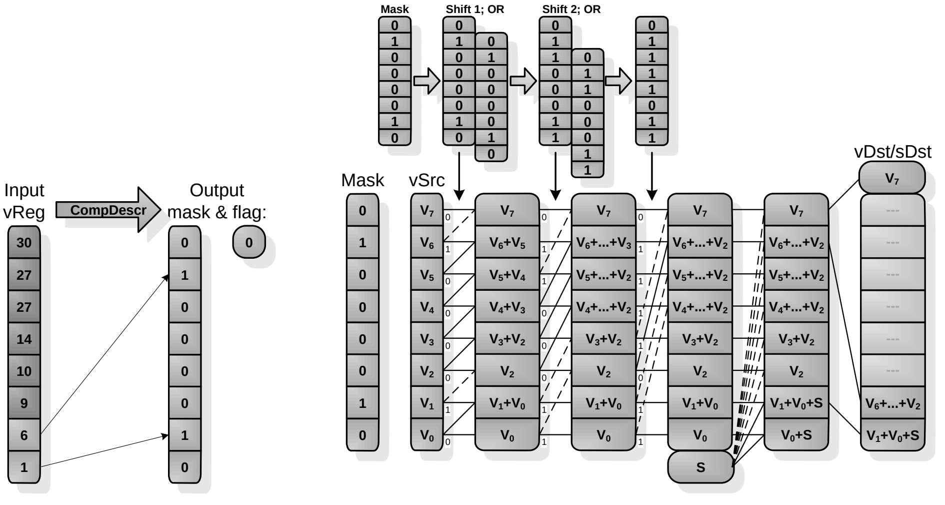


Fig. 7.1: Segmented operation using array of tail-pointers to delineate segments

Fig. 7.1 illustrates an example for a sequence of segmented operations using an array of pointers to delineate segments. Each individual operation is performed on 8-element wide vectors ($\mathrm{VL}_{\max}$). As segments may not be aligned to vector boundaries or span multiple vectors, segmented operations require scalar in- and output operands to propagate intermediate results. Furthermore, segments of length zero may require mechanisms to generate default values and insert them into corresponding result vectors; e.g. segment G inserted as the result of two equivalent pointers to element index 27 (Fig. 7.1).

7.2.3 The Basic Idea



(a) CompD instruction (b) Segmented Scan Add followed by Reduction

Fig. 7.2: Examples of segmented operations

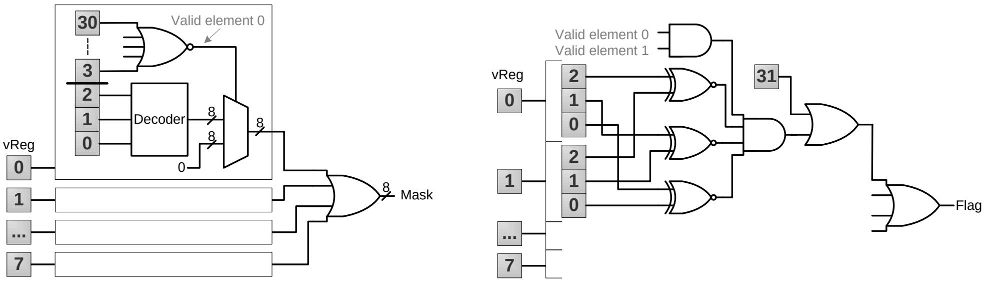
The CompD instruction implemented by ARGON generates masks in binary format based on a vector of 32-bit signed integers as input. It requires the input to be specified as an array of tail-pointers, which it decodes into mask bits describing the last element of each segment (Fig. 7.2a). Pointers exceeding $\mathrm{VL}_{\mathrm{max}}$-1 are ignored and negative values as well as two or more identical pointers are interpreted as zero-length segments indicated by raising a flag. To avoid the need for fully associative comparisons, the detection of zero-length segments is limited to monotonic in-/decreasing inputs; i.e. only consecutive pointers are compared for equivalence. The tail-pointer format simplifies the decoding hardware employed by CompD (Section 7.2.4). Head-pointers may be converted by skipping the first and subtracting 1 from all subsequent pointers; e.g. [0,2,7,10] ⇒ [1,6,9,..]. Similarly, segment length-based descriptors may be converted using scan add instructions and -1 as initial carry input; e.g. [2,5,3,1] ⇒ [1,6,9,10].

Fig. 7.2b illustrates use of the mask generated in (a) for a segmented reduction. The example is based on the FullDP timing profile used as baseline for ARGON (Section 6.4.2); i.e. it reuses an FU designed for segmented scans and performs a subsequent reduction as part of the packing stage. It can be observed that mask bits determine which operands are used to compute results corresponding to specific lanes. The details on the underlying control/bypass circuitry are described in Section 7.4. The tail-flag format simplifies masks used in subsequent scan rounds to a shift by $\log_2$R elements and an OR-operation (R.. index of current scan round: 1, 2, etc.). The mask of the final computation step is determined slightly different, because the scalar input only affects the first segment. Alternative formats such as front-flag or alternating-flag would be possible, but complicate the handling of segments that are not aligned to vector boundaries. For instance, the result of a 1-element segment described by a 1 at the most significant bit of a front-flag formatted mask register would be returned in the form of a carry output and not as part of the result vector. An alternating-flag formatted mask would require some form of information regarding the last bit of the mask used for a preceding segmented operation; e.g. the sequence [1/0]“11110000” includes two/three segments. Note that while binary masks are easily comprehensible to FUs, they cannot describe zero-length segments. Hence, corresponding segmented operations may require scalar fix-up statements to insert predetermined values for such segments. In summary, CompD is composed of the following operands:

- Input: 1 vector register
  - Array of signed 32-bit tail-pointers
  - Values $>= VL_{max}$ are ignored
  - Negative values are interpreted as zero-length segments
  - The order of the inputs is irrelevant; however, the limitation to monotonic in-/decreasing inputs simplifies detection of zero-length segments
- Output: 1 mask register
  - Mask in tail-flag format
  - Might be stored in dedicated mask register (VM), inside a conventional vector register, or a general purpose (scalar) register

- Output: 1 flag
  - Identifies zero length segments
  - ARGON reuses the carry (C) flag; it is currently considered unnecessary to introduce a dedicated flag

7.2.4 Circuit Design



(a)

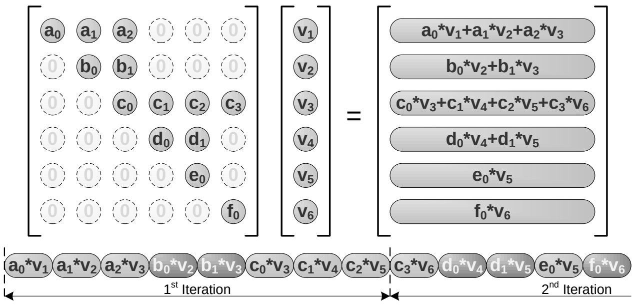
(b)

Fig. 7.3: Circuit diagrams used to compute output mask (a) and flag (b) for CompD

Fig. 7.3 illustrates how CompD might be implemented for 8-element vectors. Individual mask bits are generate by decoding the three least significant bits of each input value. The remaining bits are ORed to identify invalid elements ($x < 0 \mid\mid x \geq \mathrm{VL}_{\mathrm{max}}$). The masks for those invalid elements get zeroed using an AND operation. All remaining masks are then ORed to obtain the final output. At the same time, the three least significant bits of adjacent elements are compared against each other and – assuming that both elements are valid – ORed with the corresponding sign-bit to potentially raise the output flag. The restriction to adjacent elements for this step simplifies the underlying circuitry but restricts the detection of zero-length segments within CompD to monotonic in-/decreasing input vectors. A more generic design may implement fully associative comparators. However, this would lead to an increased energy consumption and more difficulties in meeting latency driven timing constrains; i.e. the FullDP timing profile used as baseline for ARGON conservatively assumes a 3 cycle operation latency for CompD (Section 6.4.2). Note that both circuit diagrams do not include circuitry to support per-lane predication, as it might be used to limit the number of input elements considered by the instruction.

7.2.5 Algorithmic example

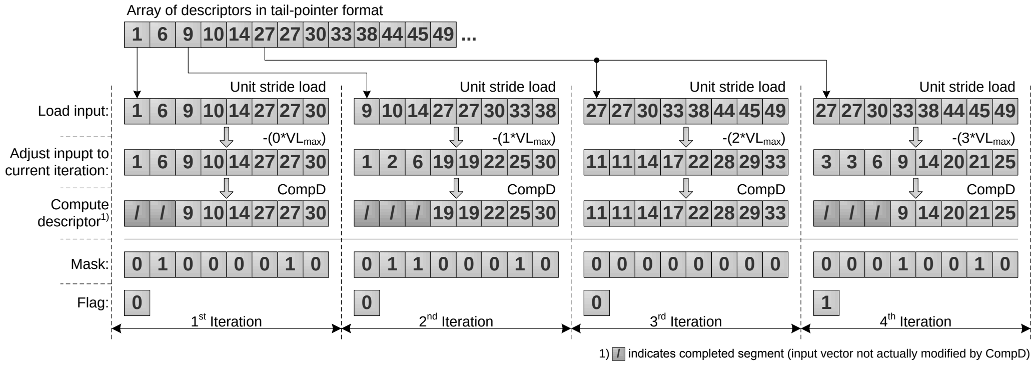
Fig. 7.4 shows an example of a sparse matrix-vector multiplication including the arrangement of intermediate products within 8-element vectors prior to a reduction operation and a summary of the instructions executed during each iteration of the underlying loop. The whole matrix can be computed using only two segmented reductions (VL=5 for the second iteration). An equivalent unsegmented variant would require 6 reductions instead. Note that the described instruction sequence performs several redundant memory accesses, as will be shown in the examples below. Heavily memory constrained systems may employ sequences that only load further descriptors if the number of elements represented by previously not completed segments is less than $\mathrm{VL}_{\mathrm{max}}$.


a_0 a_1 a_2 0 0 0
0 b_0 b_1 0 0 0
0 0 c_0 c_1 c_2 c_3
0 0 0 d_0 d_1 0
0 0 0 0 e_0 0
0 0 0 0 0 f_0
v_1 v_2 v_3 v_4 v_5 v_6
=
$a_0*v_1+a_1*v_2+a_2*v_3$
$b_0*v_2+b_1*v_3$
$c_0*v_3+c_1*v_4+c_2*v_5+c_3*v_6$
$d_0*v_4+d_1*v_5$
e_0*v_5
f_0*v_6
a_0*v_1 a_1*v_2 a_2*v_3 b_0*v_2 b_1*v_3 c_0*v_3 c_1*v_4 c_2*v_5 c_3*v_6 d_0*v_4 d_1*v_5 e_0*v_5 f_0*v_6
1st Iteration
2nd Iteration

- Load data and perform multiplications
- Load descriptors
- Decrements descriptors by number of elements already processed ($\text{loop}_{\text{index}}$ * VL_{max})
- CompD
- Perform segmented reduction
- If carry flag set:
  - Insert 0s for zero-length segments
- Store results
- Increment data pointer by number of computed elements (VL_{max})
- Increment descriptor pointer by number of completed segments (number of ones in mask + number of zero-lenght segments)

Fig. 7.4: Exemplary working set of SpMV, arrangement of intermediate products inside 8-element vectors, and summary of instructions performed during each iteration

Fig. 7.5 illustrates a sequence of processing steps related to CompD based on four iterations of the instruction sequence given in Fig. 7.4. Note that the array of tail-pointers listed in the former does not correspond to the matrix given in the later figure. A matrix of that size is considered infeasible to be displayed here. During each iteration, a vector of tail-pointers is loaded, decremented by the number of previously computed data elements, and fed into CompD. The unit stride load of the next vector of tail-pointers is offset by the number of segments completed within the previous iteration (number of ones in corresponding mask). In particular, iteration 3 does not complete any segments; it performs a reduction on the carry from iteration 2 and an 8-element data vector before forwarding the intermediate result as carry out. Iteration 4 then loads the same vector of tail-pointers to complete the segment in question. The presence of a zero-length segment represented by two equivalent pointers in the range of $0 \leq x < \text{VL}_{\text{max}}$ is indicated by a raised flag. In case of SpMV, zero-length segments are handled by inserting a zero into the corresponding position of the result vector.



1) / indicates completed segment (input vector not actually modified by CompD)

Fig. 7.5: Sequence of CompD over several loop iterations

Fig. 7.6 illustrates the need for CompD to operate on signed values. The first iteration completes VL_{max} 1-element segments. Consequently, the unit stride load at the begin of iteration 2 returns a vector of completely new pointers. Hence, after decrementing those pointers by the number of previously computed segments, zero-length segments at the begin the vector are indicated by negative values. An instruction operating on unsigned values would not be able to distinguish a zero-length (-1) from a very long segment (2^{32}-1).

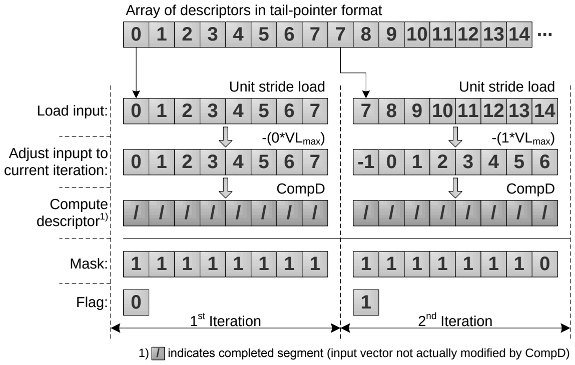


Fig. 7.6: Corner case requiring interpretation of negative descriptors

7.2.6 Concluding Remarks

This section introduced the CompD instruction used by ARGON to compute descriptors from arrays of tail-pointers into masks of tail-flags, which are easily comprehensible to FUs. Section 7.2.3 showed how other descriptor formats (e.g. head-pointers or segment lengths) can be adapted to CompD using basic operations such as add or scan add. While alternative masks formats may be employed, tail-flags simplify the handling of multiple iterations of segmented instructions by clearly identifying when intermediate values need to be propagated further or can be written-back as results. The instruction uses a flag to indicate the presence or zero-length segments, which may then be handled by corresponding fix-up statements. It detects those segments by identifying negative pointers or groups of two or more equivalent elements. By limiting this detection to groups of consecutive elements, the hardware associated with CompD can be implemented very efficiently in terms of energy consumption and latency. Consequently, the 3 cycle latencies assumed by ARGON's baseline timing model represent a rather conservative estimate.

7.3 Scan Partitioning: Achieving Adjustable Energy - Performance Trade-offs for Scan Operations

7.3.1 Motivation and Overview

Scans are primitives that perform cumulative operations over sequences of vector elements (Section 6.2.3). Scan Partitioning allows designers to decide how much hardware resources a system may dedicate to scans to achieve a specific performance/energy trade-off. In particular, executing scans in exactly two parts – each comprised of half the number of elements per vector - reduces the number of operations required and halves the width of FUs used. By modifying the underlying pipeline structure, the resulting performance can be adjusted to fall short of, match, or exceed single-part variants. Alternatively, an energy oriented system may execute N-element scans in N/2 parts. This achieves 2x speed-up over serialized scan implementations, while requiring only 50% more operations.

The following sections briefly describe the scan paradigm and related terminology (Section 7.3.2), introduce the idea of Scan Partitioning (Section 7.3.3), describe execution pipeline configurations that may be employed by partitioned scans (Section 7.3.4), demonstrate their utilization (Section 7.3.5), and the effects of per-lane predication (Section 7.3.6), before concluding in Section 7.3.7.

7.3.2 Background

The efficient execution of scans - also called prefix-sums - has been extensively studied in the context of vector super computers. However, as those systems traditionally operate on very long vectors but narrow SIMD datapaths, related studies usually focus on algorithmic instead of hardware optimizations [99]. Scan Partitioning is closer related to publications in the field of hardware adders. Knowles presents an overview on sequences describing the efficient computation of carry-propagation adds [89]. The author shows that the underlying structures can be designed - in terms of the amount of internal wiring and the fanout of individual nodes - to achieve specific performance/energy trade-offs. As those adders use binary operations and compute all intermediate bits, the corresponding computation sequences can be generalized to scan operations, including scan add, scan multiply, scan min/max, etc. Scan Partitioning specifically targets sequences based on regular patterns to allow the execution of separate parts on the same hardware and simplify routing networks. It furthermore considers pipelining as a means to reduce the complexity of scan FUs, while achieving given performance targets.

Fig. 7.7 illustrates an 8-element variant of the scan add scheme previously introduced in Section 6.2.3. It represents one example for the schemes targeted by Scan Partitioning and will be used as baseline throughout this chapter. Note that the computation of the scalar carry input during the last, rather than the first or an intermediate step, increases the number of operation involved, i.e. 4 instead of 1 when comparing Fig. 6.5c and (b).

