Overcoming the Limitations of Conventional Vector Processors

Christos Kozyrakis
Electrical Engineering Department
Stanford University
christos@ee.stanford.edu

David Patterson
Computer Science Division
University of California at Berkeley
pattrsn@cs.berkeley.edu

Abstract

Despite their superior performance for multimedia applications, vector processors have three limitations that hinder their widespread acceptance. First, the complexity and size of the centralized vector register file limits the number of functional units. Second, precise exceptions for vector instructions are difficult to implement. Third, vector processors require an expensive on-chip memory system that supports high bandwidth at low access latency.

This paper introduces CODE, a scalable vector microarchitecture that addresses these three shortcomings. It is designed around a clustered vector register file and uses a separate network for operand transfers across functional units. With extensive use of decoupling, it can hide the latency of communication across functional units and provides 26% performance improvement over a centralized organization. CODE scales efficiently to 8 functional units without requiring wide instruction issue capabilities. A renaming table makes the clustered register file transparent at the instruction set level. Renaming also enables precise exceptions for vector instructions at a performance loss of less than 5%. Finally, decoupling allows CODE to tolerate large increases in memory latency at sub-linear performance degradation without using on-chip caches. Thus, CODE can use economical, off-chip, memory systems.

1 Introduction

Modern vector and data-parallel architectures offer significant advantages over superscalar processors for a wide range of compute-intensive applications: multimedia (VIRAM [19], Imagine [15]), broadband and wireless communication (Intel IXS [26], Broadcom Calisto [20]), bioinformatics workloads (Cray X1 [1], Tarantula [7]), climate modeling (NEC ES [24]). The abundance of data-level parallelism in such tasks allows them to concurrently execute tens of arithmetic and memory operations, while issuing a single instruction per cycle [6]. For data-parallel tasks, vector processors approach the performance and power efficiency of custom designs, while maintaining the flexibility and programmability of general-purpose processors. They are also simpler to build and easier to scale with CMOS technology than superscalar processors [16, 19].

Nevertheless, there are three remaining obstacles to the widespread adoption of vector architectures in general-purpose computer systems: the complexity of a multiported centralized vector register file (*VRF*), the difficulty of implementing precise exceptions for vector instructions, and the high cost of on-chip vector memory systems.

In this paper, we introduce CODE, a microarchitecture that addresses the three limitations of conventional vector organizations. The cornerstone of the microarchitecture is a clustered vector register file (*CLVRF*) that separates the task of staging the operands for a single functional unit from the task of communicating data among functional units. Since it does not have to support all-to-all communication between the functional units, as it is the case with the centralized VRF, the CLVRF is significantly simpler. To avoid the performance loss typically associated with clustered register files [12], CODE makes extensive use of decoupling. It also uses a simple renaming table that makes its distributed nature transparent at the instruction set level.

To demonstrate the potential of CODE, we use the VIRAM instruction set for multimedia processing and compare it to the VIRAM media-processor, a multi-lane vector design with a centralized VRF. For the EEMBC benchmarks and assuming equal die area, CODE is 26% faster than VIRAM. While VIRAM is limited to 3 functional units and requires an on-chip memory system, CODE scales efficiently to 8 functional units with multiple vector lanes and can effectively hide the high latency of off-chip memory accesses. In addition, CODE can support precise exceptions for vector instructions at negligible performance loss.

The rest of this paper is structured as follows. Section 2 discusses the three limitations of conventional vector processors. Section 3 introduces the basic features of CODE. Section 4 focuses on precise exceptions for vector instructions. Section 5 proceeds with a performance and scaling evaluation of CODE, including a comparison with the VIRAM processor. Section 6 presents related work and Section 7 concludes the paper.

2 Limits of Conventional Vector Processors

Two factors contribute to the *high complexity of the centralized VRF*: it stores a large number of vector elements and, more importantly, supports high bandwidth communication of operands between the vector functional units (*VFUs*). A typical VRF stores 2 to 64 KBytes and supports 3 to 10 VFUs [13]. In general, for a vector processor with N VFUs, the VRF must have approximately $3N$ access ports. The VRF area, power consumption, and access latency are roughly proportional to $O(N^2)$, $O(log_4 N)$, and $O(N)$ respectively [21]. We can bind the VRF complexity to that of a 9-ported memory structure by using $N \leq 3$ functional units. However, this approach limits the ability to overlap the execution of more than 3 vector instructions, which is critical for applications with short vectors. Alternatively, we can time-share a small number of VRF ports between multiple VFUs, which leads to low utilization of the VFUs or increases the latency of chaining. Finally, we can use bank partitioning to implement the multi-ported VRF as a collection of banks with a smaller number of access ports per bank [2]. Bank partitioning leads to complicated cases of structural hazards in the case of non-uniform VFU latencies. In general, the techniques for reducing the VRF complexity provide a one-time reduction of the constant factors associated with its area, power consumption, and access latency and are not applicable to designs with a large number of VFUs.

Precise exceptions are required for virtual memory support and desirable for handling arithmetic faults. However, they are difficult to implement in vector processors. Each vector instruction defines a large number of element operations and takes several cycles to execute, even in multi-lane organizations. To implement in-order instruction commit, a vector processor requires a reorder buffer that can store at least one vector register for each VFU. Due to its large capacity and the need for multiple access ports for forwarding, a vector reorder buffer has similar complexity to the VRF. Alternatively, we can integrate the reorder buffer with the VRF, which exacerbates the complexity of the VRF by increasing its capacity [9]. Consequently, most vector processors support imprecise exceptions or do not support restartable exceptions at all. An additional complication for precise virtual memory exceptions for vector loads and stores is the need for a large TLB. To guarantee forward progress, the TLB must have enough entries to eventually check and translate all the virtual addresses generated by one vector instruction without generating TLB refill errors. In the worst case, an indexed vector load may access a separate memory page for each element, which requires a large number of TLB entries. For example, the Tarantula vector extension for the Alpha architecture uses 512 entries (16 32-entry fully associative TLBs) to avoid TLB thrashing on indexed vector accesses (128 elements/instruction).

Vector processors traditionally require an *aggressive memory system*. High memory bandwidth is inherently necessary to match the high throughput of arithmetic operations in VFUs with multiple datapaths (multi-lane designs). Nowadays, we can provide high memory bandwidth with a number of commodity technologies such as DDR and Rambus [4]. However, recent vector processors also rely on low memory latency to reduce the number of memory related stalls for vector instructions and simplify the overall design. For example, Tarantula uses a 4-MByte on-chip cache with 128 banks to obtain 12-cycle access latency for vector memory accesses. VIRAM uses a 13-MByte on-chip DRAM memory with 8 banks to limit latency to 8 cycles. In general, low access latency requires on-chip memory (main memory or cache), which increases the overall system cost.

3 The CODE Microarchitecture

This section introduces CODE (*Clustered Organization for Decoupled Execution*) within the context of the VIRAM ISA for multimedia processing. However, CODE is equally applicable to any other modern vector ISA, such as the Cray X1 [1] or the Alpha Tarantula [7]. A complete description of CODE is available in [18].

The VIRAM ISA is a vector load-store extension to the MIPS architecture. It defines a 8-KByte vector register file that stores 32 general-purpose registers with 32 64-bit, 64 32-bit, or 128 16-bit elements per register. Integer, floating-point, and load-store instructions operate on vector registers under vector length control. Load-store instructions can describe sequential, strided and indexed access patterns. A detailed description and analysis of the IRAM ISA for multimedia applications is available in [19].

3.1 Clustered Register File

Figure 1 presents the clustered organization of vector hardware in CODE. Each cluster contains a single vector functional unit (arithmetic or load-store) and a small number of vector registers (e.g. 4 or 8). Collectively, the clusters must store at least 32 vector registers, the number defined in the ISA. Yet, there is no upper bound to the overall number of registers. A cluster also includes an instruction queue for decoupling reasons, as well as one input and one output interface which allow vector register transfers from and to other clusters. The two interfaces do not contain any storage, except for a single cycle buffer for retiming purposes.

CODE distributes the vector registers across the clusters. The local CLVRF partition within each cluster provides operands to the local VFU and the two communication interfaces. Hence, it requires 5 access ports: 2 read and 1 write for the VFU, 1 read port for the output interface, and 1 write port for the input interface. The number of ports is independent of the overall number of clusters. In