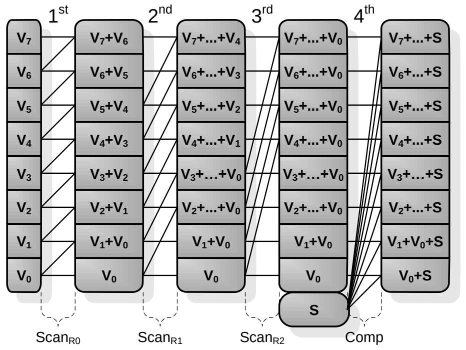

1st
2nd
3rd
4th
V_7 V_6 V_5 V_4 V_3 V_2 V_1 V_0
V_7+V_6 V_6+V_5 V_5+V_4 V_4+V_3 V_3+V_2 V_2+V_1 V_1+V_0 V_0
$V_7+\ldots+V_4$ $V_6+\ldots+V_3$ $V_5+\ldots+V_2$ $V_4+\ldots+V_1$ $V_3+\ldots+V_0$ $V_2+\ldots+V_0$ V_1+V_0 V_0
$V_7+\ldots+V_0$ $V_6+\ldots+V_0$ $V_5+\ldots+V_0$ $V_4+\ldots+V_0$ $V_3+\ldots+V_0$ $V_2+\ldots+V_0$ V_1+V_0 V_0
S
$V_7+\ldots+S$ $V_6+\ldots+S$ $V_5+\ldots+S$ $V_4+\ldots+S$ $V_3+\ldots+S$ $V_2+\ldots+S$ V_1+V_0+S V_0+S
Scan_{R0}
Scan_{R1}
Scan_{R2}
Comp

Fig. 7.7: Single-part vector scan add

However, this also moves the dependency associated with the carry input to the end of the scan, allowing separate parts to be pipelined more efficiently (Section 7.3.3). The terminology used in this chapter includes:

- **(Scan) Part:** Subset of consecutive vector elements to be processed concurrently; i.e. an 8-element scan may be processed in two 4-element parts
- **(Pipeline) Stage:** Unity of combinational logic and registers that operate on values and preserve them between CPU cycles (Section 1.3)
- **(Scan) Round:** Subset of operations performed in parallel as part of a scan sequences; e.g. Scan_{R0}, Scan_{R1} and Scan_{R2} in Fig. 7.7
- **(Processing) Step:** Subset of operations performed in parallel as part of a computation sequence; e.g. three scan rounds and one computation step in Fig. 7.7

7.3.3 The Basic Idea

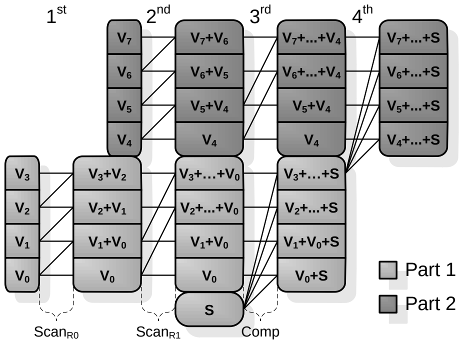
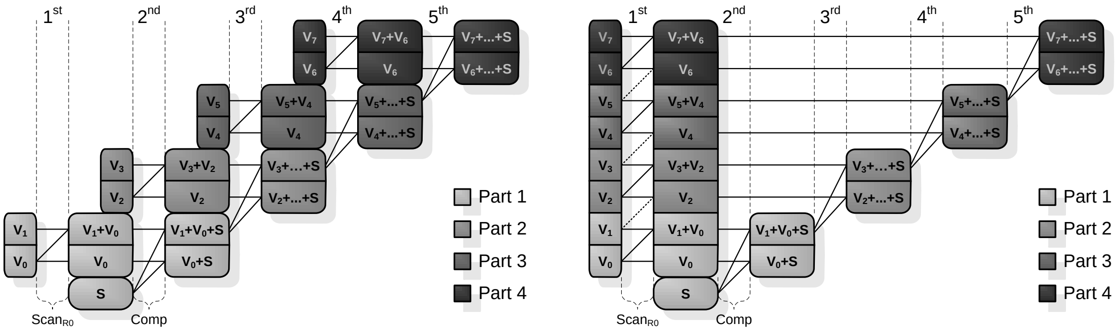

1st
2nd
3rd
4th
V_7 V_6 V_5 V_4
V_7+V_6 V_6+V_5 V_5+V_4 V_4
$V_7+\ldots+V_4$ $V_6+\ldots+V_4$ V_5+V_4 V_4
$V_7+\ldots+S$ $V_6+\ldots+S$ $V_5+\ldots+S$ $V_4+\ldots+S$
V_3 V_2 V_1 V_0
V_3+V_2 V_2+V_1 V_1+V_0 V_0
$V_3+\ldots+V_0$ $V_2+\ldots+V_0$ V_1+V_0 V_0
S
$V_3+\ldots+S$ $V_2+\ldots+S$ V_1+V_0+S V_0+S
Part 1
Part 2
Scan_{R0}
Scan_{R1}
Comp

Fig. 7.8: Two-part scan add

Fig. 7.8 represents an 8-element scan partitioned into two 4-element parts. The underlying FU can be more energy efficient than its equivalent for single-part scans (Fig. 7.7), because it operates only on half the datapath width. Furthermore, by pipelining both parts, the scheme does not increase the number of computation steps required (4), while reducing the number of binary operations performed ($25 \Rightarrow 18$); i.e. elements can more frequently bypass energy intensive computation units such as FP multipliers. Note that to obtain the same results from single- and multi-part implementations the underlying binary operations have to be associative. This is not the case for most FP operations, which can complicate the verification of vectorized FP code on differing platforms. To remedy this, programmers are encouraged to use working sets including values within well defined boundaries, when intending to verify FP code.



(a) Homogeneous Steps

(b) Heterogeneous Steps

Fig. 7.9: N/2-part scan add

The implementation illustrated in Fig. 7.8 may be further partitioned into a 4-part scan. Generally, Scan Partitioning covers all scans with 2 to N/2 parts, while excluding the trivial cases of one and N parts (Fig. 7.7 and Fig. 6.5a, respectively). Fig. 7.9 illustrates two ways an N/2-part scan may be processed. Both require approximately half the number of steps of a fully serialized variant (8 ⇒ 5), while increasing the number of binary operations performed by only 50% (8 ⇒ 12). The homogeneous variant performs the same three operations on every pair of vector elements. Hence, the routing paths between individual steps can be hardwired into the underlying scan FU to avoid the need for complex multiplexer networks. The heterogeneous variant (Fig. 7.9b) begins by performing a single vector instruction comprising all lanes. As the generation of the second operand only requires a 1 element shift, an existing vector ALU may be reused for this step. Given some form of temporary storage for the intermediate results, the subsequently used scan FUs can be further simplified.

	Number of Parts					
N	1	2	4	8	16	32
2	2	—	—	—	—	—
4	3	3	—	—	—	—
8	4	4	5	—	—	—
16	5	5	6	9	—	—
32	6	6	7	10	17	—
64	7	7	8	11	18	33

(a) Number of computation Steps required

	Number of Parts					
N	1	2	4	8	16	32
2	3	—	—	—	—	—
4	9	6	—	—	—	—
8	25	18	12	—	—	—
16	65	50	36	24	—	—
32	161	130	100	72	48	—
64	385	322	260	200	144	96

(b) Number of Operations required

Implementation variants:

- [□] N-part (serialized)
- [░] single-part
- [▒] 2-part
- [▓] N/2-part
- [■] overlap of N/2-part with others

Tab. 7.1: Number of computation steps and operations required by partitioned scans relative to the number of elements per vector (N)

The partitioning granularity of a specific scan may be chosen based on its frequency of use and/or the complexity of the underlying operation; e.g. 32-bit operations are usually more common than 8-bit operations and an integer add is less complex than a FP multiplication. Tab. 7.1 lists examples for the step and operation counts of scans with respect to the number of parts (P) and elements per vector (N). The number of steps required may be generalized to $log_2\,({}^{N}\!/_{P}) + P$. Note that this parameter is not necessarily equivalent to the number of CPU cycles spend for the execution of the corresponding scan (Section 7.3.4).

The number of operations required can be expressed as $N + P \cdot \sum_{i=0}^{log_2(N/P)-1} \left(N/P - 2^i\right)$. When comparing a single-part against a serialized scan in conjunction with both tables presented in Tab. 7.1, it becomes apparent how a reduced step count correlates with a drastically increased number of operations, and why an adjustable performance/energy trade-off is desirable. The previously discussed 2-part and N/2-part variants represent the lower and upper bounds of partitioning and might be employed as follows:

- **2-Part Scans:**
  - On high-end cores, to achieve high performance for less energy than single-part variants (reduced operation count, narrower FUs)
  - On all core types, to accelerate frequently used scans that are based on relatively simple operations; e.g. 32-bit integer scan add, as used for address computations
- **N/2-Part Scans:**
  - On high-end cores, to reduce the energy consumption and implementation complexity of rarely use scans and/or those based on complex operations; e.g. double precision FP multiplications
  - On low to mid-end cores to achieve speedups over serialized implementations, while respecting given area and energy constrains

7.3.4 Pipeline Layout

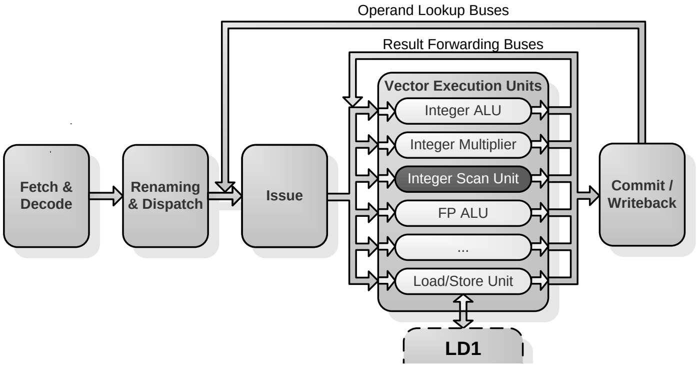
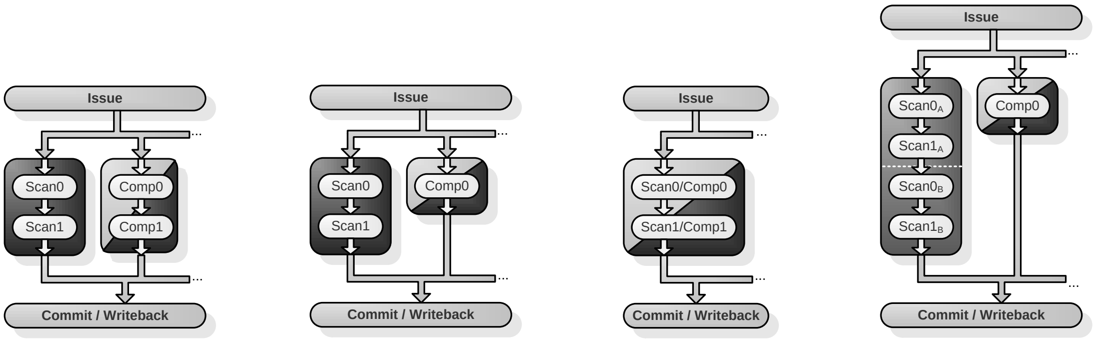


Fig. 7.10: Extract of processor pipeline focused on vector execution units (based on Fig. 1.9)

The performance of partitioned scans relative to single-part and serialized variants depends on the pipeline structure employed for their execution. This section describes different pipeline configuration, which may be employed to realize particular performance/energy trade-offs further described in Section 7.3.5. Fig. 7.10 illustrates an extract of an OoO processor pipeline with emphasis on the vector execution units (Section 1.3). A dedicated Integer Scan Unit as highlighted in the figure is only one potential implementation variant. An alternative would be to modify existing FUs to support scan operations.

Fig. 7.11 shows examples for execution pipelines capable of processing the round and computations steps forming a scan. The terms separate and merged refer to implementations with two or more individual FUs or a single shared FU to process rounds and computation steps. Furthermore, a pipeline configuration is described as being balanced / unbalanced if



(a) Balanced/Separate (b) Unbalanced/Separate (c) Merged (d) Partially unrolled

• Underlying datapath width: [□] N, [▨] 2N/P, or [■] N/P elements

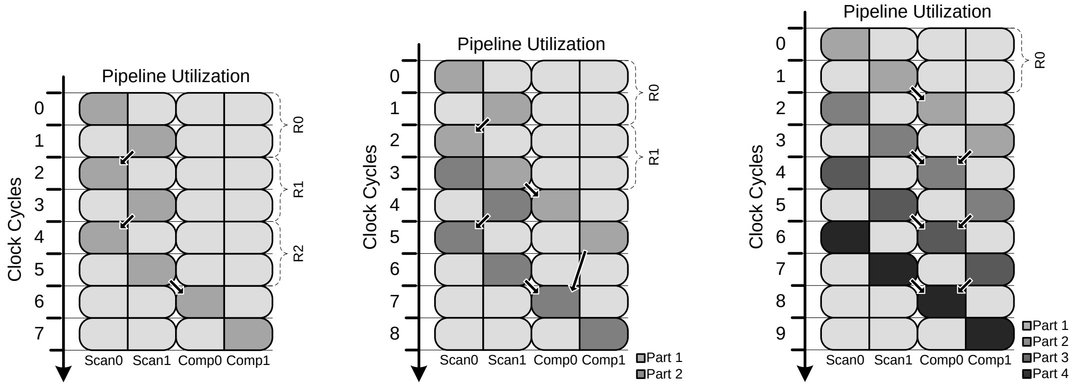
Fig. 7.11: Exemplary execution pipeline configurations analyzed in Section 7.3.5

the number of stages within a round is equivalent to / differs from a computation step. Balanced pipelines can be the result of reusing an existing FU (e.g. a vector ALU) for all steps of a scan, or implementing similar circuitry for round and computation steps (Fig. 7.11a and Fig. 7.11c, respectively). In contrast, designers may implement slower but more efficient scan FUs to reduce hardware complexity and/or increase their suitability for pipelining. In particular, partitioned scans offer performance benefits to FUs that support pipelining in round steps, while not requiring this feature for computation steps (Section 7.3.5). Fig. 7.11 uses shading to distinguish the datapath width of components. For instance, scan partitioning reduces the number of elements required within scan units to N/P; e.g. 4 elements for an (N=8)-element scan with (P=2)-parts. Bicolored components indicate the possibility of either having a dedicated (narrow) unit to process computation steps, or reuse an existing FU such as an N-element wide ALU.

Note that the computation of individual rounds requires all scan stages; e.g. Scan0 and Scan1. To avoid the replication of computational resources such as adders for each round, a routing network may be employed to allow a single set of resources to be reused. While this does not affect the performance of single-part scans, it can limit the degree of pipelining applicable to partitioned scans (Section 7.3.5). Hence, for certain configurations it can be beneficial to partially unroll scans; i.e. replicate scan stages either inside two separate FUs or as consecutive parts of the same FU (Fig. 7.11d). The diagrams in Fig. 7.11 do not include the unpacking/packing stages required to route elements between their packed representation inside registers to individual datapath lanes, because these are required by all vector operations and are not specific to scans (Section 1.4.3). Similarly, the figures also omit FU internal forwarding paths and registers that may be required by specific implementations.

7.3.5 Pipeline Utilization

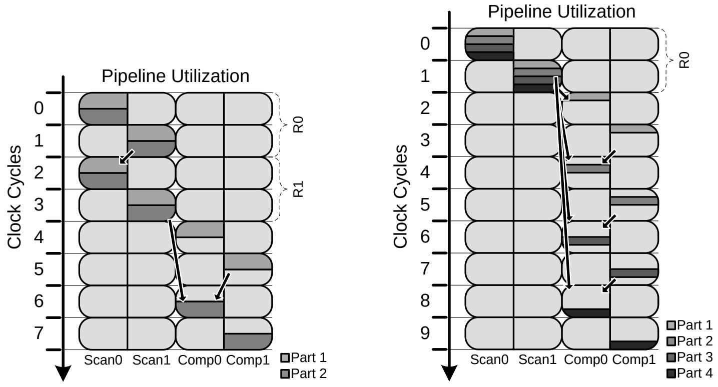
Fig. 7.12 illustrates the utilization of a balanced pipeline processing single-, two- and N/2-part scans. It can be observed that scan partitioning effectively shifts dependencies from consecutive rounds to consecutive computation stages, e.g. part two cannot complete until



(a) Single Part (b) Two Parts (c) N/2 Parts

Fig. 7.12: Utilization of a balanced pipeline (employing two 8-, 4- and 2-element wide FUs for single-, two- and N/2-part scans on an 8-element vector, respectively)

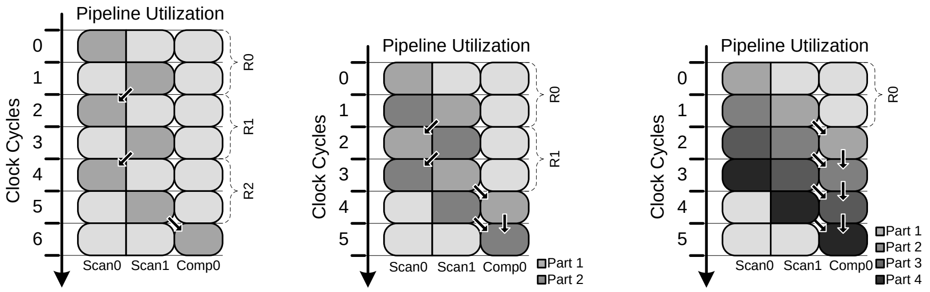
the carry of part one is known (arrows in Fig. 7.12). In consequence, the combination of finite resources (scan stages reused between rounds, Section 7.3.4) and these dependencies can lead to an increased cycle count for partitioned scans. Furthermore, the design in Fig. 7.12b requires a scalar register to temporarily buffer the carry output generated by the computation stage of the first part until the intermediate vector result of the second part is ready. However, this scalar operations can be more energy efficient than the forwarding of an N-element vector between the additional scan round of the single-part variant. Moreover, as the FUs associated with partitioned scans operate on fewer elements and perform less operations, their energy consumption is further reduced (Tab. 7.1).



(a) Two Parts (b) N/2 Parts

Fig. 7.13: Utilization of a balanced pipeline (employing two 8-element wide FUs for , two- and N/2-part scans on an 8-element vector)

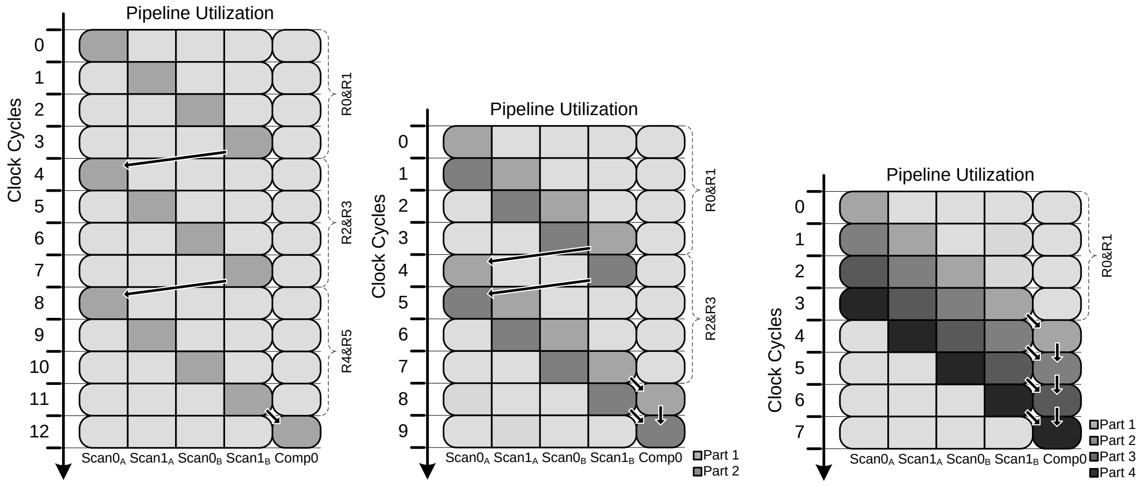
Fig. 7.13a highlights the flexibility of scan partitioning in terms of achievable performance. By executing the rounds associate with both parts in parallel, the cycle count of a two-part scan on the given pipeline configuration can be reduced to match a single-part variant. This assumes the presence of either two scan units or one unit capable of operating on two N/2-element parts in parallel. Note that the corresponding computation FUs still only have to operate on N/P elements. Hence, the computation steps in Fig. 7.13a could employ an existing N-element wide vector ALU and VM to disable inactive lanes.



(a) Single Part (b) Two Parts (c) N/2 Parts

Fig. 7.14: Utilization of a unbalanced pipeline (employing two 8-, 4- and 2-element wide FUs for single-, two- and N/2-part scans on an 8-element vector, respectively)

Fig. 7.14 illustrates the examples used in Fig. 7.13a on an unbalanced pipeline. It can be observed that the partitioned variants exploit the lower execution time of computation steps - when compared to scan rounds - to achieve high datapath utilization and increased performance. As mentioned in Section 7.3.4, the increased number of scan stages is based on slower but more energy efficient circuitry. Note that the utilization of N-element wide FUs to process multiple parts in parallel would - in this case - not impact the overall cycle count, but free up intermediate pipeline stages to be used by other instructions.

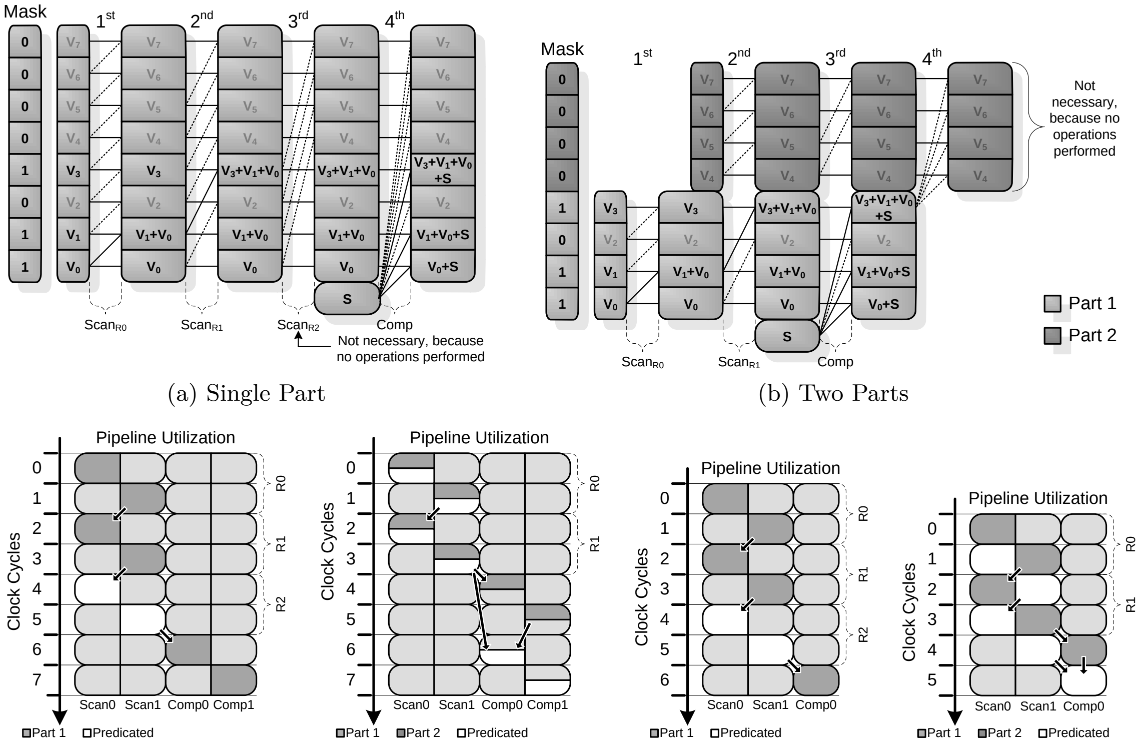


(a) Single Part (b) Two Parts (c) N/2 Parts

Fig. 7.15: Utilization of an unbalanced pipeline after partial Scan-Unrolling (employing two 8-, 4- and 2-element wide FUs for single-, two- and 4-part scans on an 64-element vector, respectively)

An alternative method to implement unbalanced pipelines is scan-unrolling. Fig. 7.15 illustrates the execution of a 64-element scan on a partially unrolled pipeline configuration (Fig. 7.11d); i.e. the replication of the two pipeline stages Scan0 and Scan1 allows computation of two rounds per pass of the scan FU. While unrolling does not affect the performance of the single-part variant, it enables significant speedups for both partitioned implementations. One concern is the handling of scans on vectors with lengths unequal to a power of 4. A pipeline configuration that implements unrolling by replicating scan stages within the same FU (Fig. 7.11d) would require a bypassing circuitry to skip round R5 (Fig. 7.15a). However, this would not be the case for a configuration that replicates scan stages within two individual FUs. Instead, it would just delay issuing of R5 by the number of cycles theoretically required to compute R4. Hence, the result of R4 reaches the computation stage at the same time as the carry output from R3 and does not need to be temporarily buffered.

7.3.6 Compatibility with Per-Lane Predication



(a) Single Part

(b) Two Parts

(c) Single Part (Balanced Pipeline)

(d) Two Parts (Balanced Pipeline)

(e) Single Part (Unbalanced Pipeline)

(f) Two Parts (Unbalanced Pipeline)

Fig. 7.16: Execution of predicated 8-element scan

ARGON introduced per-lane predication as a means to increase the vectorizability of general purpose code (Section 6.2.1). It can be observed in Fig. 7.16 that a vector length of $1/2$ $\mathrm{VL}_{\mathrm{max}}$ effectively predicates one round within single-part scans, while predicating a complete part within corresponding 2-part variants. Assuming the presence of a scheme to handle 0-length micro-ops (Section 7.6), the latencies of both (single- and 2-part variants) may be reduced to 5 and 4 clock cycles on balanced and unbalanced pipeline configurations, respectively. This effectively removes the performance advantage of the 2-part variant on unbalanced pipelines, while maintaining high energy efficiency.

7.3.7 Concluding Remarks

This section showed how scan partitioning allows designers to target specific performance/energy trade-offs for scan operations depending on their frequency of use and complexity. In particular, it generally increases energy efficiency by employing narrow FUs and performing fewer operations per scan. Furthermore, by effectively moving dependencies from scan rounds to computation steps, partitioned scans executed on unbalanced pipeline configurations can exceed the performance of single-part variants. Such pipelines may result from the implementation of slower but more efficient scan FUs, or scan-unrolling. Finally, scan partitioning is compatible with all features investigated by ARGON as well as all microarchitecture optimizations presented in this chapter.

7.4 Efficient Handling of Masks for Vector Predication and Segmentation

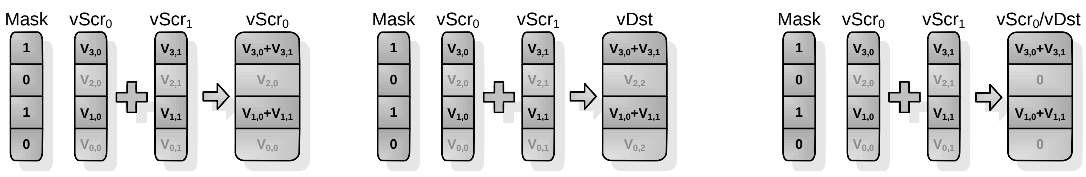
7.4.1 Motivation and Overview

Per-lane predication, as analyzed in the context of ARGON in Section 6.2.1, introduces a requirement for circuitry to implement merging/zeroing behavior for inactive vector elements based on predicate flags (Section 7.4.2). Using dedicated hardware to implement this functionality imposes additional area and energy costs. This section proposes a method to re-use existing circuitry to mitigate these costs. The forwarding operations as required by segmented scans (Section 6.2.4) can be implemented, too, by further expanding the augmented control circuitry to take two flags into account. The relevance of merging/zeroing circuitry becomes apparent in the context of the average vector length utilized by individual benchmarks, e.g. 3.5, 13.9, 7.4 and 6.0 elements for the baseline implementation of BackProp, BitAlloc, PathFind and SpMV, respectively (Fig. 6.12; Fig. 7.22 for a breakdown by specific lengths). In summary, the proposed method avoids the cost of additional multiplexing circuitry usually required to;

- Handle predicated elements of merging vector instructions
- Handle predicated elements of zeroing vector instructions
- Handle data-movements for inactive elements within predicated scan instructions
- Handle the isolation of independent segments within segmented scans

The following sections give background information on the treatment of inactive elements by predicated instructions and the data-movements involved within predicated and segmented scans (Section 7.4.2). They furthermore introduce the basic idea behind the proposed method (Section 7.4.3) and its compatibility with Scan Partitioning (Section 7.4.4), before concluding in Section 7.4.5.

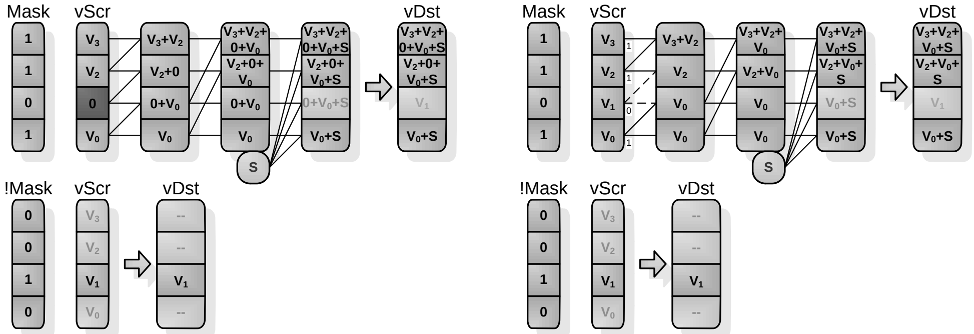
7.4.2 Background



(a) Merging (destructive) (b) Merging (non-destructive) (c) Zeroing (destr./non-destr.)

Fig. 7.17: Examples for the treatment of inactive elements by predicated instructions

Section 6.2.1 introduced vector masks (VM) to selectively control operations on individual vector elements. Fig. 7.17 illustrates potential use-cases for masks as predicates. Merging describes the process of combining computation results with values previously residing in the destination register. While a destructive instruction re-uses one of its sources as destination, a non-destructive instruction provides a dedicated operand, which effectively introduces an additional input dependency. The renaming of registers as performed by OoO processors may require both types of instructions to transfer inactive elements from their original to their (renamed) destination register. Zeroing is an alternative interpretation of predicates that removes this dependency by setting all inactive elements to zero instead of preserving their value (Fig. 7.17c). Both interpretations (merging/zeroing) can be implemented destructively and non-destructively, and be used to derive each other. Their relevance for this section is the requirement to provide unaltered input elements or insert zeros into the corresponding destination registers.



(a) Insert predetermined Element

(b) Bypass inactive Elements

Fig. 7.18: Execution of a predicated scan add (parallel move instruction required to transfer inactive elements to destination register for non-destructive merging)

There are a number of considerations regarding the interpretation of predicates for scans. For instance, the parallelization according to the scheme introduced in Section 6.2.3 requires the computation of intermediate results within lanes corresponding to inactive elements (Fig. 7.18). Non-destructive merging requires inactive values to be preserved during the scan, or an additional move operation using the inverse of the predicate mask. This is not the case for zeroing. There are several more methods to treat inactive values; e.g. replace them with the last active value preceding them ($V_1 \Leftarrow V_0$+S for the example in Fig. 7.18). Most of these methods can be reduced to data-movements or insertions.

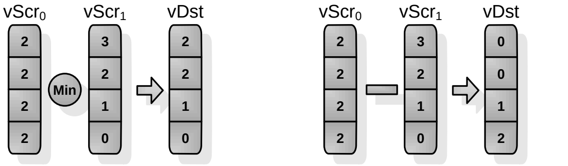
Fig. 7.18a illustrates how predicated scans may be implemented without modifying the operation of the underlying FU. Dummy elements are inserted into inactive lanes, so that they do not affect the outcome of the actual scan, but still allow intermediate results to propagate trough. A disadvantage of this approach is the dependency of those elements on the type of the scan operation; e.g.:

- ScanAdd: insert 0
- ScanMul: insert 1
- ScanMin/Max: insert maximum/minimum value (depends on data type)

An alternative approach that operates independently of instruction and data types is depicted in Fig. 7.18b. It employs multiplexers to prioritize contents of active lanes over those of inactive lanes, within the first scan round. Fig. 7.2b illustrates an approach similar to

Fig. 7.18 for a segmented scan. It uses a descriptor rather than a predicate mask to delineate segments, is capable of performing data-movement in all processing steps (not just the first scan round), and does not exhibit inactive lanes in its result. Nevertheless, handling of predicate and descriptor masks - as well as combinations of both - can be reduced to either perform the binary operation described by the underlying instruction, or propagate one input of said operation in unmodified form. The following section describes a hardware optimization technique to implement the mechanisms required for handling masks without the need for additional routing structures.

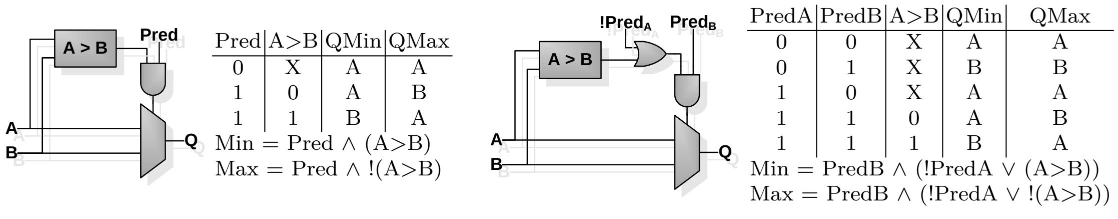
7.4.3 The Basic Idea



(a) Minimum (b) Saturating Subtraction

Fig. 7.19: Commonly implemented vector instructions employing resources that may be re-used for the efficient handling of masks

The previous section described the need for data-movement and insertion capabilities for FUs to implement merging and zeroing, and delineate segments based on masks. Fig. 7.19 illustrates examples for instructions used to determine the minimum between elements of two vectors or performing a saturating subtraction. While the former uses multiplexers to selected specific vector elements, the later employs multiplexers or some other form of combinational logic to substitute negative values with zero.



Pred	A>B	QMin	QMax
0	X	A	A
1	0	A	B
1	1	B	A

Min = Pred ∧ (A>B)
Max = Pred ∧ !(A>B)

PredA	PredB	A>B	QMin	QMax
0	0	X	A	A
0	1	X	B	B
1	0	X	A	A
1	1	0	A	B
1	1	1	B	A

Min = PredB ∧ (!PredA ∨ (A>B))
Max = PredB ∧ (!PredA ∨ !(A>B))

(a) Predicate of current Lane (b) Predicate of both binary Operands

Fig. 7.20: Control logic for multiplexer used by max operation augmented to incorporate predicates

Fig. 7.20a illustrates how the addition of a single AND gate per lane enables existing multiplexers, previously only used to select between two elements for comparison results, to handle the merging behavior of inactive elements of predicated instructions. The corresponding circuitry to implement zeroing re-uses the circuitry previously employed by saturating subtractions. A predicated binary vector instruction such as an add would - based only the predicate of the current lane (Fig. 7.20a) - select between results obtained from a full adder or the augmented multiplexer (Fig. 7.17a). As both values are processed by the same ALU, the corresponding selection circuitry is already in place. Hence, no further delays or energy consuming routing networks are introduced. A two input variant considering the predicates corresponding to the lanes of both input operands may be used to handle the data-movement involved with scan and segmented scan instructions (Fig. 7.20b). It operates similar to Fig. 7.20a, but furthermore allows operand B to be propagated into lane A.

7.4.4 Compatibility with Scan Partitioning and Scalar FUs

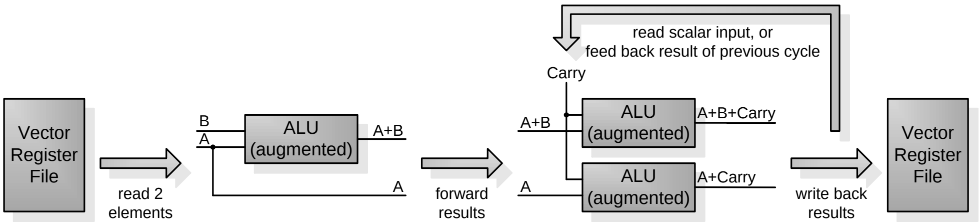


Fig. 7.21: Implementation example for homogeneous N/2-part scan (Fig. 7.9a) using augmented ALU circuitry

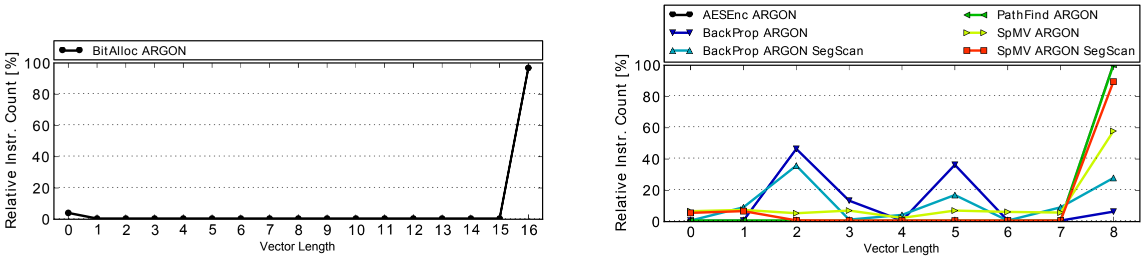
The augmented multiplexer structures introduced in the previous section are fully compatible with Scan Partitioning. Systems operating on wide datapaths may employ fully augmented vector ALUs; e.g. modifying each individual lane of FUs computing single- or 2-part scans. Alternatively, energy oriented processors may employ heavily partitioned or even fully serialized datapaths. Fig. 7.21 illustrates how three augmented ALUs can be employed to implement homogeneous N/2-part scans as proposed in Fig. 7.9a. The low number of ALUs required makes it feasible to use multiple scalar FUs (if available) to compute vector instructions without the need for a dedicated vector datapath. As scalar ALUs rely on flags rather than data-movements to evaluate the relation between two input values (min/max), the multiplexers re-used to merge inactive elements would be related to select rather than max instructions. However, zeroing circuitry would still be based on saturating subtractions.

7.4.5 Concluding Remarks

This section introduced a method to extend the control logic of multiplexing structures – previously only used to select between two elements for comparison results or insert zeros for saturating subtractions – to handle predicated data-movement (merging) and insertion (zeroing). When implemented as part of existing ALUs, this method is unlikely to impose any additional latency on the critical path. It reduces energy consumption and area requirements by removing the need for structures dedicated to handle merging/zeroing behavior. Besides predication, the method also enables the forwarding operations required by segmented scans (Section 6.2.4). Finally, the augmentation of multiplexers within lane specific ALUs allows their utilization within fully parallelized, partitioned, as well as serialized vector FUs.

7.5 Acceleration of Scans on incomplete Vectors

7.5.1 Motivation and Overview

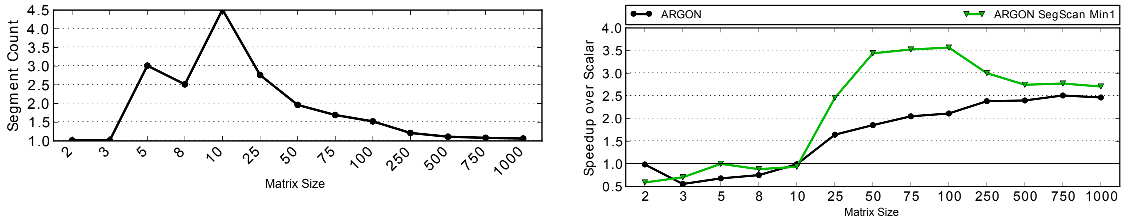


(a) 16-bit Elements on 256-bit wide Datapath

(b) 32-bit Elements on 256-bit wide Datapath

Fig. 7.22: Average vector length utilized by VBench when executed on baseline configuration (Section 6.5.1)

Fig. 7.22 illustrates the distribution of vector lengths utilized by VBench benchmarks based on 16-bit and 32-bit wide data types. It can be observed that the majority of instructions provides optimal hardware utilization by operating on the full vector length. However, several benchmarks exhibit a significant number of operations on vector lengths of two and five elements. Reasons for this can be fix-up statements at the end of vectorized loops or nested loops that do not exhibit sufficient vectorization potential to utilize complete vectors. While, 2-element vectors could be sped up by omitting the final scan round (Fig. 7.25), this is not the case for 5-element vectors. This section proposes a restructured processing scheme that increases the probability of scans on incomplete vectors ($\mathrm{VL} < \mathrm{VL}_{\max}$) being able to omit one or more computation step. In particular, the proposed scheme allows the omission of one step for $\mathrm{VL} \leq \mathrm{VL}_{\max}$ - 1 rather than $\mathrm{VL} \leq {}^1/_2\ \mathrm{VL}_{\max}$.