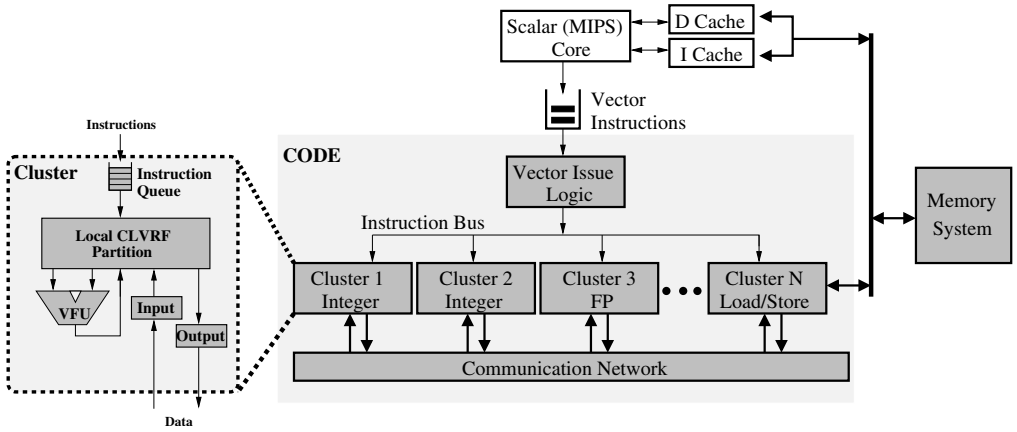

Figure 1. The block diagram of CODE. The total number of clusters (N) is a design parameter. A cluster contains an integer, floating-point, or load-store VFU. We can also have a cluster without a VFU that merely introduces extra vector registers. The overall number of vector registers across all clusters can be larger than 32.

other words, the area, power consumption, access latency, and complexity of each CLVRF component are constant. The overall CLVRF area and power consumption are proportional to N, yet so is the total number of vector registers. Each additional cluster contributes both a VFU and a few vector registers that will stage its operands and temporary results. Therefore, the area and power consumption per vector register in CLVRF are constant. With the centralized VRF, on the other hand, the number of registers is constant, but the area and power consumption grow with $O(N^2)$ and $O(log_4 N)$ respectively.

3.2 Communication Network

CODE requires a separate communication network for transferring vector registers between clusters. If the source operand for a vector instruction is not readily available in the cluster selected for its execution, we must move it across clusters. Communication takes place by executing a tagged *move-to* instruction in the output interface of the sending cluster and a tagged *move-from* instruction in the input interface of the receiving cluster respectively. All move instructions are generated by hardware in the vector issue logic. Since we use the same tag for the two corresponding move instructions, the high-level control of the network is straightforward. The input and output interface of the clusters pass to the network the tag of the move instruction they want to execute. As soon as a tag match is established between one move-to and one move-from instruction, we can initiate the transfer[1]. The full vector register transfer will take several cycles to complete since the network will likely be able to move only a few vector elements per cycle.

[1]If all input and output interfaces execute move instructions in the order issued to their cluster, we can prove that the network is deadlock-free.

For a fair comparison between the CLVRF and the centralized organization, we must include the area, power, and complexity of the communication network. The centralized VRF incorporates a single-cycle crossbar between the input and output operands of all functional units. If the CODE network has to provide similar functionality, the CLVRF benefits will likely be canceled out. Nonetheless, several reasons suggest that the network does not have to be a full crossbar. First, certain pairs of VFUs are rarely used in the same program or hardly ever communicate (e.g. integer and floating-point units). In addition, vector instructions frequently use scalar values as one of their source operands. Finally, the result of one instruction is often used by just one or two subsequent instructions before it is discarded [3]. Hence, the overall number of times we will need to move each vector result is inherently small. We can also try to assign each vector instruction to the cluster that already has most of its operands in the local CLVRF partition. In Section 5.1, we show that for multimedia tasks the communication bandwidth necessary to achieve full performance is only a couple of 64-bit words per cycle.

By separating the exchange of vector operands between functional units from the VRF, CODE turns the exact structure of the communication network into a separate design decision. A low-cost processor implementation may select a simple network (e.g. bus or ring) that has low area and power overhead. On the other hand, a high-end implementation may favor a low latency, high bandwidth network that can minimize the number of cycles each VFU has to wait for its source operands to arrive from another cluster. In general, we can trade off area, power, and complexity in the network for performance without affecting the design of the clusters.