(a) Average Number of Segments computed by ARGON SegScan M1

(b) Performance over Scalar Implementation

Fig. 7.23: Segment count and performance of SpMV over matrix size (same configuration as Fig. 6.15a)

Above figures show the average number of segments computed and the potential performance benefits for SpMV with respect to the matrix size. When comparing the speedups of the two vectorized variants in (b), it can be observed that segmented scans are particularly beneficial for matrix sizes between 25*25 and 100*100 elements. In the context of (a), this indicates that instructions have to operating on an average of at least 1.5 segments in order to compensate for the computational overhead associated with segmentation. Similarly to incomplete vectors, the execution of such multi-segment scans can be improved by omitting one or more scan rounds. However, this requires the absence of dependencies between elements operated upon by specific rounds, e.g. two independent 4-element segments within an 8-element input vector do not require round 2 (Fig. 7.25). The restructured computation scheme described below loosens this restriction to the absence of dependencies between the most significant vector element and the scalar input. In practice, this is equivalent to the presence of any two or more segments.

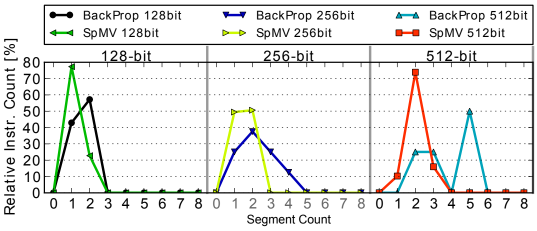


Fig. 7.24: Average number of segments processed over datapath widths ranging from 128- to 256-bit

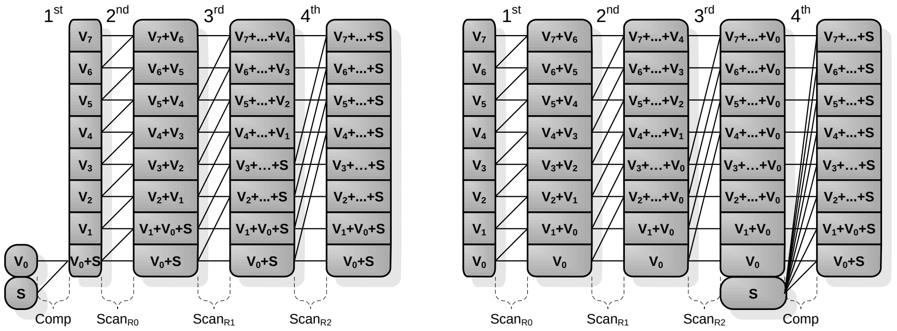
Fig. 7.24 illustrates the average number of segments processed by SpMV and BackProp with respect to the underlying datapath width. It can be observe that with the exception of SpMV on a 128-bit wide datapath, the majority of segmented instructions compute two or more segments, and would therefore benefit from the proposed scheme. Note that the number of segments employed by BackProp is highly predictable, as it depends on the number of neurons within each layer of the neural network. This is not the case for SpMV, because its workloads are based on randomly generated matrices; hence, the corresponding data points are less clustered / wider distributed.

In summary, this section introduces a restructured processing scheme for scans, to increase the probability of being able to omit one or more computation steps. The corresponding performance impact is thereby instruction dependent; e.g. an integer add or a FP multiplication operation may save 1 or 4+ CPU cycles, respectively. The scheme is fully compatible with the data-movements and insertions as described in Section 7.4, as well as Scan Partitioning. As the newly introduced operations follow the same computational patters as used for the remaining vector elements, they can be implemented without diminishing the efficiency of existing FUs (Section 7.5.4). The scheme allows:

- Scans operating on $VL \leq VL_{max}$ - 1 to save one computation step; a conventional implementation allows this only for $VL \leq 1/2\ VL_{max}$ element scans
- Segmented scans that include two or more segments of arbitrary length may save one computation step; a conventional implementation allows this only if there are no dependencies between the lower and upper half of the input vector

The following sections give relevant background information (Section 7.5.2), introduce the restructured scheme for scans and segmented scans (Section 7.5.3), discuss its compatibility with Scan Partitioning (Section 7.5.4), and present derivatives that improve its flexibility (Section 7.5.5), before concluding in Section 7.5.6.

7.5.2 Background



(a) Add Scalar first

(b) Add Scalar last

Fig. 7.25: Conventional implementations of 8-element vector scan add

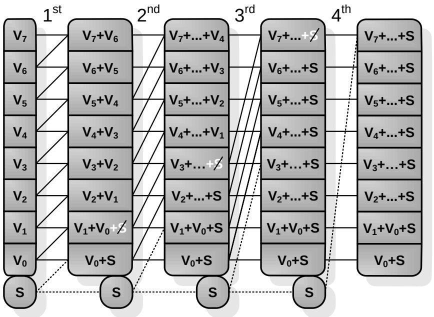
Fig. 7.25 illustrates two implementation variants for a vector scan add (Section 6.2.3). Both perform all operations concerning vector elements in log$_2$N steps and require one additional step to account for the scalar input. Fig. 7.25a processes the scalar input first, resulting in an overall operation count of $1 + \sum_{i=0}^{log_2 N-1}(N - 2^i)$. In contrast, (b) delays this computation until the end, resulting in N instead of 1 operations for this step. While a lower operation count indicates a higher efficiency for (a), its utilization of a scalar ALU for the initial step imposes undesired data-movements between scalar and vector FUs and increased control complexity. Said variant furthermore suffers performance penalties in the context of Scan Partitioning (Section 7.3). The alternatives avoids those by effectively shifting dependencies between consecutive parts - i.e. operations on the scalar input - to the last processing step, which allows a high degree of pipelining between individual parts.

A commonality of the variants in Fig. 7.25 is the requirement for log$_2$(N)+1 steps to process an N-element vector and a scalar input. This is equivalent to a (2*N)-element scan without a scalar input. However, reducing the current vector length to N-1 or employing a corresponding mask does not reduce the step count accordingly. The nature of the operations performed by individual steps requires VL to be less than or equal $^1/_2$ VL$_{\text{max}}$ in order to save one complete step (Scan$_{\text{R2}}$, Fig. 7.25). Note that data-movements as described in Sections 7.4 and 7.6 may still be required to account for the merging/zeroing behavior of inactive elements within omitted steps. In general:

⇒ Given VL $\leq$ $^1/_{2^n}$ VL$_{\text{max}}$ and n = 1,2,..., the last n rounds of Fig. 7.25a and (b) may be omitted

7.5.3 The Basic Idea

The restructured computation sequence illustrated in Fig. 7.26 is an alternative to those presented in the previous section. It requires the same number of processing steps to compute full-length vectors. However, instead of accounting for the scalar input within the first or last step, it uses said value once within each processing step. In consequence, the final step incorporates only one operation. As this operation is between the carry input

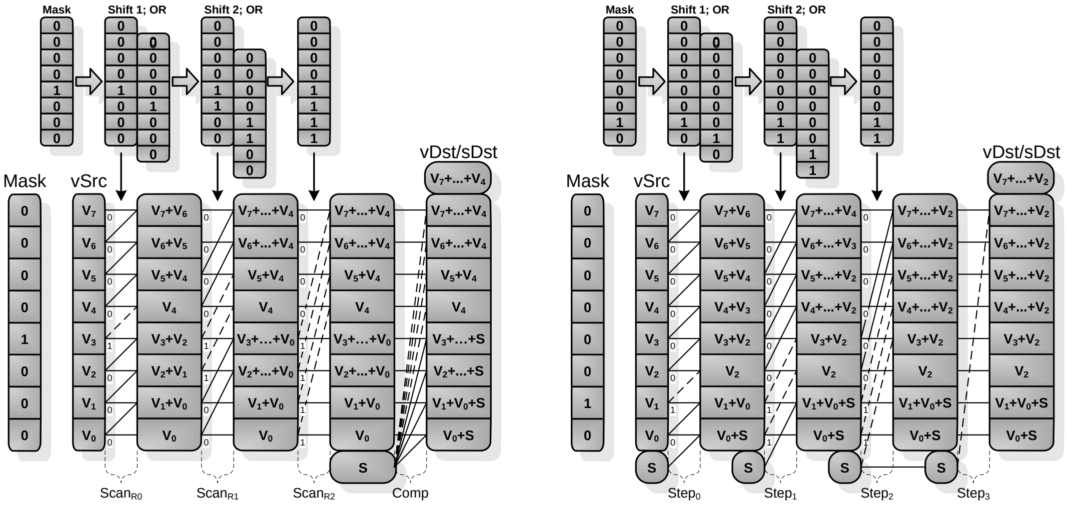


- Crossed-out elements indicate values that would be present in Fig. 7.25a
- Dashed lines indicate operations concerning the scalar input

Fig. 7.26: Restructured scan operation

and the most significant vector element, it and the associated step can be omitted for VL $\leq$ VL$_{\max}$ - 1. This significantly increases the probability of scans being able to save their last computation step. In general:

⇒ Given VL $\leq$ ($^1/_{2^{n-1}}$ VL$_{\max}$) - 1 and n = 1,2,..., the last n rounds of Fig. 7.26 may be omitted



(a) Add Scalar last

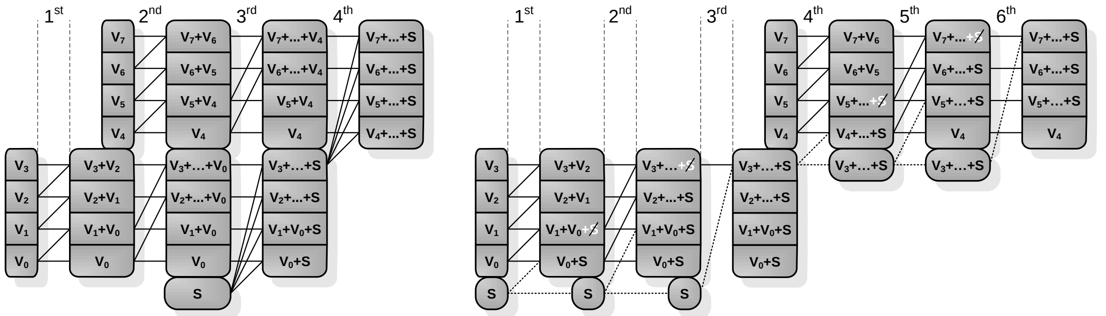
(b) Restructured

Fig. 7.27: 8-element segmented scan add

Fig. 7.27a illustrates the computation of an 8-element segmented scan using the "add scalar last" scheme (Sections 7.2 and 6.2.4). It can be observed that saving the final round step (Scan$_{\text{R2}}$) requires the absence of any dependencies between elements of the lower and upper half of the input vector, i.e. a segment boundary at element index 4. One or more boundaries at any other position do not permit this optimization. In contrast, as the only operation performed by the restructured variant (Fig. 7.27b) concerns the scalar input and the most significant vector element, the presence of any two or more segments allows the omission of Step$_3$. Note that this does not include an N-element segment ending at the most significant element positions followed by a new segment without any element within the current input vector. More general, this condition might be expressed as the presence of two or more segments of length VL$_{\max}$ -1 or less. The corresponding hardware circuitry might identify such cases using a simple OR operation on all mask bits except the most significant one.

Another advantage of the restructured scheme is its regularity. The introduced operations follow the same computational pattern employed to handle the remaining vector elements and the number of operations per step does not exceed N. Hence, scans do not require additional computation circuitry, but may re-use existing ALUs that are otherwise employed to operate upon two N-element vectors. Furthermore, as the computation step to be potentially saved is the last one to be processed, it can be identified and handled - e.g. by squashing of the corresponding micro-op - without introducing additional latencies. Note that those instructions that can be computed using ternary operations may combine multiple computation steps to increase their performance even on full-length vectors. For instance, a scan add instruction could employ a three input FU within its least or most significant lane to save the initial or final computation step corresponding to the schemes in Fig. 7.25a or Fig. 7.26, respectively.

7.5.4 Compatibility with Scan Partitioning



(a) Conventional (add last)

(b) Restructured

Fig. 7.28: Partitioned 2-part scan add

Fig. 7.28a and Fig. 7.28b illustrate conventional and restructured representations of a 2-part scan add, respectively. The partitioning of scans as proposed in Section 7.3, allows designers to weight the complexity and energy consumption of vector FUs against their performance. The conventional computation sequence leans itself particularly well to this execution paradigm. However, a naive conversion of the restructured scheme results in a severe performance penalty; i.e. requirement for six instead of four steps (Fig. 7.28).

Fig. 7.29 shows a hybrid of the conventional and restructured scheme that avoids performance penalties and increases the probability to achieve speedups for scans on incomplete vectors. It implements the conventional scheme for all but the first part of the scan to enable a high degree of pipelining between individual parts. This approach does scale with the addition of more parts, but is most suitable for FUs with coarse grain partitioning. The regularity of the newly introduced operations and the fact that they are performed in parallel to similar operations of the conventional scheme, allows FUs to support both schemes without significant overhead; i.e. a restructured sequence for part one pipelined

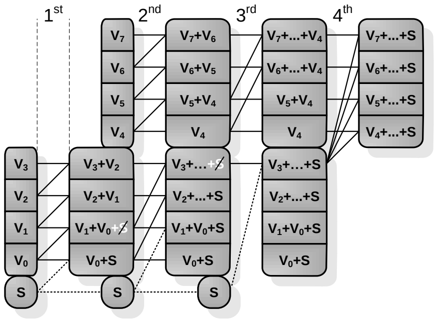


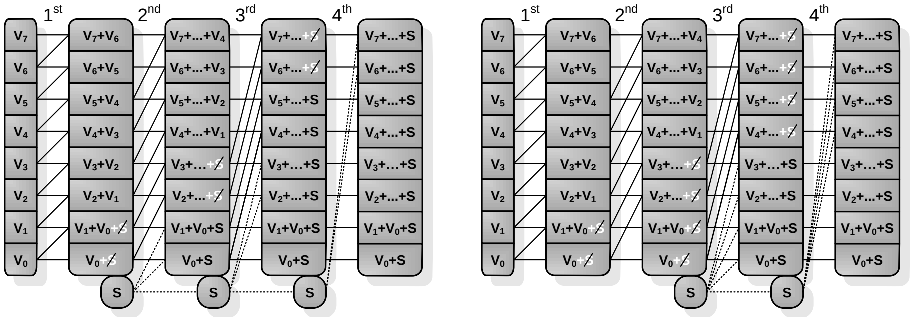
Fig. 7.29: Hybrid of conventional and restructured scheme to process a 2-part scan add

with conventional sequences for the remaining parts. Tab. 7.2 lists the number of steps that may be omitted with respect to specific vector lengths for the conventional and hybrid scheme.

Steps omitted	Conventional	Hybrid
1	VL ≤ $1/2$ VL_{max}	VL ≤ $1/2$ VL_{max}
2	VL ≤ $1/4$ VL_{max}	VL ≤ $1/2$ VL_{max} - 1
3	VL ≤ $1/8$ VL_{max}	VL ≤ $1/4$ VL_{max} - 1
		n < P: VL ≤ $1/2^n$ VL_{max}
n.. for n = 1,2,...	VL ≤ $1/2^n$ VL_{max}	n ≥ P: VL ≤ $1/2^{n-1}$ VL_{max} - 1
		P.. number of parts

Tab. 7.2: Number of steps potentially omitted relative to underlying vector length of 2-part scan

7.5.5 Derivative Schemes to Delay Scalar Input Dependency



(a) 1-Step Delay ⇒ VL ≤ VL_{max} - 2 (b) 2-Step Delay ⇒ VL ≤ VL_{max} - 4

Fig. 7.30: Derivative schemes based on Fig. 7.26

The previous section discussed the performance advantage achieved by delaying operations regarding scalar inputs to the final computation step to maximize the suitability of specific scan partitions for pipelining. A similar advantage can be observed for scans on multi-issue OoO processors. In particular, a scan may be issued while its scalar input is still being processed, by a preceding scan or a number of scalar instructions. The restructured scheme presented in Section 7.5.3 does not support this kind of acceleration due to its dependency on the scalar input within the first step. Fig. 7.30 illustrates two derivatives that delay computations involving the scalar value for the cost of having more operations within subsequent steps. Consequently, the omission of steps requires a appropriately shortened vector length. In general:

⇒ Given a delay of x steps, with x = 0,1,2 … ($\log_2$N -1), and VL ≤ ($1/2^{n-1}$ VL$_{\max}$) - 2^x, with n = 1,2,…, the last n rounds of a restructured scan may be omitted

While the number of operations regarding the scalar input is proportional to the delay ($2^x\,(1 - x + log_2 N)$), the sum of operations per step does not exceed N. Hence, all variants derived from the restructured scheme may employ existing ALUs to avoid the need for additional hardware components. In the context of segmentation, the last computation step may be omitted for scans that exhibit no dependencies between the first VL$_{\max}$ $- 2^x$ elements and the remainder of the input vector (Fig. 7.27b). The corresponding circuitry might identify these cases using a simple OR operation on all relevant mask bits.

7.5.6 Concluding Remarks

This section introduced a processing scheme that increases the probability of scans being able to omit the execution of specific steps. In particular, it enables scans operating at vector lengths of VL$_{\max}$ - 1 rather than 1/2 VL$_{\max}$ to save one step. It allows similar savings for segmented scans comprising two or more segments of length VL$_{\max}$ - 1 instead of specifically requiring a segment boundary separating the upper and lower half of the input vector. The scheme is fully compatible with the data-movements and insertions discussed in Section 7.4. A partitioned variant can be implemented without performance degradation by combining the proposed with a conventional (add scalar last) scheme within a hybrid variant (Section 7.5.4). The newly introduced operations mimic the computational patterns already implemented by scan FUs and can therefore be implemented without diminishing the efficiency of existing datapaths. Derivatives of the scheme allow the dependency of a scan on its scalar input to be shifted into a later computation step in order to increase issue rates while decreasing the probability of being able to save individual steps (Section 7.5.5).

7.6 Efficient Handling of Zero Length Vector Micro-Ops

7.6.1 Motivation and Overview

The motivation for the restructured processing scheme introduced in Section 7.5 was based on the average vector length utilized by VBench illustrated in Fig. 7.22. The same figure also shows that up to 5% of the vector instructions within certain benchmarks exclusively operate on inactive elements, i.e. they do not perform any actual computations. This section introduces mechanisms to increase the energy efficiency and performance of these instructions by identifying them during dispatch and/or issue and substituting them based on their type either with NOPs or vector moves. The mechanism are transparent to the programmer and do not impose any overhead in form of additional instructions. As they operate on a micro-op granularity, they are able to exploit optimization potential on a very fine granularity, extending to an intra-instruction level when employing partitioned datapaths (Section 7.3). The potential performance gained from substitutions depends thereby on the latency and availability of the FUs involved, i.e. a multi-cycle computation might be replaced by a single cycle move or an instantaneous NOP. Furthermore, the number of input dependencies of substituted micro-ops may be reduced to a degree that allows them to be issued sooner. The increase in energy efficiency is based on the complexity of the original operations and the amount of leakage saved due to increased performance. In general, move operations are highly efficient as they primarily rely on routing networks and short term storage elements. The following use cases were identified based on VBench (Chapter 6):

- Fix-up statements following vectorized loops to handle iterations with less than $\mathrm{VL}_{\mathrm{max}}$ elements
- Operations following type pro-/demotions
- Workloads/algorithms with limited vectorization potential executed on partitioned datapaths

The following sections elaborate on these use cases and introduce the term 0-length instruction (Section 7.6.2), present a categorization scheme for the handling of different types of those instructions (Section 7.6.3.1), and propose corresponding hardware adaptations (Section 7.6.3.2), before concluding in Section 7.6.4.

7.6.2 Background

The introduction of per-lane predication into a vector ISA gives rise to the possibility of instructions exclusively comprised of inactive elements (e.g. VL = 0 and/or mask = all 0's); henceforth, referred to as 0-length instructions. Individual instructions might be split into multiple micro-ops that either operate on a subset of elements or perform a multitude of operation on the same data. In the former case, only a subset of the available predicate information corresponds to elements of specific micro-ops; e.g. the upper and lower half of the mask = 0000 1111 may be processed by two micro-ops, only one of them being 0-length.

```
for(i = 0; i < x; ++i) {
  rest  = nElem[i] & (VL_max - 1);
  vElem = nElem[i] - rest;
  vsetVL(VL_max);
  for(j = 0; j < vElem; ++j){
    ... loop body (vectorized) ...
  }
  if (rest) { // Guard
    vsetVL(rest);
    ... loop body (vectorized) ...
  }
}
```

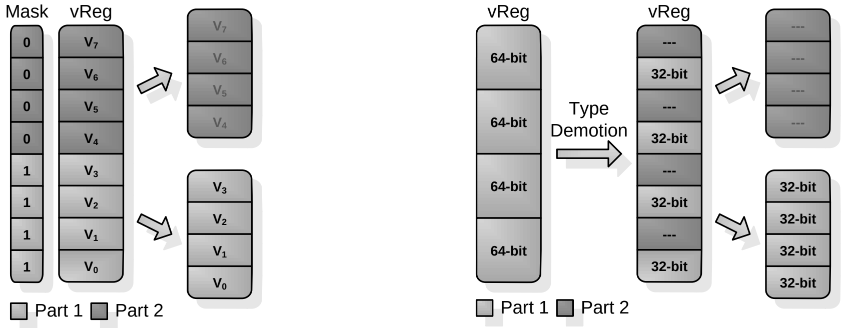
(a) Fix-up Statements following inner Loop

```
for (i = 0; i < x; ++i) {
  ... load vectors of half-words vA and vB ...
  mask = (vA >= vB);  // Mask (half-words)

  mask_p0 = mask;      // Split in 2 masks (words)
  mask_p1 = mask >> VL_max;

  if (mask0_p0) { // Guard
    ... operations on words granularity ...
  }
  if (mask0_p1) { // Guard
    ... operations on words granularity ...
  }
}
```

(b) Split and Reinterpretation of vector Mask after Type Promotion

Fig. 7.31: Use cases for branches guarding the execution of 0-length instructions

The AVX-512 ISA extensions and the Larrabee architecture both support some form of per-lane predication and suggest the use of enclosing branches to guard the execution of zero length instructions [24, 70] (Fig. 7.31). However, this approach relies on energy intensive branch predictors and the regularity of vectorized code segments to exploit potential performance benefits. The penalties for miss speculations can be severe. For instance, the greatest impact of the guarding branch in Fig. 7.31a can be observed for the ARGON SegScan Min1 variant of SpMV. Its baseline configuration exhibits a performance reduction from approximately 3.5x to 1.5x when comparing version without and with said branch (Section 6.5.1). Reasons for this are a low number of instructions within the kernel's inner loop and the random structure of the underlying input set. Another consideration is the granularity of guarding branches. For example, Fig. 7.31b illustrates two separate branches guarding the upper and lower half of a mask after a type promotion. A coarse grain variant might use a single guard employing the initial half-word based mask. The optimal granularity depends on factors such as the number and type of instructions to be guarded, the workload to be processed, and the branch predictor and pipeline specifics.



(a) Partitioned in upper/lower Half

(b) Partitioning in even/odd Elements

Fig. 7.32: 0-length micro-ops within partitioned vector instructions

Fig. 7.32a illustrates an example for a 0-length micro-op as it might be exhibited frequently by algorithms or workloads with limited vectorizability. As the original instruction still needs to be executed, this use case cannot be covered by a guarding branch. It is particularly relevant for long latency instructions such as vector multiplications or scans that might be executed on partitioned datapaths to save energy and hardware complexity (Section 7.3). For instance, an underutilized vector instruction might exhibit a performance close to or worse than a scalar equivalent; whereby 0-length components not just impose undesired latency but also energy consumption. Fig. 7.32b depicts an alternative partitioning scheme based on the separation of even and odd vector elements. This scheme may be employed on unpacked or hybrid datapaths to simplify type conversion to simple reinterpretations and avoid data movement within vectors (Section 1.4.3; Fig. 6.16). Other partitioning schemes or combinations thereof are possible and become more feasible with increasing datapath widths. In any case, the presence of per-lane predication on a partitioned datapath gives rise to the possibility of 0-length micro-ops that may be optimized for energy consumption and performance.

7.6.3 The Basic Idea

The preceding section discussed guarding branches as one method of reducing the energy and performance impact of 0-length instructions. However, this method is limited by its coarse granularity and the cost of miss speculations. The microarchitecture optimizations presented here operate on a micro-op rather than a instruction granularity and focus on processing 0-length operations efficiently instead of guarding them from execution. The proposed modifications are transparent to the programmer and not meant to replace guard statements completely, but rather limit them to long or particularly well predictable code sequences, while benefiting short sequences and exploiting intra-instruction optimization potential. The following subsections discuss how 0-length micro-ops may be substituted based on their type either with NOPs or vector move operations (Section 7.6.3.1), and how they might be identified and handled during dispatch or while residing inside their corresponding issue queue (IQ). The potential performance gained from substitutions depends on the latency and availability of the FUs involved. For example, the basic timing profile employed for the analysis of VBench assumes three execution cycles for ALU based vector instructions and only two for moves (FullDP profile; Section 6.4.2). As the corresponding baseline configuration furthermore assumes dual issue capability for the vectorized datapath and two permutation FUs, up to two 0-length micro-ops might be processed in parallel, leaving the originally targeted FUs free to be used by subsequent instructions. The increase in energy efficiency due to substitutions is based on the complexity of the original operation and the amount of leakage saved due to increased performance. In general, the hardware resources involved in move operations are very simple and energy efficient as they are primarily composed of routing networks and short term storage elements.

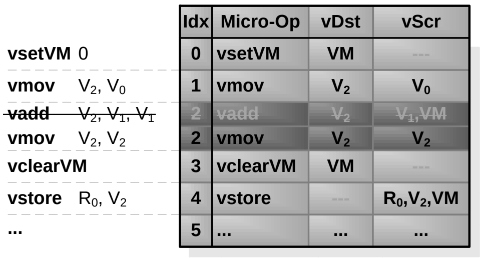
7.6.3.1 Categorization and Handling of 0-length Micro-Ops

Instruction Type	Zeroing	Destructive & Merging	Non-destr. & Merging
Description	Predicated elements of the destination register are set to zero.	One source register implicitly acts as destination register. Predicated elements of this register are preserved. Due to register renaming, OoO processors may need to copy elements from the original (prior renaming) to the new (after renaming) register.	Allow separate operands for source and destination registers. Predicated elements of the destination register are replaced by corresponding elements of a source register.[1]
Substitute	vmov vDst, 0	In-order: NOP OoO: vmov $\mathrm{vDst}_{\mathrm{new}}$, $\mathrm{vDst}_{\mathrm{org}}$	vmov vDst, vSrc
Dependencies	None	In-order: None OoO: $\mathrm{vDst}_{\mathrm{org}}$	vSrc

[1] Note that by replacing inactive elements of the destination with elements of the source register, the input dependency to said destination is removed and therefore the probability of stalls due to pending register updates reduced. An alternative would be to preserve inactive destination register elements and handle 0-length micro-ops as described for destructive & merging instructions.

Tab. 7.3: categorization and handling of 0-length micro-ops

The handling of 0-length micro-ops for ARGON distinguishes the three instruction types listed in Tab. 7.3. Note that in order to avoid nondeterminism, it is under most circumstances insufficient to replace micro-ops with NOPs; instead vector move operations (vmov) are required to update the corresponding destination registers (vDst) and potentially preserve input dependencies. Special care is to be taken for instructions that would naturally perform flag updates. As the actual number of vector instructions falling into this category is very small and inactive (predicated) elements are generally considered irrelevant to flag updates, vmov FUs may be extended to support this functionality.



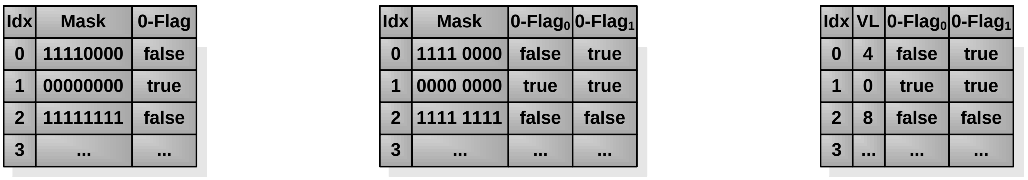
vsetVM 0
vmov V_2, V_0
~~vadd V_2, V_1, V_1~~
vmov V_2, V_2
vclearVM
vstore R_0, V_2
...

Idx	Micro-Op	vDst	vScr
0	vsetVM	VM	---
1	vmov	V_2	V_0
~~2~~	~~vadd~~	~~V_2~~	~~V_1,VM~~
2	vmov	V_2	V_2
3	vclearVM	VM	---
4	vstore	---	R_0,V_2,VM
5	...	...	...

Fig. 7.33: Preservation of input dependencies for issue queue entries after substitution to avoid invalid re-ordering

Fig. 7.33 illustrates a sequence of IQ entries to exemplify the relevance of preserving data dependencies as listed in Tab. 7.3. In particular, the vadd at index two is assumed to be non-destructive & merging. Hence, the corresponding substitute has to wait until the preceding vmov – that does not depend on VM – completes. Otherwise, the subsequent store would operate on the elements initially residing in V_1 instead of those moved in from V_0, and consequently corrupt architectural state. In contrast, a move corresponding to a zeroing instruction would not exhibit this dependency and could bypass the preceding vmov, should V_0 not be available at the time.

7.6.3.2 Detection of 0-length Micro-Ops



Idx	Mask	0-Flag
0	11110000	false
1	00000000	true
2	11111111	false
3	...	...

(a) One 0-Flag per Mask

Idx	Mask	0-Flag$_0$	0-Flag$_1$
0	1111 0000	false	true
1	0000 0000	true	true
2	1111 1111	false	false
3	...	...	...

(b) Separate 0-Flags for upper/lower Half of each Mask

Idx	VL	0-Flag$_0$	0-Flag$_1$
0	4	false	true
1	0	true	true
2	8	false	false
3	...	...	...

(c) Separate 0-Flags for upper/lower Half of each VL

Fig. 7.34: Extended mask and vector length register files

The following paragraphs describe two distinct ways to identify and handle 0-length micro-ops that may be employed in tandem or independently. Both rely on the ability to discern if a particular mask register or a specific part thereof is exclusively composed of zeros. A naive approach to implement this ability would employ a NOR network during each access to the mask register file to compute corresponding flags (0-flags). Note that ARGON supports only a single set of implicitly addressed VM and VL. This limitation is favorable in terms of encoding space and hardware complexity, but adversely impacts code density and performance (Section 6.6). Hence, a vector oriented architecture may support multiple explicitly addressable VMs within a mask register file and emulate the functionality of VL using masks.

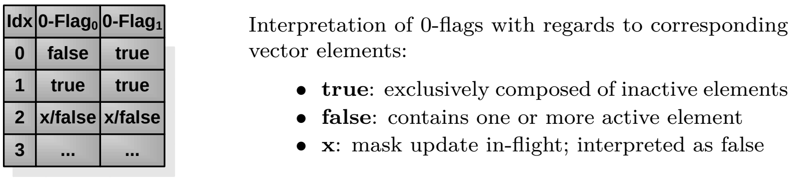
Fig. 7.34a and Fig. 7.34b show examples for extended mask register files that exploit the fact that the majority of vector instructions is limited to mask reads rather than writes. They avoid the energy consumed by repeated flag computations, by performing those only once during mask updates and storing their results for later use. Besides reducing energy consumption, they furthermore shifts computation latencies from mask reads to writes, i.e. in the direction of infrequently used, low latency control instructions. The difference between the presented mask register files is the distinction of individual parts within a given mask. This separation is relevant for processors with partitioned datapaths that split vector instructions into multiple micro-ops, each operating on a subset of elements (Fig. 7.32). In this context, 0-flag$_0$ and 0-flag$_1$ correspond to the lower and upper half of a mask, respectively. However, alternative schemes - i.e. split by even/odd elements - are conceivable, too. Fig. 7.34c illustrates a similarly extended vector length register file. In praxis, implementations are unlikely to employ a dedicated vector length register file alongside a mask register file. Nevertheless, both implementation variants are compatible with the micro-architectural optimizations described here.

Late detection inside issue queue

The key advantage of performing the substitutions suggested in Tab. 7.3 during issue rather than dispatch, is the ability to exploit 0-length micro-ops even though an update to the respective vector mask register is currently in-flight. Fig. 7.33 illustrates this based on a vadd operation residing behind an older control instruction (vsetVM). Assuming that both instructions operate on the same mask, the corresponding 0-flag is not available during dispatch; hence, no substitutions can be performed at that point. The unaltered vadd is therefore dispatched to the IQ and waits until the vsetVM completes and broadcasts the resulting mask. This broadcast mechanism can be extended to transfer 0-flags. In case of said flag being false, the vadd proceeds as normal, otherwise it is substituted based on its instruction type. Similarly, all other read accesses to the mask register file - e.g. by instructions not proceeded by an older vsetVM - are accompanied by the corresponding 0-flags and may cause a substitution.

There a two potential concerns when performing substitutions within the IQ. First, instructions replaced by a NOP do not update their destination register. As subsequent instructions already residing inside the IQ may depend on said register (or potential flag updates), it is necessary to inform them via a broadcast that the register is ready. Second, processors may employ multiple separate IQs associated with different vector FUs, e.g. integer, FP and load/store FUs. In case of either IQ not being able to access a FU capable of performing a vmov, said IQ would either not be able to perform substitutions or need to transfer substituted entries to an alternative IQ. The latter is infeasible in so far that the register dependencies of all instructions within the alternative IQ would have to be taken into account. A transfer between IQs would only be practical for very small or empty queues. However, a vmov FU primarily consists of a set of wires and flip-flops for routing and temporary storage. The costs associated with additional units are therefore considered low.

Early detection during dispatch



Idx	0-Flag$_0$	0-Flag$_1$
0	false	true
1	true	true
2	x/false	x/false
3	...	...

Interpretation of 0-flags with regards to corresponding vector elements:

- **true**: exclusively composed of inactive elements
- **false**: contains one or more active element
- **x**: mask update in-flight; interpreted as false

Fig. 7.35: 0-flag register file as part of the dispatch unit

An alternative (or addition) to the previously described approach avoids retroactive IQ updates by identifying 0-length micro-ops during dispatch rather than issue. The 0-flags corresponding to instructions about to be dispatched are evaluated and substitutions performed in accordance with Tab. 7.3. To reduce the pressure on the mask and/or vector length registers files, a 0-flag register file may be employed (Fig. 7.35). The entries of this structure are updated as follows:

- Invalidate (x/false) on dispatch of write to specific mask register
- Reset (false) not required; default, due to implicit invalidation to x/false
- Set (true):
  a) All mask register updates (true/false ⇒ true)
     - ∗ Does not require extended mask register file (Fig. 7.34)
     - ∗ Increased interaction between execution and dispatch stage
  b) Only on changing 0-flags (false ⇒ true)

    - ∗ Reduced interaction between execution and dispatch stage

- No further invalidation needed during pipeline flushes, because mask register updates are non-speculative

The advantages of this approach are its simplicity and the low impact on subsequent pipeline stages. However, on processors that allow a high number of in-flight instructions, this method might result in several instructions being dispatched while a corresponding mask register update is in-flight. As all 0-flags associated with outstanding updates are conservatively invalidated (x/false), this may result in missed optimization potential for several 0-length micro-ops. A performance focused implementation might consequently implement both, a modified dispatch and issue circuitry to handle 0-length micro-ops.

7.6.4 Concluding Remarks

This section introduced mechanisms to increase the energy efficiency and performance of instructions that, based on per-lane predication would operate exclusively on inactive elements. This is achieved by identifying so called 0-length micro-ops and substituting them based on their type either with NOPs or vector moves. The identification process may be implement during dispatch and/or issue to achieve specific trade-offs between hardware complexity and coverage. The mechanisms are transparent to the programmer and meant to be used in conjunction with branches guarding the execution of 0-length code segments. In particular, as they do not introduce any overhead in form of additional instructions, do not impose miss-speculation penalties and operate on micro-ops, they are able to exploit optimization potential on a very fine granularity. This can extend to an intra-instruction level when employing a partitioned datapath. The potential performance gained from substitutions depends on the latency and availability of the FUs involved, i.e. a multi-cycle computation might be replaced by a single cycle move or an instantaneous NOP. Furthermore, the number of input dependencies of substituted micro-ops may be reduced to a degree that allows them to be issued sooner. The increase in energy efficiency is based on the complexity of the original operation and the amount of leakage saved due to increased performance. In general, move operations are highly efficient as they primarily rely on routing networks and short term storage elements.

7.7 Multi-Register Operations: Improving the Utilization of the Vector Register File

7.7.1 Motivation and Overview

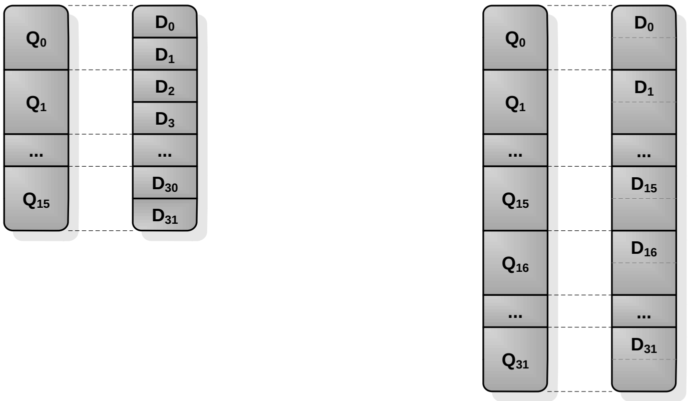
The development of the Vector Benchmark Suite in Chapter 6 revealed that the vectorized code segments of the analyzed benchmarks utilize less than half of the available vector register bank. Inspired by ARMv7 NEON's support of Q registers, ARGON was extended to support so called double length (L-type) instructions that operate on logical pairs of physical registers. The comparison of benchmark implementations operating on datapath widths ranging from 128- to 512-bit and employing conventional (nL-type) and L-type instructions showed that - depending on the vectorization potential of a given algorithm and workload - the performance of L-type implementations reach up to 70% of the improvement achieved when operating on datapaths twice as wide (Section 6.2.5).

This section generalizes the idea of double length registers to logical registers composed of an arbitrary (usually power of 2) number of physical registers. In contrast to ARMv7 NEON, physical registers are not split into multiple individually addressable parts, but combined into logical groups; e.g. pairs in case of L-type operations. Instructions are split into one or more micro-ops during decode, depending on the number of physical registers addressed by each of their operands. The individual micro-ops are then executed in pipelined or – if redundant FUs are available – parallel fashion on the same datapath utilized by conventional operations. Hence, the energy consumption imposed by the support of multi-register operations is limited to modified control circuitry instead of costly datapath extensions. The achievable performance benefits are based on improved vector register file utilization, higher code density, and reduced pressure on fetch and decode circuitry.