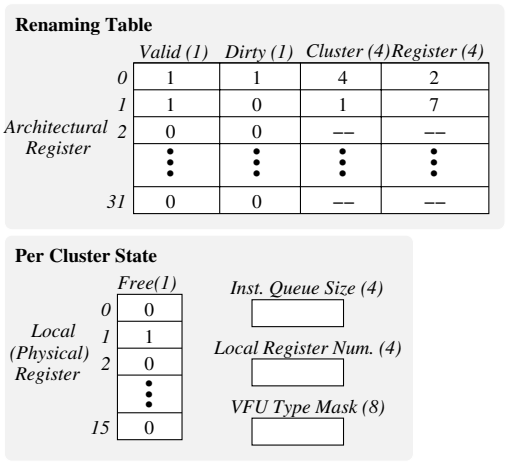

Figure 2. The renaming table and per-cluster state in the vector issue logic. The specified bit widths can support a system with 16 clusters and 16 vector registers per cluster. The valid and dirty bits for each architectural register are also defined in the VIRAM ISA. In this example, the table indicates that the contents of architectural vector register 0 are available in the physical vector register 2 of cluster 4.

3.3 Vector Issue Logic

The distributed register file in CODE deviates significantly from the register file model defined in the VIRAM ISA. The processor may include more than 32 vector registers across all clusters. In addition, each VFU has direct access only to a subset of the registers. We address these two issues in the vector issue logic block.

The basic role of the issue logic is to select the proper cluster to execute each instruction. This task becomes trivial when there is only one cluster with the proper VFU. The issue logic may also need to generate the proper inter-cluster transfers that will move the source operands of the instruction to the selected cluster. Hence, it must track the physical location of the 32 architectural vector registers, i.e. the cluster and vector register within the cluster that currently stores the values of its elements. Figure 2 presents the renaming table that maintains the mapping of architectural to physical vector registers. Renaming also eliminates WAW and WAR hazards between vector instructions. Figure 2 also presents the per-cluster state that the issue logic maintains, namely a free-list for the physical registers in the cluster and a few control registers.

The issue logic reads the renaming table once for each instruction in order to find the location of its operands. After it selects a cluster, it updates the table to reflect the new physical location of architectural registers due to the execution of the instruction. All the instruction operands will be local to the selected cluster, which requires up to 3 vector register transfers[2]. Since each cluster can store a limited number of vector registers, there may not be sufficient space for the instruction operands. In this case, we first need to transfer some of the locally stored architectural registers to other clusters with unallocated physical registers. Overall, the execution of an instruction may require 0 to 6 register transfers. The issue logic generates the move-to and move-from instructions for the transfers and issues them to the proper clusters along with the original instruction.

Obviously, minimizing the number of transfers per instruction is a critical issue. A large number of transfers will likely lead to frequent stalls, as VFUs will have to wait for their data to arrive from other clusters. To avoid performance degradation, we will need high bandwidth on the inter-cluster network, which is expensive in terms of area and power. We can reduce the number of transfers by always selecting the candidate cluster that requires the least transfers. Still, this approach is only locally optimal and can lead to load imbalance. We will analyze the impact of cluster selection policy in Section 5.1.

The issue logic in CODE is similar in functionality to that of multi-cluster superscalar processors [11]. However, it is much simpler, as it is sufficient to maintain an issue rate of one vector instruction per cycle. Each vector instruction defines tens of element operations and will occupy a cluster for several clock cycles. In Section 5.5, we demonstrate that a single-issue CODE implementation can scale to 8 clusters. On the other hand, a superscalar processor must issue at least one instruction per cluster per cycle to make efficient use of its hardware. The wide instruction issue with the potential for dependencies between instructions leads to exponential growth in issue logic complexity.

It is important to note that the issue logic dispatches load-store, arithmetic, and move instructions to the clusters, but does not control their execution. Each cluster contains the proper decoding logic to execute instructions, and the network is responsible for sequencing vector element transfers between input and output interfaces.

3.4 Discussion

CODE executes sequences of vector instructions in a decoupled manner. Execution within each cluster proceeds in-order, with every instruction typically taking several cycles to complete. However, we can exploit the instruction queues to decouple execution across clusters. Clusters need to synchronize only when queues become full or when data dependencies arise and they need to execute the move instructions that implement register transfers. We can hide a significant portion of the overhead of inter-cluster transfers by allowing the input and output interfaces to look ahead in

[2]Up to 2 transfers for source operands. The third transfer for the destination is only needed for predicated execution of vector instructions.

the instruction queue and start executing move instructions prior to older arithmetic instructions, if there are no dependencies. Overall, decoupled execution is the key feature that allows CODE to avoid the typical performance degradation associated with clustered register files (see Section 5.2).