The following sections give insight on the background of multi-register operations in the context of different mapping schemes employed by existing vector ISA extensions (Section 7.7.2). Next, the basic concept of multi-register operations is introduced, including explicit and implicit methods for their invocation, potential register mapping schemes, and adaptations for decode & issue circuitry (Section 7.7.3). Finally, a set of concluding remarks is presented in Section 7.7.4.

7.7.2 Background

Modern vector ISA extensions tend to support increasing SIMD widths with every iteration to enable greater performance benefits. For instance, Intel went from 64-bit MMX to 128-bit SSE and 256-bit AVX up to 512-bit wide registers for the upcoming AVX-512 [24]. However, this trend also increases the risk of unnecessary energy dissipation due to datapath underutilization (Section 6.5.2), and requires special consideration to support legacy code. A common approach to both concerns is the mapping of logical onto physical register. Fig. 7.36a exemplifies this for ARMv7 NEON, which maps two 64-bit D registers into each



(a) Multiple logical per physical Register (b) One logical per physical Register

Fig. 7.36: Mapping of double- (D) onto quadword (Q) registers by ARMv7 (a) and ARMv8 (b) NEON

128-bit Q register. While this enables the full utilization of the vector register file by both types, it limits the number of Q registers to 16 rather than 32. Widening the register file from 1 KByte to 2 KByte would remedy this, but also either increase the number of bits required to encode D registers or leave half the register file inaccessible. Moreover, the logical partitioning of physical into multiple logical registers increases the complexity of the corresponding access network. This is particularly relevant for low- to mid-range processors (e.g. ARM A8 and A9) that execute vector instructions in 64-bit parts to save hardware complexity and energy consumption. The partitioned vector register files proposed by Lee and Smith takes this approach even further, i.e. by partitioning the Cray Y-MP's eight 256-element wide physical into thirty-two 64-element wide logical registers [100]. To reduce hardware complexity, they limit logical accesses to one per physical register and derive an algorithm for register allocation to lower the probability of access conflicts. However, this approach is not compatible with OoO processors, which may rename registers at runtime.

An alternative mapping approach employed by ARMv8 NEON and in similar form by the different Intel AVX versions is depicted in Fig. 7.36b. It maps one D register into the lower half of each Q register. This avoids any access conflicts, while keeping the hardware complexity low. Furthermore, it requires the same number of encoding bits for both logical types, but limits the utilization of the vector register file for the D type to 50%. To mitigate this underutilization Intel offers prefixes, which enable narrow logical types to address the upper bits of wider register types; e.g. prefixes VEX.256 and EVEX.512 refer to bits 255:128 and 511:256 of a 512-bit wide register, respectively. In summary, Fig. 7.36 illustrates two approaches commonly used to allow the execution of legacy code and avoid underutilization of wide datapaths. Both map one or more logical register within each physical register and use instructions to explicitly access certain register types. The two variants trade of hardware complexity against underutilization of the register file by narrower types.

The approach introduced in the next section differs in that it focuses on achieving energy and performance benefits by increasing the utilization of the vector register file, improving code density as well as improving fetch & decode speeds. Instead of partitioning physical registers it combines them into wider logical registers. It can do this explicitly or implicitly in form specific instructions or execution modes. Multi-register operations are split into micro-ops and executed in pipelined fashion; hence, the underlying datapath is unaltered and potential performance gains can be achieved by micro-ops overtaking each other. The combination of registers is done on a logical level and performed on partitions of the available register space. This avoids issues arising from register renaming and increases performance by allowing speculative decoding of implicitly mapped registers.

7.7.3 The Basic Idea

Multi-register operations are based on the idea of partitioning a vector register file into a set of equally sized parts and introducing logical registers comprising one physical register per part. In contrast to mapping multiple logical registers into each physical register as described in the previous section (Fig. 7.36a), this approach does not require any datapath alterations. Instead, multi-register operations are split into a number of micro-ops that are executed in pipelined or – if redundant FUs are available – parallel fashion. The development of the Vector Benchmark Suite revealed that the vectorized code segments of the analyzed benchmarks did only require 16 or less of the 32 vector registers provided by ARGON. To improve the utilization of the vector register file, ARGON was extended to support so called double length (L-type) instructions, operating on pairs instead of individual physical registers; i.e. the L-type variant of “vadd V_2, V_1, V_0” refers to “vadd V_2, V_1, V_0’ and “vadd V_{18}, V_{17}, V_{16}’ (Tab. 7.4). Section 6.2.5 shows that depending on the vectorization potential of a given algorithm and workload, the performance of L-type implementations reaches up to 70% of the improvement achieved when operating on datapaths twice as wide.

A similarly improved utilization of the vector register file can be realized by partially unrolling vectorized loops, i.e. replacing individual vector instructions with congeneric sequences. Nevertheless, the higher code density of multi-register operations results in reduced pressure on the L1I as well as the fetch and decode units. Multi-register operations may be used in combination with per-lane predication (Section 6.2.1) and the efficient handling of 0-length micro-ops (Section 7.6) to allow individual loop iterations to operate on a high number of vector elements without the risk of significant performance degradation due to datapath underutilization; i.e. 0-length micro-ops might be substituted according to Tab. 7.3.

Invocation of Multi-Register Operations

There are two distinct methods to invoke multi-register operations:

Explicit:

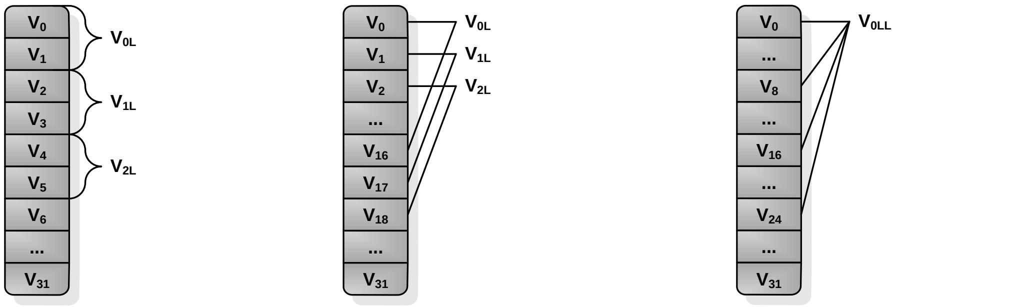
- Each instruction specifies its type as part of its opcode
- Advantage: allows mix of L- & nL-type instructions without additional overhead
- Disadvantage: increased number of encoding bits required

Implicit:

- Additional parameter in control instructions manipulating VL to specify current register type; e.g. vsetVL and vsetVL$_\mathrm{L}$
- Register type of subsequent vector instructions determined by current mode
- Control instructions are not blocking if speculative decoding supported
- Disadvantage: additional state (current mode) to be preserved over function calls, exceptions, context switches, etc.

Similar to ARM NEON and Intel AVX (Section 7.7.2), ARGON provides explicit L- & nL-type instructions to distinguish logical register types. The advantages of this method are low overheads when combining different instruction types and the absence of extra architectural state (e.g. an L-flag). However, the limited encoding space provided by RISC ISAs, such as ARMv7 underlying ARGON, discourages the utilization of additional opcode bits. Hence, future RISC ISA extensions potentially operating on double length, quadruple length or even wider registers are likely to favor implicit invocation methods.

Register layout



(a) Pair adjacent Registers (b) Pair Registers over Half of Register File (c) Group Registers over four equally sized Parts of Register File

Fig. 7.37: Mapping of logical registers for multi-register operations

Fig. 7.37a and 7.37b illustrate two approaches to pair registers for L-type operations. Note that Fig. 7.37 represents an architectural view on register mapping; i.e. the renaming stage of an OoO processors may assign the logical register V_x to any available position in an extended physical register file, a specific ROB entry, any other buffering structure it may posses, or an actual architectural register file (Section 1.3). The pairing of adjacent registers as depicted in Fig. 7.37a is similar to ARMv7 NEON's Q registers (Section 7.7.2). A concern regarding this approach in the context of implicitly invoked multi-register operation is its applicability to speculative decoding. For example, in case of an in-flight control instruction, "vadd V$_2$, V$_1$, V$_0$" might be speculatively decoded to "vadd V$_4$, V$_2$, V$_0$" and "vadd V$_5$, V$_3$, V$_1$". Should the second (speculative) micro-op need to be discarded later on, the operands used within the first micro-op would need to be updated. Fig. 7.37b illustrates an approach more suitable for this case, which pairs registers by striding over register partitions. The corresponding decoder output would be "vadd V$_2$, V$_1$, V$_0$" and "vadd V$_5$, V$_4$, V$_3$". Should the second micro-op need to be discarded, the first could remain unaltered. Similar considerations are valid for larger numbers of combined registers. For instance, Fig. 7.37c depicts an example for the mapping of quadruple length registers based on four adjacent register file partitions of equal size.

Decode Stage

	nL-Type	L-Type

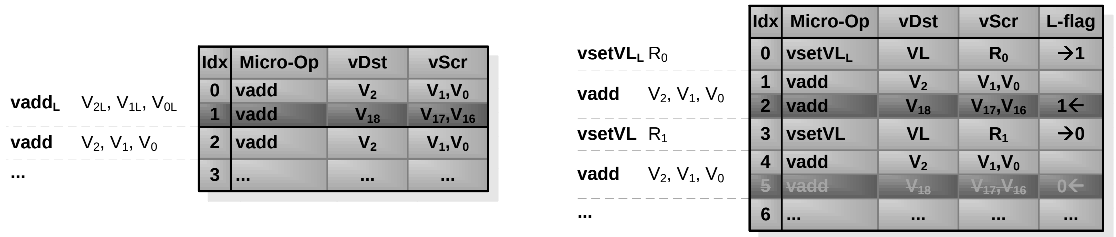
Pseudo Code	vsetVL(...)
	vC = vA + vB
		vsetVL$_L$(...)
		vC = vA + vB

Assembly	vsetVL R_0 vadd V_2, V_1, V_0	vsetVL$_L$ R_0 vadd V_2, V_1, V_0
Decode	L-flag = 0 $V_2 = V_1 + V_0$	L-flag = 1 1st micro-op: $V_2 = V_1 + V_0$ • Same as nL-type 2nd micro-op: $V_{18} = V_{17} + V_{16}$ • Register index cannot exceed V_{15} • Issue dependency on L-type

Tab. 7.4: Example for decoding of implicitly invoked L- & nL-type operations

While the explicit invocation of multi-register operations employs different versions of each instruction, and implicit invocation relies on the same instructions for all cases. Hence, a decoder corresponding to the latter requires some form of status information to distinguish logical register types, e.g. an L-flag for L-/nL-types. A naive OoO implementation would therefore not be able to decode vector instructions while an control instruction that potentially updates said status information is in-flight. Consequently, all those control instruction would have to be blocked, potentially leading to performance degradation due to stalls. A more aggressive implementation may avoid this blocking behavior by employing a register mapping similar to Fig. 7.37b, which treats all vector instructions as L-type and discards speculative components later on if necessary. Tab. 7.4 demonstrates this on an example including the nL- and L-type variants of an addition. Special consideration might be given to instructions with operands ranging from V_{16} to V_{31} as this region of the vector register file is not directly addressable by L-types; hence, corresponding instructions could be decoded non-speculatively to nL-type.

Issue Queue



Idx	Micro-Op	vDst	vScr
0	vadd	V_2	V_1,V_0
1	vadd	V_{18}	V_{17},V_{16}
2	vadd	V_2	V_1,V_0
3	...	...	...

(a) Explicit Invocation

Idx	Micro-Op	vDst	vScr	L-flag
0	vsetVL$_L$	VL	R_0	→1
1	vadd	V_2	V_1,V_0	
2	vadd	V_{18}	V_{17},V_{16}	1←
3	vsetVL	VL	R_1	→0
4	vadd	V_2	V_1,V_0	
5	vadd	V_{18}	V_{17},V_{16}	0←
6	...	...	...	...

(b) Implicit Invocation

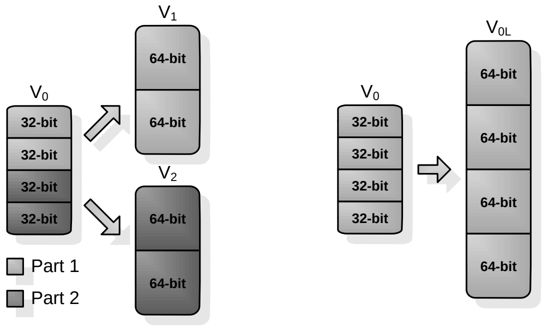
Fig. 7.38: Example for handling of L- & nL-type operations inside the issue queue

Explicitly invoked vector instructions can be decoded non-speculatively and enter the IQ in form of conventional micro-ops (Fig. 7.38a). In contrast, implicitly invoked vector instructions exhibit an additional dependency on the current mode. Fig. 7.38b exemplifies how a sequence of L- & nL-type instructions might be handled based on an L-flag. The speculatively decoded and renamed components of vector instructions (2nd micro-op) are executed as follows:

- L-flag set:
  - Execute like nL type micro-op
- L-flag not set:
  - Discard/commit on issue (or instantaneously if oldest valid IQ entry)
  - Add registers to free list
  - Don't perform any register updates (write-back)

Note that it is not necessary to update the destination registers of discarded micro-ops, because all subsequent speculative instructions depending on them will also be discarded. Discarding/committing instructions immediately allows functional units to be used by other instructions, or left inactive to conserve energy. However, to avoid gaps between valid IQ entries this may have to be delayed until the regular commit. In consequence, the number of instruction issued in parallel may be impacted negatively in the context of systems with small issue windows. To mitigate this, designers could exploit the fact that IQs are usually implemented as ring buffers, and allow the corresponding head pointer to skip multiple previously discarded elements if necessary. In general, the speculative decoding and renaming of vector instructions increases the pressure on processor internal queues, which potential leads to performance degradation for sequences of interleaved L- and nL-type instructions. As frequent alterations of VL and VM are generally discouraged, such sequences are considered rare.

Type Promotion / Demotion



(a) nL-Type

(b) L-Type

Fig. 7.39: Example for promotion of 32-bit to 64-bit elements in packed register format

The padding performed following data type promotions within packed vector registers increases the number of bits required per vector element. Consequently, architectures without the support of L-types either require two separate instructions to promote the lower/upper half of a register, or one instruction writing back to two output registers; the latter being unlikely, due to the high costs associated with vector result buses (Fig. 7.39a). Note that the same is the case for the mapping scheme described in Fig. 7.36b for ARMv8 NEON and the different Intel AVX versions. In contrast, architectures with L-type support or ARMv7 NEON like register mapping (Fig. 7.36a) may use a single instruction to cast between nL- and L-type, or D and Q registers, respectively (Fig. 7.39b). In case of implicit invocations, these cast instructions may be accompanied with an additional control bit to indicate if a transition from nL- to L-types or vice versa (L-flag update) is desired . This could further increased code density and provide performance benefits for algorithms that exhibit repeated type conversions.

7.7.4 Concluding Remarks

This section introduced the idea of partitioning a vector register file into a set of equally sized parts and operating on logical registers comprising one physical register per part. So called multi-register operations do not increase hardware complexity, as they are split into a number of micro-ops which are then executed in pipelined or – if redundant FUs are available – parallel fashion on the same datapath utilized by conventional operations. Their performance benefits originate from improved vector register file utilization, a higher code density, and a reduced pressure on fetch and decode circuitry. Logical register types may be addressed either explicitly or implicitly; i.e. by addressing them directly within opcodes or by setting corresponding flags. Speculative decoding allows implicitly invoked multi-register instructions to be dispatch even though an update to said flags is in flight. The miss-speculation penalties of this mechanism are reduced by register mapping that allows speculatively decoded micro-ops to be discarded without affecting their non-speculative counterpart. Multi-register operations may be combined with per-lane predication (Section 6.2.1) and mechanism that efficiently handle 0-length instructions (Section 7.6) to allow individual iterations of vectorized loops to operate on a high number of vector elements without the risk of significant performance degradation due to datapath underutilization.

Analyses in Section 6.5.2 showed that algorithms that operate only on a small number of vector registers and do not exhibit dependencies on high latency operations may employ multi-register instructions to achieve significant speedups without the need for wider data paths. However, for algorithms that require a high number of registers to be preserved between loop iterations or function calls, this feature may yield lower performance, due to the need for long latency vector fills and spills.

7.8 Concluding Remarks

This chapter introduced a number of microarchitecture optimizations derived from the preceding analysis of the ARGON ISA extension. Some of the concepts presented here were directly fed back into ARGON and further investigated in Section 6.5. This includes the CompD instruction, multi-register operations in form of double length instructions, and Scan Partitioning. The latter is basis for the timing profiles in Section 6.4.2, e.g. FullDP and HalfDP representing a full and half sized datapath including single- and two-part scans, respectively. The SerialScan profile describes a combination of a conventional datapath and fully serialized scans. At this point, the presented results are considered sufficient to approximate the performance achievable by various datapath configurations. More detailed analyzes should be reserved for later design stages that provide better constrained parameters. This includes analyzes concerning accumulated benefits due to the interaction between the proposed optimizations, too. For instance, how the efficient handling of 0-length micro-ops can be extended to an intra-instruction granularity when operating on a partition datapath or in the context of multi-register operations.

The lack of quantitative energy estimates in this chapter is based on the limited capabilities of the ARGON simulation framework. It was design for configurability, to allow evaluations on different points of the design space, and automation, to reduce turnover times and potential error sources when introducing new or modifying existing features. The implementation of features introduced here and in the previous chapter significantly impact nearly all processor components. However, the development of an all encompassing RTL model is out of the scope of this work. In particular, RTL simulations are considered not flexible enough to accurately model the wide range of design points investigated here. A second limitation of the current simulation framework is the inability to speed up operations at run time depending on per-lane predicates. Hence, performance analyses regarding the acceleration of scans on incomplete vectors and the efficient handling of 0-length micro-ops are limited to estimates based on the average vector length / number of active elements utilized (Sections 7.5 and 7.6, respectively).

8 | Conclusions and Future Work

Modern mobile devices operate under severe energy constrains, but are required to provide increasing levels of performance to process a continuously growing number of computation intensive applications. This resulted in design efforts focused on exploiting increasing amounts of instruction- and data-level parallelism as indicated by a transition from single-issue in-order to superscalar out-of-order execution units and from uni- to multi-core processors on high-end systems. However, in contrast to media or signal processing algorithms, general purpose code usual does not exhibit sufficient regularity to offset the latency and energy costs associated with data-movements between CPUs and throughput accelerators such as specialized ASICs, FPGAs, DPSs or GPUs. This thesis aims to expand the applicability and energy efficiency of vector ISA extensions as a way to extract fine-grain data-level parallelism. It considers the high memory bandwidth demands and potential energy costs associated with vector processing when focusing on the efficient execution of parallel memory accesses and the optimization of vector functional units. Besides saving energy due to an improved efficiency, it attempts to reduce computation times to increase the proportion of time a processor may spend in low power states. The following sections summarize the contributions introduced in the preceding chapters, evaluate them in context of this thesis's objectives (Section 8.1), and briefly discuss a number of areas for future research to improve upon and extend them (Section 8.1.1).

8.1 Summary of Contributions

The initial objective of this thesis was to identify to what extend parallelization affects the memory access behavior of general purpose algorithms, and derive implications for the design of appropriately adapted memory systems. It is met by a two part analyses of memory access patterns presented in Chapter 3. The most notable conclusions based on the reference stream between processor and L1D are that:

- Approximately 50% of loads are immediately followed by requests to the same cache lane. This ratio increases to more than 80% when up to four intermediate accesses to different cache lines are permitted.
- Similar analyses on a page granularity result in ratios of 70% and 95%.

The naive vectorization of the underlying benchmarks reveals that:

- Unit strides and non-unit strides are predominant when compared to indexed memory accesses.
- The majority of vector accesses do not cross page boundaries.
- An eight element wide vector load references in average four different lines, each line being referenced by two - most likely consecutive - elements.

Using these insights for the design of a scalable high level cache interface, capable of providing the desired degree of parallelism within an energy constrained system, leads to the

Multiple Access Low Energy Cache (MALEC). It exploits the observations that consecutive memory references tend to access the same page of memory and that loads & stores are likely to access the same cache line, if a certain number of intermediate accesses to different lines is permitted. Under the term Page-Based Memory Access Grouping, it implements mechanisms to share memory address translation results between multiple loads and stores, simplify store and merge buffer lookups, and share L1 data among loads accessing the same cache line (Chapter 4). This enables a single-ported TLB and L1D to achieve performance similar to multi-ported structures, while exhibiting significantly lower energy consumption. MALEC's energy efficiency is further improved by extending it with Page-Based Way Determination, which refers to the concept of holding way information on recently accessed cache lines in small memories that are closely coupled to TLB lookups (Chapter 5). By exploiting the restriction to accesses only one page per cycle, way information corresponding to all memory accesses in any given cycle can be provided without the need for a dedicated lookup structure. The concept is highly scalable in terms of parallel memory accesses and allows approximately 94% of the analyzed memory references to bypass tag-arrays and directly access one specific way of a set-associated L1D. The example of MALEC shows how information obtained from analyses of memory access patterns can be exploited to mitigate the high memory bandwidth demands associated with vector processing in an energy efficient manner.

The thesis furthermore established clear statements regarding the performance impact of advanced SIMD features in the context of vector ISA extensions utilized for the execution of general purpose algorithms. For this purpose, Chapter 6 introduces an ARMv7 NEON based vector ISA extension (ARGON), a parameterizable simulation framework and a corresponding benchmark suite. It demonstrates how per-lane predication, indexed memory accesses, scans, segmented scans and wider datapaths improve performance by increasing the vectorizability and datapath utilization exhibited by general purpose code. Furthermore, it deduces several design guidelines for future vector ISA extensions and processors based on their respective emphasis on scalar or vectorized computation.

The final objective of this work was the development of techniques to improve the applicability of advanced SIMD features within energy constrained systems, by reducing the complexity and improving the utilization of functional units associated with them. Chapter 7 addresses this by proposing the following set of architectural optimizations:

- **The Compute Descriptor Instruction**
  - Computes hardware interpretable masks for segmented operations based on pointer arrays. Used in conjunction with segmented scans to improve SIMD datapath utilization by allowing arbitrary length segments within vectors to be processed in parallel; hence, nested loops with limited iteration counts for the innermost loop may be collapsed.
- **Scan Partitioning**
  - Allows designers to target specific performance/energy trade-offs for scan operations depending on their frequency of use and implementation complexity.

Generally increases energy efficiency by employing narrow FUs and performing fewer operations per scan.

- **Efficient Handling of Masks for Vector Predication and Segmentation**
  - Extends the control logic of existing multiplexing structures to handle predicated data-movement and insertion. Does not impose additional latency and reduces energy and area consumption by removing need for dedicated routing networks.
- **Acceleration of Scans on incomplete Vectors**
  - Increases probability of scans being able to omit the execution of specific steps. Potentially accelerates execution of predicated and segmented scans and reduces energy consumption by omitting unnecessary computation steps.
- **Efficient Handling of Zero Length Vector Micro-Ops**
  - Increases energy efficiency and performance of instructions that - due to per-lane predication - would operate exclusively on inactive elements. Corresponding micro-ops are identified and substituted based on their type either with NOPs or vector moves.
- **Multi-Register Operations**
  - Vector register file partitioned into equally sized parts. Operate on logical registers comprised of one physical register per part. Split multi-register operations into micro-ops employing the existing datapath. Performance benefits originate from improved vector register file utilization, higher code density, and reduced pressure on fetch and decode circuitry.

In summary, the contributions presented in this thesis provide insights into the impact of parallelization of general purpose algorithms on high level memory interfaces, and vector functional unit design. The introduced proposals represent guidelines for the development and optimization of upcoming vector ISA extensions and the computation hardware associate with them. The conclusions drawn in this thesis are based on representative workloads subjected to cycle accurate simulations, which were verified against estimates derived from existing hardware components. It is hoped that the presented work will influence future academic and industrial development in this area of research.

8.1.1 Future Work

There are several avenues of research which lead on from this work:

Energy Analyses of proposed Architecture Optimizations

- The simulation framework used for the evaluation of ARGON in Chapter 6 is currently limited to performance estimates. RTL simulations based on the optimizations proposed in Chapter 7 could increase the level of confidence invested in them and help to identify potential issues arising from specific implementation variants.

Exploration of complex SIMD Features

- The example of segmented scans - evaluated and expanded upon in Chapter 6 and Chapter 7, respectively - demonstrates how complex vector instructions can provide performance and energy benefits by increasing vectorizability and datapath utilization. Other instructions of similar complexity are thinkable. They are likely to involve limited degrees of datapath divergence and/or interactions between individual lanes.

Advanced Vector Compilation Techniques

- The benchmarks introduced in Chapter 6 were vectorized based on ARGON intrinsics. While certain features such as per-lane predication and indexed memory access often permit direct translations between scalar and vectorized code segments, others may impose significant programming challenges. In particular segmented scans may require indexes to be preprocessed, data structures to be re-arranged and/or nested loops to be split or completely restructured. One approach to these challenges would be auto-vectorization compilers as described in [72, 73]. Alternatively, light wait tools to assist programmers by suggesting certain data and loop structures may achieve even better results in the context of hand optimized code as it is used to accelerate performance critical program phases [78].

Performance Evaluation for Server Workloads

- This thesis focused on designs based on mobile platforms. However, SIMD features are also employed alongside many-core designs and throughput accelerators to implement high-end servers and HPCs. Hence, analyses of SIMD features and programming paradigms in the context of very wide datapaths and homo- as well as heterogeneous processing is required. These could among other things yield insights into specialized memory interfaces and reconfigurable datapaths. The latter represents the idea of a runtime adjustable datapath width that increases energy efficiency by disabling (e.g. power gating) underutilized lanes and partitions vector instructions to mitigate the impact of short code segments exhibiting high degrees of vectorization.

Latency free Merging of Vector Accesses on VIPT Caches

- MALEC was designed in the context of PIPT caches as they are common in modern energy oriented microprocessors and avoid issues associated with aliasing, homonyms, ect. However, more performance oriented systems may implement VIPT caches to improve L1 access latencies for the cost of reduced energy efficiency. As MALEC overlaps comparator delays with TLB accesses, alternative merging schemes are desirable to avoid additional latencies within the critical path. One option may perform merging in parallel to the address generation step of vector memory references. For example, unit strides may determine mergeable accesses using the base address and data type width; non-unit strides may perform approximate multiplications or employ lookup structures for the most common stride distances; indexed memory accesses may perform comparisons on partial element offsets and correct results using specific bits of the base address. While this approach would permit intra-instruction merging, it does not permit comparisons between multiple vector and/or scalar memory references.

LSQ Design for Vector ISA Extensions

– One issue encountered during the design of the ARGON evaluation framework is the need for a load-store queue (LSQ) capable of efficiently handling the high number of memory references associated with vector instructions. In particular, the current framework interprets vector loads/stores as a multitude of micro-ops to be allocated to individual LSQ entries. A potential optimization would be the allocation of LSQ entries based on virtual page IDs and the provision of multiple slots per entry to allow merging of accesses. This would not just permit intra-, but also inter-instruction merging as performed by MALEC. Abella et al. proposed a similar design in an attempt to reduce the energy consumption of fully associated LSQs in the context of scalar processors [101].

Distribution of Elements read from Memory

– MALEC employs priority multiplexers to align data read from memory, select portions desired by specific lanes and combine those results with data read from SB and MB entries (Fig. 4.2). While this is common practice within current designs, it does not scale well with increasing datapath widths. Hence, further research into alternative designs such as skew multiplexers with delay registers, rotational buffers and crossbars is required.

A | Appendix

A.1 The SPEC2000 and MediaBench2 Benchmark Suites

The following two tables give an overview of SPEC2000 and MediaBench2 benchmarks including information on language, category, instruction count, and simulation points. The presented instruction counts were obtained during the analysis of program intervals as described in Section 3.2.

Suite	Benchmark	Language	Category	Instruction Count
SPEC2000 Integer	gzip_source	C	Compression	1,613,274,088
	vpr_place	C	FPGA Circuit Placement and Routing	121,066,832,828
	gcc_166	C	C Programming Language Compiler	41,330,744,832
	mcf	C	Combinatorial Optimization	52,426,070,381
	crafty	C	Game Playing: Chess	230,720,837,992
	parser	C	Word Processing	542,082,919,539
	eon_cook	C++	Computer Visualization	68,378,382,169
	perlbmk_diffmail	C	PERL Programming Language	36,330,549,028
	gap	C	Group Theory, Interpreter	226,974,646,908
	vortex1	C	Object-oriented Database	141,818,919,039
	bzip2_source	C	Compression	101,981,749,633
	twolf	C	Place and Route	705,831,986,548
SPEC2000 Floating Point	wupwise	Fortran77	Physics/QuantumChromodynamics	381,949,172,941
	swim	Fortran77	ShallowWaterModeling	406,058,835,584
	mgrid	Fortran77	Multi-gridSolver:3DPotentialField	676,743,422,795
	applu	Fortran77	Parabolic/EllipticPartialDifferentialEquations	526,219,649,901
	mesa	C	3-DGraphicsLibrary	301,059,598,517
	galgel	Fortran90	ComputationalFluidDynamics	310,594,680,929
	art470	C	ImageRecognition/NeuralNetworks	78,245,876,171
	equake	C	SeismicWavePropagationSimulation	126,963,686,250
	facerec	Fortran90	ImageProcessing:FaceRecognition	-
	ammp	C	ComputationalChemistry	343,424,392,946
	lucas	Fortran90	NumberTheory/PrimalityTesting	283,913,580,818
	fma3d	Fortran90	Finite-elementCrashSimulation	444,358,125,919
	sixtrack	Fortran77	HighEnergyNuclearPhysicsAcceleratorDesign	535,010,729,906
	apsi	Fortran77	Meteorology	428,987,895,686
MediaBench2	cjpeg	C	Image Compression Encoder	59,300,327
	djpeg	C	Image Compression Dencoder	32,827,277
	h263enc	C	Videa Compression Encoder	24,144,735,027
	h263dec	C	Videa Compression Dencoder	844,101,547
	h264enc	C	Videa Compression Encoder	-
	h264dec	C	Videa Compression Dencoder	-
	jpg2000enc	C	Image Compression Encoder	1,055,846,103
	jpg2000dec	C	Image Compression Dencoder	685,499,296
	mpeg2enc	C	Videa Compression Encoder	34,183,685,806
	mpeg2dec	C	Videa Compression Dencoder	994,836,515
	mpeg4enc	C	Videa Compression Encoder	1,898,878,677
	mpeg4dec	C	Videa Compression Dencoder	251,828,628

Tab. A.1: Overview: SPEC2000 and MediaBench2 benchmark suites

vpr place	gcc 166	mcf	crafty	parser	eon cook	perlbmk diffmail	gap	vortex1	bzip2 source	twolf
114 2%	33 20%	20 4%	187 8%	244 3%	35 12%	1 3%	85 3%	3 8%	97 7%	206 5%
105 4%	35 24%	23 12%	130 10%	467 7%	59 24%	21 28%	136 8%	45 6%	98 12%	687 6%
120 1%	28 27%	9 25%	58 12%	10 2%	43 19%	2 3%	106 2%	93 6%	27 47%	413 5%
63 9%	9 5%	32 12%	208 10%	512 4%	39 16%	13 3%	224 1%	32 4%	7 14%	0 0%
84 9%	4 7%	2 10%	39 21%	391 6%	6 19%	24 3%	187 3%	105 13%	2 21%	313 7%
0 1%	17 10%	48 13%	61 17%	24 17%	11 10%	35 31%	192 2%	86 27%		541 4%
46 5%	0 5%	22 21%	197 10%	33 10%		0 3%	105 2%	12 9%		458 5%
110 5%	30 2%	13 4%	227 3%	293 2%		10 28%	218 4%	46 2%		363 10%
73 10%			206 10%	533 3%			179 1%	42 9%		253 7%
24 5%				451 11%			199 0%	126 16%		505 8%
18 5%				350 13%			163 4%			17 5%
117 3%				169 12%			195 2%			574 5%
4 6%				315 9%			96 8%			51 4%
34 4%							39 8%			110 5%
29 4%							189 1%			80 4%
98 9%							64 3%			645 5%
40 5%							206 4%			611 5%
54 7%							0 0%			147 4%
11 6%							223 0%			173 5%
							49 11%			
							17 3%			
							140 11%			
							178 2%			
							5 5%			
							215 2%			
							166 4%			
							21 3%			
							116 2%			
							142 2%			

wupwise	swim	mgrid	applu	mesa	galgel	art470	equake	facerec	ammmp	lucas	fma3d	sixtrack	apsi
201 18%	63 2%	406 0%	464 5%	276 7%	113 1%	11 5%	44 8%	0 1%	174 4%	227 5%	9 3%	0 0%	219 2%
210 11%	0 0%	413 5%	27 6%	268 9%	164 6%	12 1%	16 3%	35 8%	284 5%	221 4%	28 1%	206 9%	263 5%
213 16%	82 40%	416 6%	0 0%	154 2%	107 9%	76 4%	125 6%	30 3%	92 4%	154 7%	255 12%	460 7%	362 3%
4 1%	53 32%	217 22%	319 4%	130 4%	94 7%	1 6%	90 10%	98 2%	76 7%	127 5%	2 1%	386 1%	265 5%
294 20%	35 1%	98 3%	282 6%	132 2%	173 3%	72 6%	96 8%	90 3%	162 3%	0 1%	399 9%	443 6%	28 2%
0 0%	124 1%	417 6%	59 3%	299 1%	305 4%	29 18%	6 5%	14 17%	1 0%	2 0%	260 1%	61 4%	352 6%
2 1%	1 0%	412 5%	501 7%	15 7%	129 2%	44 5%	19 2%	27 18%	225 1%	108 4%	30 0%	5 1%	344 3%
222 21%	64 16%	536 3%	451 4%	0 0%	242 0%	7 3%	1 2%	34 9%	149 7%	38 2%	359 7%	313 3%	299 3%
21 2%	70 5%	671 4%	67 4%	113 2%	165 4%	14 14%	105 9%	54 8%	85 3%	164 6%	17 2%	454 6%	400 2%
140 11%	217 1%	81 2%	187 4%	242 1%	127 17%	55 5%	39 10%	96 2%	163 2%	122 5%	118 33%	493 8%	150 4%
	235 1%	509 8%	99 4%	181 7%	236 12%	17 4%	51 9%	97 8%	0 0%	53 6%	440 10%	209 8%	66 4%
		508 7%	253 4%	243 5%	49 9%	0 1%	22 1%	39 6%	184 8%	184 4%	361 16%	388 2%	7 4%
		423 23%	278 6%	196 7%	234 8%	46 3%	21 6%	63 10%	165 10%	226 4%	370 2%	10 0%	356 13%
		649 1%	11 5%	150 2%	296 3%	57 8%	93 17%	80 2%	56 6%	207 3%	401 2%	229 3%	195 5%
		20 5%	127 4%	266 5%	122 14%	16 6%	31 5%	74 5%	135 5%	88 7%	403 2%	425 4%	232 2%
			110 3%	269 9%		43 3%			132 4%	24 2%		96 15%	74 3%
			371 6%	17 7%		41 5%			154 3%	232 5%		280 8%	133 4%
			182 6%	260 9%		71 3%			250 14%	31 1%		434 7%	9 9%
			375 7%	284 4%					70 3%	203 3%		320 8%	71 2%
			385 5%	143 2%					82 10%	77 3%			334 6%
			183 6%	39 7%						92 3%			0 0%
										27 2%			220 4%
										100 4%			103 6%
										13 1%			
										121 3%			
										253 3%			
										189 5%			
										136 2%			

- SPEC-Int and SPEC-FP benchmarks in top and bottom part of table, respectively
- Columns divided in simulation points (left) and corresponding weights (right)
- Highlighted cells indicate simulation points with highest weights (earliest chosen for simulations in Chapter 3, to reduce computation time required for snapshot generation)
- *gzip_sourc* executes only 1,613,274,088 instructions; hence, only one full analysis interval → first 1 billion

Tab. A.2: Simulation points and weights for SPEC2000 benchmarks

A.2 Further Analysis on Consec. Accesses per Cache Block

Section 3.5.1 investigates the number of read accesses to consecutive addresses received by a first level cache. Fig. A.1 includes corresponding graphs concerning write access. It can be observed that writes are even more suitable for techniques that allow consecutive accesses to the same cache line to be merged. A common method to exploits the spatial locality of stores is the implementation of a merge buffer (Section 1.1.2). Those buffers are used by the simulation environment employed in Chapters 4 and 5.