CODE also simplifies the implementation of chaining, the vector equivalent of forwarding, which is a critical performance factor for any vector processor. CODE implements chaining across a large number of clusters in a scalable manner that uses only intra-cluster control logic. For example, assume that `vadd` and `vsub` execute in cluster 1 and 2 respectively and that `vsub` uses the outcome of `vadd` as a source operand. In cluster 1, the VFU that executes the `vadd` will chain to the output interface. As soon as `vadd` produces one element of its result, the output interface can push it toward cluster 2 through the network. In cluster 2, the VFU that executes the `vsub` chains to the input interface: as soon as another element arrives, `vsub` can execute the corresponding element operation. In a multicluster system, the VFU in one cluster can chain to the VFU in any other cluster just by checking the progress of move instructions in its local input and output interfaces. In a conventional vector processor with a centralized VRF, on the other hand, chaining requires control logic that checks the progress of execution in all possible pairs of VFUs.

Finally, the clustered organization of CODE is compatible with multi-lane implementations. Multi-lane vector processors use parallel datapaths and address generators within each VFU to execute a large number of element operations per cycle for each instruction [13]. A lane is a partition of the vector processor that contains one datapath from each VFU and a set of elements from each vector register. Lanes and clusters are basically two orthogonal ways to add extra hardware resources (datapaths) in a vector design. Lanes allow the parallel execution of *multiple element operations per instruction* and clusters allow the concurrent execution of *multiple vector instructions*. Hence, we can apply CODE to a multi-lane vector processor by organizing the hardware resources within each lane into multiple clusters. We evaluate the two scaling approaches, clusters and lanes, in Section 5.5.

4 Support for Precise Exceptions

At first sight, the decoupled nature of CODE seems to create further obstacles to implementing precise exceptions. However, we can exploit the renaming capabilities in the issue logic to achieve in-order commit of vector instructions without significant performance loss. We can also adjust the definition of precise exceptions for vector instructions to eliminate the need for a large TLB.

4.1 Microarchitecture Support

The key to in-order commit is not to allow a vector instruction to change the value or the physical location of an architectural register until it is known that this and all preceding instructions will complete without faults. On issuing an instruction, we assign unallocated registers in the selected cluster for its destination or any source registers that must be moved across clusters. We do not release the physical registers with the old value of the destination or the old locations of the sources until the exception behavior is known. However, after we have preserved the old value of a vector register once, we do not preserve additional intermediate values until we issue another instruction that may cause a fault (e.g. vector load). This optimization can reduce significantly the pressure on vector registers, if a sufficient number of the vector instructions executed can never to generate faults (e.g. vector shift).

We can implement in-order commit without modifying the clusters by adding a history buffer in the issue logic [23]. The history buffer keeps track of the changes to the renaming table and the per-cluster state for each instruction in case we need to undo them. We insert and extract instructions from the history buffer in issue order. If the instruction at the buffer head completes without exception, we release any physical vector registers that maintain old values or locations for its operands. If the instruction has caused an exception, we must scan the history buffer from tail to head and restore the old vector register mappings for all pending vector instructions before we flush the buffer. It is important to note that the history buffer does not store the old vector register values, just the old mappings. We only need to enqueue and dequeue one vector instruction per cycle, which makes the buffer simpler than a history buffer for a superscalar processor.

Supporting exceptions places increased pressure on vector registers. Most vector registers in a cluster may be unavailable because they store old register values for pending instructions. Hence, we may not be able to issue a new instruction that needs to transfer its operands to this cluster. In other words, supporting precise exceptions may cause frequent stalls in the issue logic and lead to performance loss. We measure the performance impact of precise exceptions in Section 5.3.

4.2 ISA Support

The history buffer allows CODE to implement precise exceptions for vector instructions, but it does not simplify the TLB for vector loads and stores. For this purpose, we need an ISA level change.

The fundamental cause of the TLB requirement is that the definition of precise exception requires that the faulting instruction cannot modify architecture state. As a result,

an indexed vector load must be able to translate all its element addresses without exceptions in order to make forward progress. We can relax the TLB requirement by allowing the faulting instruction to update architecture state up to the first element that causes an exception. For example, if an indexed vector load causes translation errors for elements 10 and 15, we allow the loading of the first 9 elements to commit. We still expect that all previous instructions have fully completed and all following instructions have not updated architecture state at all. When we resume from the exception, we restart the faulting instruction from the 10th element. All following instructions proceed as usual.

With the proposed modification, a large TLB is no longer a functional requirement, just a performance optimization. Even with a single-entry TLB, an indexed load or store is guaranteed to advance by at least one element operation after each TLB miss exception. Processors for applications with mostly sequential accesses or small working sets may use a small number of TLB entries. Processors optimized for indexed access patterns on large working sets may still choose a large TLB.