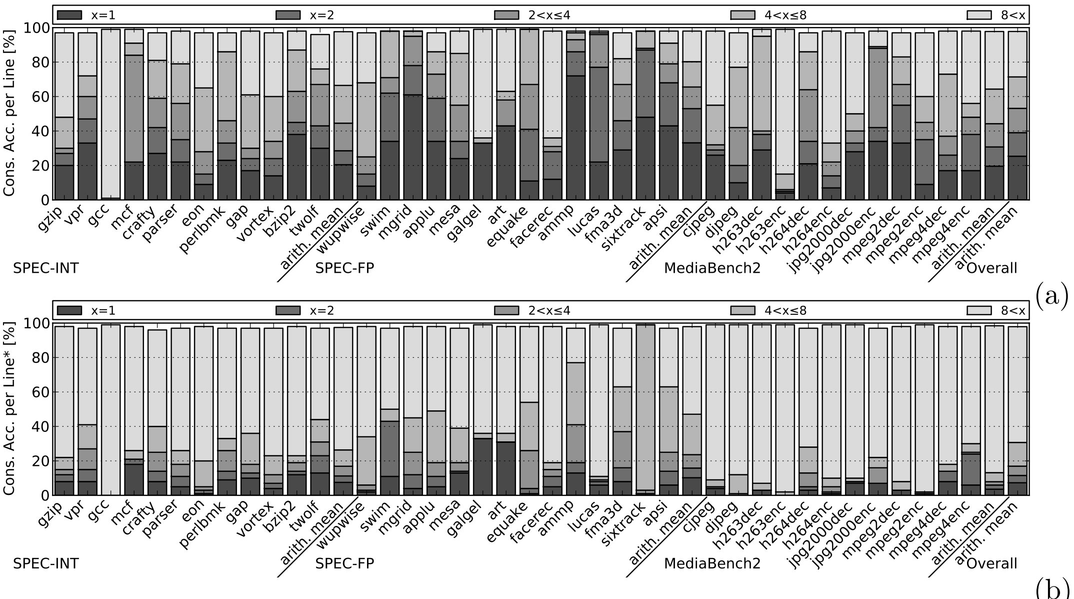
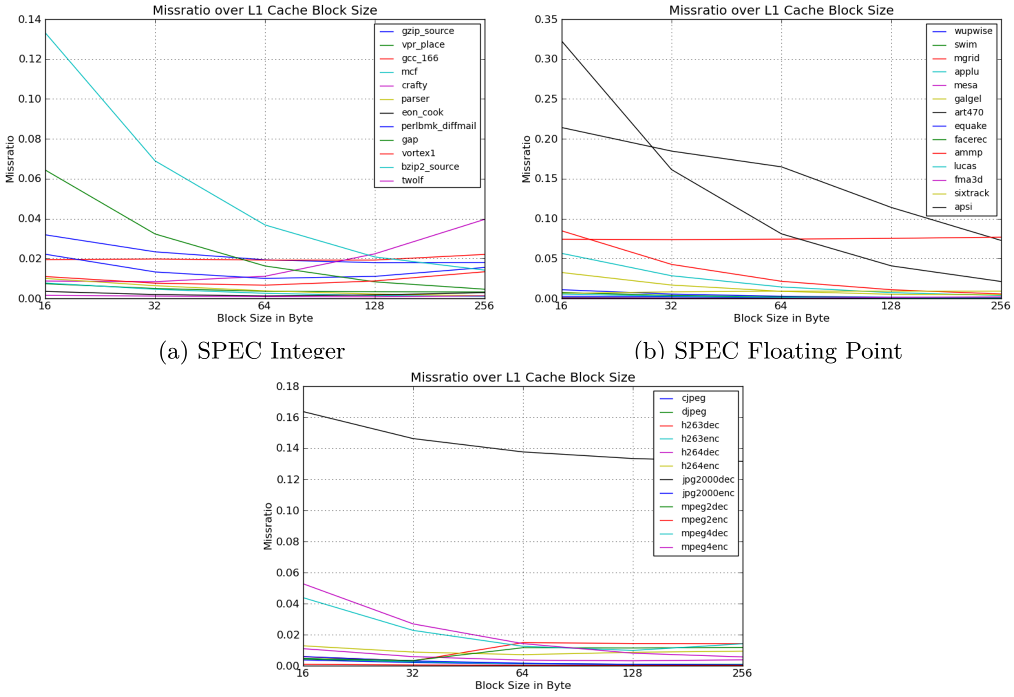


Fig. A.1: Consecutive write accesses per cache block

Section 3.5.3 investigates the dependency between cache line size and the number of consecutive accesses per cache line. However, the size of cache lines also affects other cache parameters. For this reason Fig. A.2 shows its relation to the L1D miss ratio for all SPEC2000 and MediaBench2 benchmarks. The graphs clearly show the positive effect of wider lines for the majority of programs up to a value of 128 byte. This effect can be explained with the implicit prefetching associated with every line received from lower cache levels. Each line does not only contain the requested address, but also those in close proximity to it. Due to the high spatial locality exhibited by most applications, this results in decreased miss ratios. However, as all analyses are performed for the same overall cache size, very long lines tend to evict useful data and thereby reverse this trend.



(a) SPEC Integer

(b) SPEC Floating Point

(c) MediaBench2

Fig. A.2: Miss ratio over L1D cache block size

A.3 Further Analysis on the Influence of Vectorization on Memory Access Patterns

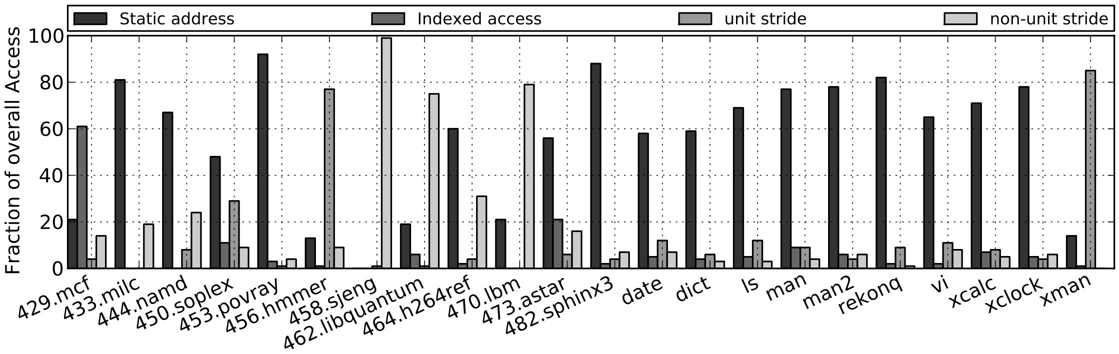


Fig. A.3: Distribution of memory access patterns (stores)

Section 3.7.2 analyses load access patterns for selected SPEC2006 benchmarks and a small set of linux based programs. Fig. A.3 illustrates results of a similar analysis for store accesses. The observable patterns are very similar to those in Fig. 3.11 of Section 3.7.2. As it is more efficient to optimize memory systems for loads rather than for stores (Section 3.3), the above graph is only listed for reference and will not be further discussed here.

References

[1] S. Borkar and A. A. Chien, “The future of microprocessors,” *Commun. ACM*, vol. 54, no. 5, 2005.

[2] W. Wulf and S. McKee, “Hitting the memory wall: implications of the obvious,” *ACM SIGARCH Comput. Archit. news*, vol. 23, no. 1, 1995.

[3] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design*, 4th ed. Burlington, MA: Morgan Kaufmann, 2009.

[4] P. V. Bolotoff. (2007, April) Functional Principles of Cache Memory. [Online]. Available: http://alasir.com/articles/cache_principles/

[5] Intel Corporation, “Intel 64 and IA-32 Architectures Optimization Reference Manual,” Tech. Rep. July, 2013.

[6] Advanced Micro Devices, “Software Optimization Guide for AMD Family 15h Processors,” Tech. Rep. 47414, 2011.

[7] J. L. Hennessy and D. A. Patterson, *Fundamentals of Computer Design*, 3rd ed. Morgan Kaufmann, 2007.

[8] ARM Limited, “Cortex -A15 MPCore,” Tech. Rep., 2012.

[9] H. Ghasemzadeh, S. Mazrouee, and M. Kakoee, “Modified Pseudo LRU Replacement Algorithm,” in *ECBS*, 2006.

[10] A. Tanenbaum and A. S. Woodhull, *Operating systems: design and implementation*, 3rd ed. Prentice Hall, 2006.

[11] A. Grama, A. Gupta, G. Karypis, and V. Kumar, *Introduction to Parallel Computing*, 2nd ed. Harlow, England: Pearson Education Limited, 2003.

[12] J. M. Crichlow, *An introduction to distributed and parallel computing*, 2nd ed. Upper Saddle River, New Jersey: Prentice Hall, 1996.

[13] J. Smith and G. Sohi, “The microarchitecture of superscalar processors,” *Proc. IEEE*, vol. 83, no. 12, 1995.

[14] D. Culler, J. Singh, and A. Gupta, *Parallel computer architecture: a hardware/software approach.* San Francisco, CA: Morgan Kaufmann Publishers, Inc, 1998.

[15] K. Kobayashi *et al.*, “A vector-pipeline DSP for low-rate videophones,” in *ASP-DAC*, 2001.

[16] J. Redford, B. Bersack, M. Monk, F. Huettig, and D. Fitzgerald, “A Vector DSP For Imaging,” in *CICC*, 2002.

[17] E. Matu, H. Seidel, T. Limberg, P. Robelly, and G. Fettweis, “A GFLOPS Vector-DSP for Broadband Wireless Applications,” in *CICC*, Sep. 2006.

[18] P. Westermann, G. Beier, H. Ait-Harma, and L. Schwoerer, “Performance analysis of W-CDMA algorithms on a vector DSP,” in *ECCSC*, Jul. 2008.

[19] M. Van Der Horst, K. Van Berkel, J. Lukkien, and R. Mak, “Recursive Filtering on a Vector DSP with Linear Speedup,” in *ASAP*, 2005.

[20] C.-h. Lin and A.-y. Wu, “Mixed-scaling-rotation CORDIC (MSR-CORDIC) algorithm and architecture for high-performance vector rotational DSP applications,” *Trans. Circuits Syst. I*, vol. 52, no. 11, 2005.

[21] J. Markoff, “Intel’s Big Shift After Hitting Technical Wall,” The New York Times, 17. May 2004.

[22] G. Amdahl, “Validity of the single processor approach to achieving large scale computing capabilities,” in *AFIPS*, 1967.

[23] R. M. Russell, “The CRAY-1 computer system,” *Commun. ACM*, vol. 21, no. 1, 1978.

[24] Intel Corporation, “Intel Architecture Instruction Set Extensions Programming Reference,” Tech. Rep. July, 2013.

[25] M. Kamble and K. Ghose, “Analytical energy dissipation models for low power caches,” in *ISLPED*, 1997.

[26] C.-L. Su and A. M. Despain, “Cache designs for energy efficiency,” in *HICSS*, 1995.

[27] J. A. Rivers, G. S. Tyson, E. S. Davidson, and T. M. Austin, “On High-Bandwidth Data Cache Design for Multi-Issue Processors,” in *MICRO*, 1997.

[28] J. Kin, M. Gupta, and W. H. Mangione-Smith, “The filter cache: an energy efficient memory structure,” in *MICRO*, 1997.

[29] D. Nicolaescu, A. Veidenbaum, and A. Nicolau, “Reducing data cache energy consumption via cached load/store queue,” in *ISLPED*, 2003.

[30] Y. Chang and M. Lan, “Two new techniques integrated for energy-efficient TLB design,” *Trans. VLSI Syst.*, vol. 15, no. 1, 2007.

[31] T. M. Austin and G. S. Sohi, “High-bandwidth address translation for multiple-issue processors,” in *ISCA*, 1996.

[32] J. Fu and J. Patel, “Data prefetching strategies for vector cache memories,” in *IPPS*, 1991.

[33] C. Batten, R. Krashinsky, S. Gerding, and K. Asanovic, “Cache Refill/Access Decoupling for Vector Machines,” in *MICRO*, 2004.

[34] T. Juan, J. J. Navarro, and O. Temam, "Data caches for superscalar processors," in *ICS*, 1997.

[35] S. Tong and Y. Qing, "A comparative analysis of cache designs for vector processing," *IEEE Trans. Comput.*, vol. 48, no. 3, 1999.

[36] B. R. Rau, "Pseudo-randomly interleaved memory," in *ISCA*, 1991.

[37] E. G. Hallnor and S. K. Reinhardt, "A fully associative software-managed cache design," in *ISCA*, 2000.

[38] J. Cho, H. Chang, and W. Sung, "An FPGA based SIMD processor with a vector memory unit," in *ISCAS*, 2006.

[39] S. McKee, S. Moyer, W. Wulf, and C. Hitchcock, "Increasing memory bandwidth for vector computations," in *PLSA*, 1994.

[40] R. Espasa *et al.*, "Tarantula: a vector extension to the alpha architecture," in *ISCA*, 2002.

[41] A. Seznec and R. Espasa, "Conflict-Free Accesses to Strided Vectors on a Banked Cache," *IEEE Trans. Comput.*, vol. 54, no. 7, 2005.

[42] B. Calder, D. Grunwald, and J. Emer, "Predictive sequential associative cache," in *HPCA*, 1996.

[43] K. Inoue, T. Ishihara, and K. Murakami, "Way-predicting set-associative cache for high performance and low energy consumption," in *ISLPED*, 1999.

[44] M. Powell, A. Agarwal, T. Vijaykumar, B. Falsafi, and K. Roy, "Reducing set-associative cache energy via way-prediction and selective direct-mapping," in *MICRO*, 2001.

[45] R. Min, W. Jone, and Y. Hu, "Location cache: a low-power L2 cache system," in *ISLPED*, 2004.

[46] G. Keramidas, P. Xekalakis, and S. Kaxiras, "Applying decay to reduce dynamic power in set-associative caches," in *HiPEAC*, 2007.

[47] A. Ma, M. Zhang, and K. Asanovic, "Way memoization to reduce fetch energy in instruction caches," in *ISCA Work. Complexity- Eff. Des. ISCA*, vol. 20, no. June. Citeseer, 2001.

[48] E. Rotenberg, S. Bennett, and J. Smith, "Trace cache: a low latency approach to high bandwidth instruction fetching," in *MICRO*, 1996.

[49] D. Nicolaescu, A. Veidenbaum, and A. Nicolau, "Reducing Power Consumption for High-Associativity Data Caches in Embedded Processors," in *DATE*, 2003.

[50] M. Ghosh, E. Ozer, S. Ford, S. Biles, and H. Lee, "Way guard: a segmented counting bloom filter approach to reducing energy for set-associative caches," in *ISLPED*, 2009.

[51] C. Zhang, F. Vahid, J. Yang, and W. Najjar, "A Way-Halting Cache for Low-Energy High-Performance Systems," *TACO*, vol. 2, no. 1, 2005.

[52] Y. Chang and S. Ruan, "Sentry tag: an efficient filter scheme for low power cache," in *Proc. ACSAC'2002*, 2002.

[53] Cray Research Inc., "The CRAY-2 Computer System," Tech. Rep., 1985.

[54] Cray Research Inc., "The CRAY X-MP Series of Computer Systems," Tech. Rep., 1985.

[55] Cray Research Inc., "The CRAY X-MP Computer Systems Functional Description Manual," Tech. Rep., 1990.

[56] P. A. Agarwal *et al.*, "Cray X1 Evaluation Status Report," ORNL, Tech. Rep., 2004.

[57] Cray Research Inc.,, "Cray X2 Vector Processing Blade," Tech. Rep., 2008.

[58] Intel Corporation, "Intel Xeon Phi Coprocessors: Developers's Quick Start Guide," Tech. Rep., 2012.

[59] NVIDIA Corporation, "NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110," Tech. Rep., 2012.

[60] Advanced Micro Devices Inc., "AMD FirePro S10000," Tech. Rep., 2013.

[61] H. Meuer, E. Strohmaier, J. Dongarra, and H. Simon. (2013, Nov.) Top 500 Supercomputer Sites. [Online]. Available: www.top500.org

[62] CHREC. (2014, Feb.) CHREC: NSF Center for High-Performance Reconfigurable Computing. [Online]. Available: www.chrec.org

[63] Krste Asanovic, "Vector Microprocessors," Ph.D. dissertation, University of California, Berkeley, 1998.

[64] C. Kozyrakis and D. Patterson, "Scalable vector processors for embedded systems," *IEEE Micro*, vol. 23, no. 6, 2003.

[65] R. Krashinsky, C. Batten, and K. Asanović, "Implementing the scale vector-thread processor," *TODAES*, vol. 13, no. 3, 2008.

[66] P. Yiannacouras, J. Steffan, and J. Rose, "VESPA: portable, scalable, and flexible FPGA-based vector processors," in *CASES*, 2008.

[67] M. Naylor, P. J. Fox, A. T. Markettos, and S. W. Moore, "Managing the FPGA memory wall: Custom computing or vector processing?" in *FPL*, 2013.

[68] ARM Limited, "Introducing NEON," Tech. Rep., 2009.

[69] ARM Limited, “ARM Compiler toolchain: Assembler Reference,” Tech. Rep., 2011.

[70] L. Seiler *et al.*, “Larrabee: A Many-Core x86 Architecture for Visual Computing,” *ACM Trans. Graph.*, vol. 27, no. 3, 2008.

[71] V. Govindaraju, T. Nowatzki, and K. Sankaralingam, “Breaking SIMD shackles with an exposed flexible microarchitecture and the access execute PDG,” in *PACT*, 2013.

[72] D. Nuzman and R. Henderson, “Multi-platform Auto-vectorization,” in *CGO*, 2006.

[73] K. Stock, L.-N. Pouchet, and P. Sadayappan, “Using machine learning to improve automatic vectorization,” *TACO*, vol. 8, no. 4, 2012.

[74] S. Campanoni, T. Jones, G. Holloway, W. Gu-YEON, and D. Brooks, “The HELIX project: overview and directions,” in *DAC*, 2012.

[75] Cray Research Inc. (2014, Feb.) Optimizing applications on the Cray X1 system. [Online]. Available: http://docs.cray.com/books/S-2315-52/html-S-2315-52

[76] ARM Limited. (2014, Feb.) Project Ne10. [Online]. Available: http://projectne10.github.io/Ne10

[77] Intel Corporation. (2014, Feb.) Short Vector Math Library Intrinsics. [Online]. Available: http://software.intel.com/en-us/node/461298

[78] Vector Fabrics B.V., “Pareon: multicore software optimization tool,” Tech. Rep., 2012.

[79] J. Gebis and D. Patterson, “Embracing and Extending 20th-Century Instruction Set Architectures,” *IEEE Comput.*, vol. 40, no. 4, 2007.

[80] T. U. of Michigan. (2011, July) The gem5 Simulator System. [Online]. Available: http://gem5.org

[81] Valgrind$^{\mathrm{TM}}$ Developers. (2011, July) Valgrind. [Online]. Available: http://valgrind.org

[82] J. Henning, “COMPUTING PRACTICES-SPEC CPU2000: Measuring CPU Performance in the New Millennium,” *Computer.*, vol. 33, no. 7, 2000.

[83] MediaBench Consortium. (2010, Dec.) MediaBench II Benchmark. [Online]. Available: http://euler.slu.edu/~fritts/mediabench/

[84] G. Hamerly, E. Perelman, J. Lau, and B. Calder, “Simpoint 3.0: Faster and more flexible program phase analysis,” *JILP*, vol. 7, no. 4, 2005.

[85] N. Muralimanohar and R. Balasubramonian, “CACTI 6.0: A tool to model large caches,” Tech. Rep., 2009.

[86] T. Barr, A. Cox, and S. Rixner, “SpecTLB: a mechanism for speculative address translation,” in *ISCA*, 2011.

[87] A. Randy and K. Kennedy, *Optimizing Compilers for Modern Architectures.* San Francisco, CA: Morgan Kaufman, 2001.

[88] P. M. Kogge and H. S. Stone, “A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations,” *IEEE Trans. Comput.*, vol. C-22, no. 8, Aug. 1973.

[89] S. Knowles, “A family of adders,” in *ARITH*, 1999.

[90] A. Krste *et al.*, “The Landscape of Parallel Computing Research: A View from Berkeley,” EECS Department, University of California, Berkeley, Tech. Rep., Dec. 2006.

[91] National Technical Information Service (NTIS), “Announcing the ADVANCED ENCRYPTION STANDARD (AES),” Tech. Rep., 2001.

[92] D. Rumelhart, G. Hinton, and R. Williams, “Learning representations by back-propagating errors,” *Nature*, vol. 323, 1986.

[93] The Embedded Microprocessor Benchmark Consortium. TeleBench^{TM} Version 1.1 .

[94] E. Dijkstra, “A note on two problems in connexion with graphs,” *Numer. Math.*, vol. 1, 1959.

[95] Center for Discrete Mathematics & Theoretical Computer Science. 9th DIMACS Implementation Challenge - Shortest Paths.

[96] J. T. Schwartz, “Ultracomputers,” *TOPLAS*, vol. 2, no. 4, 1980.

[97] G. E. Blelloch, *Vector Models for Data-Parallel Computing*, 75th ed. The MIT press, 1990, vol. 2, no. 5.

[98] S. Sengupta, “Efficient Primitives and Algorithms for Many-core architectures,” Ph.D. dissertation, University of California, 2010.

[99] S. Chatterjee, G. Blelloch, and M. Zagha, “Scan primitives for vector computers,” in *Supercomputing*, 1990.

[100] C. Lee and J. Smith, “A study of partitioned vector register files,” in *Supercomputing*, 1992.

[101] J. Abella and A. González, “SAMIE-LSQ: set-associative multiple-instruction entry load/store queue,” in *IPDPS*, 2006.