Allowing partial completion of one vector instruction is very similar to the ability of VLIW processors to resume from exceptions in the middle of a long instruction word. The idea is to recognize that each vector or VLIW instruction defines multiple operations with independent exception behavior. Apart from the control register for the faulting PC, the implementation of this technique requires a control register that indicates the first element that caused an exception. We can use the same register when we resume to indicate that the first vector instruction must not start from element 0.

5 Evaluation

To evaluate CODE, we developed a trace-driven, parameterized performance model that accounts for detailed timing events. The model includes a simple, in-order scalar core with 8-KByte first-level caches. The default configuration has similar execution capabilities and occupies approximately the same area as a single-lane version of the VIRAM processor [19]. It includes 2 integer and 1 load-store cluster, each with 8 vector registers and a 8-entry instruction queue. Arithmetic VFUs can execute 1 64-bit, 2 32-bit, or 4 16-bit operations per cycle. The load-store VFU can generate one address and exchange up to 64 bits with the memory system per cycle. An additional (4th) cluster introduces 16 vector registers, but includes no VFU[3]. The network bandwidth is two 64-bit words per cycle and its latency is 2 cycles. The memory system uses 8 banks of DRAM main memory and has 8 cycles access latency. Unless otherwise stated, the following sections assume the parameters of the default configuration.

[3]The default configuration includes 40 vector registers, 8 more than VIRAM. However, the reduced number of access ports of the CLVRF components leads to approximately the same die area as VIRAM.

Consumer Benchmarks	
Rgb2cmyk	Converts an RGB image to the CMYK format
Rgb2yiq	Converts an RGB image to the YIQ format
Filter	High-pass gray-scale image filter
Cjpeg	JPEG image compression
Djpeg	JPEG image decompression
Telecommunications Benchmarks	
Autocor	Voice compression using autocorrelation
Convenc	Convolutional encoder for modems
Bital	Bit allocation to frequency bins for ADSL
Fft	256-point fixed-point fast Fourier transform
Viterbi	Viterbi decoding for wireless applications

Table 1. The consumer and telecommunication benchmarks in the EEMBC suite [5]. The benchmarks include no floating-point operations.

Our evaluation uses the ten benchmarks from the EEMBC suite presented in Table 1. They are representative of the workload of multimedia devices with wireless or broadband communication capabilities. The use of EEMBC enables direct comparisons to the VIRAM processor. The benchmarks are highly vectorizable, which allows us to concentrate on the vector hardware design, which is the main focus of this paper. Unlike scientific applications, the EEMBC benchmarks include both long and short vector operations, which is typical of multimedia applications. An analysis of the vectorization of the EEMBC benchmarks is presented in [19]. The traces used with the CODE performance model were generated from the C benchmark code using the VIRAM vectorizing compiler and ISA simulator.

5.1 Microarchitecture Tuning

Before we proceed with the evaluation of CODE, we tune some of the basic microarchitecture parameters.

Figure 3 present the impact of cluster selection policy in a CODE configuration with 2 clusters for each type of vector instructions (integer, load-store). We report speedup over a configuration with a single cluster per type (no selection necessary). By selecting the cluster that requires the minimum number of register transfers, we reduce the pressure on the communication network but create load imbalance. On average, this policy performs slightly worse than random selection, which is trivial to implement. We can eliminate load imbalance by taking into account the size of cluster instruction queues. The load balancing policy in Figure 3 selects the cluster for minimum communication, unless the difference in the queue sizes for the two candidates is more than half the queue capacity (4 instructions).

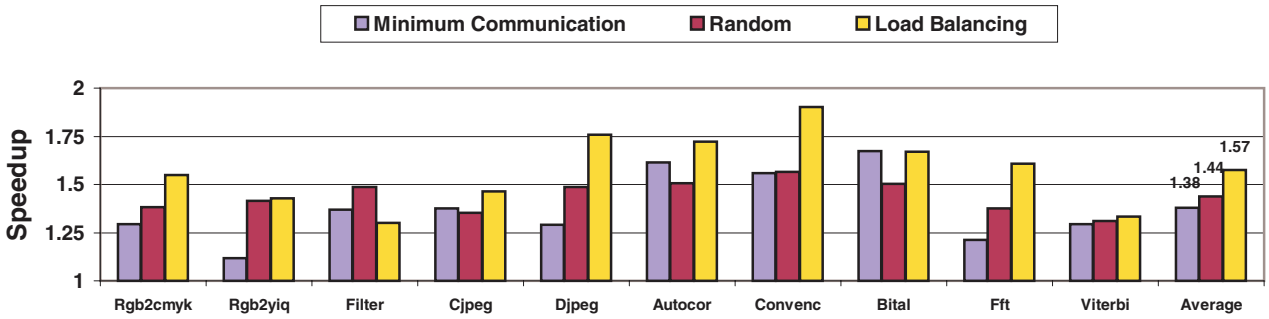

Figure 3. The effect of cluster selection policy on performance.

This policy leads to 14% performance improvement over the minimum communication policy.

The number of vector registers per cluster is another critical parameter. Without sufficient local registers, each vector instruction will require a large number of inter-cluster transfers, which will likely lead to performance loss (see discussion in Section 3.3). Figure 4 presents the number of vector register transfers per instruction for the default configuration as a function of the number of vector registers per cluster. In all cases, the average number of transfers is much lower than the worst-case 6, due to the reasons discussed in Section 3.2. Eight registers per cluster are sufficient to hold the operands and temporary results of each VFU and lead to less than one transfer per two vector instructions for all benchmarks excluding `Filter`. The main kernel of `Filter` uses all 32 architectural registers for temporary results, and its register access pattern interacts poorly with the load balancing cluster selection policy.

Figure 4 suggests that the actual requirements for inter-cluster communication are low. Our experiments indicate that the ability to execute in parallel 2 register transfers at the rate of one 64-bit word per transfer per cycle is sufficient to reach the full performance potential with the default configuration. This is significantly lower bandwidth than that supported by the full crossbar integrated in a centralized VRF. The communication latency has no significant effect on the performance of CODE, since decoupling provides sufficient tolerance for reasonable latency values.

5.2 Comparison to VIRAM

Figure 5 presents the performance improvement of CODE over VIRAM. This baseline comparison assumes that both microarchitectures use the same scalar core and memory system, have identical computational capabilities (number of VFUs and datapaths per VFU), and occupy approximately the same die area. We also assume that the two operate at the same clock frequency, even though the smaller number of access ports per CLVRF component in CODE allows for higher frequency than VIRAM.

VIRAM uses a centralized VRF, hence there is no performance penalty for communicating vector results between its VFUs. On the other hand, CODE makes extensive use of decoupling, which can hide the latency of inter-cluster communication. Decoupling also supports a limited form of out-of-order execution across clusters, which hides other sources of latency, such as stalls due to bank conflicts in the memory system or ineffective static scheduling. The VIRAM pipeline is strictly in-order with no decoupling across VFUs. A stall in any VIRAM VFU causes all other functional units to stall as well, even in the absence of dependencies. The main motivation for this rigid pipeline in VIRAM is to simplify chaining decisions.

Figure 5 shows that VIRAM slightly outperforms CODE for benchmarks with frequent inter-cluster transfers (`Filter`) or no opportunities for decoupling across clusters (`Bital`). However, CODE outperforms VIRAM for most other benchmarks, especially for those with many strided accesses that cause frequent bank conflicts (`Rgb2cmyk`). On the average, CODE outperforms VIRAM by 21% to 42%, depending on the number of lanes used with each microarchitecture. The performance advantages of CODE are consistent, yet somewhat smaller with 4 or 8 lanes. A large number of lanes limits to an extent the ability of CODE to hide the latency of communication events (see further discussion in Section 5.5).

For comparison, we should note that, for the EEMBC benchmarks, the 200MHz 4-lane VIRAM processor is 60% to 200% faster than GHz-level OOO processors and wide VLIW designs with specialized SIMD instructions [19].

5.3 Cost of Precise Exceptions

The results so far do not consider the potential cost of precise exception support. Figure 6 presents the performance impact of turning precise exceptions on for vector instructions in the default CODE configuration. The performance degradation is due to the additional pressure on the vector registers within each cluster and the resulting stalls in the issue logic. Figure 6 measures the impact on regu-

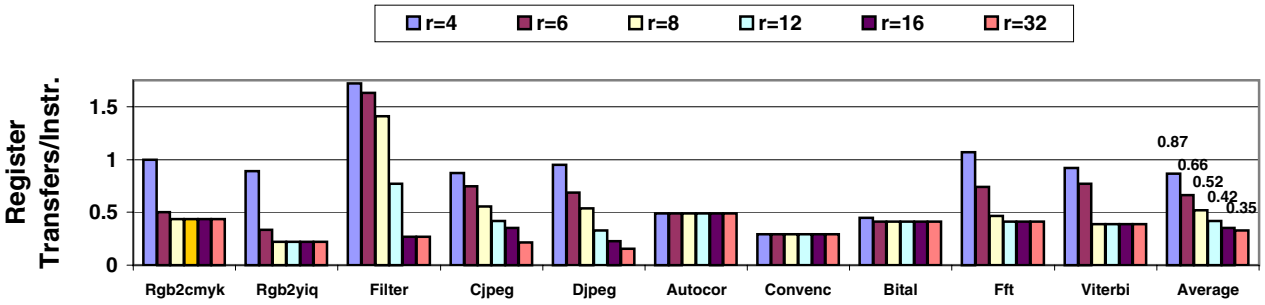

Figure 4. The number of inter-cluster vector register transfers per vector instruction as a function of the number of local vector registers per cluster (r).

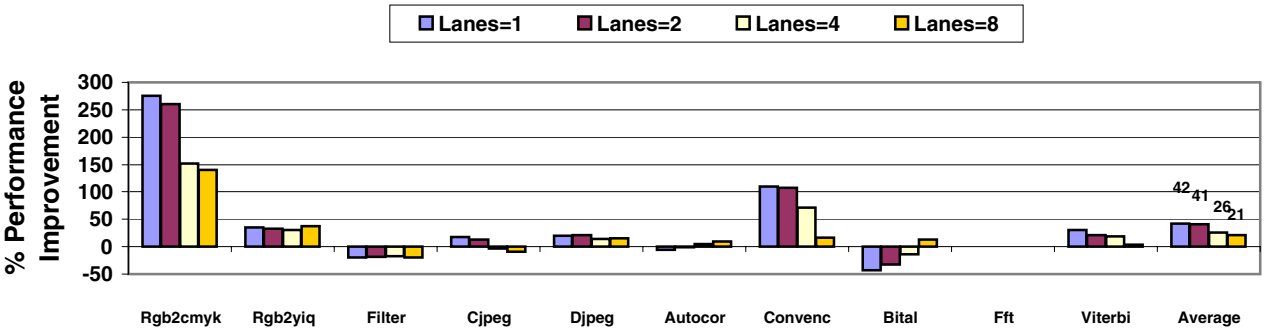

Figure 5. The percentage (%) of performance improvement of CODE over VIRAM. Negative percentage indicates that VIRAM is actually faster than CODE.

lar program execution without any exception actually taking place. It does not attempt to quantify the overhead of operating system code in the event of a vector exception. The VIRAM ISA provides mechanisms that minimize the number of exceptions that lead to saving/restoring vector registers and reduce the amount of vector state that is involved with each context switch. The evaluation of these mechanisms is beyond the scope of this paper.

Figure 6 shows that, with 8 registers per cluster, the performance impact of precise exceptions is less than 5% on the average[4]. Eight local registers are sufficient to store the operands, temporary results, and any preserved old values for the VFU in the cluster. With less than 8 registers, the pressure on local registers becomes significant, because the VFU needs 2 to 3 vector registers just for the operands of the currently executing instruction. In this register limited case, the performance loss can be significant (average 15%, worst case 40%).

[4]In a few cases, supporting precise exceptions leads to speedup (negative slowdown in Figure 6). Stalls in the vector issue logic due to local register pressure sometimes lead to better cluster selection.

5.4 Memory Latency Tolerance

The original motivation for decoupling techniques was to hide the latency of memory accesses [22]. Vector processors are also capable of tolerating memory latency on their own, as they can amortize the cost of initiating a memory transfer over a large number of element operations [6].

Figure 7 quantifies the potential of combining the two latency tolerance techniques in CODE. It presents the increase in execution time as we increase the latency per element access from 1 to 128 processor cycles. In this case, we assume that there is no limit to the number of outstanding memory requests, either at the processor or memory side (i.e. infinite bandwidth). Even though practical implementations will have to set some limit, it is interesting to determine the maximum potential. Figure 7 shows that CODE can tolerate latencies up to 32 clock cycles with less than 25% performance loss over the single-cycle access case (perfect memory). With a latency of 128 cycles, execution time increases by 1.8 times on the average (worst case 3x). In contrast, the performance loss in the VIRAM processor with increasing memory latency would be dramatic, because its rigid pipeline is designed specifically for the 8-cycle access la-

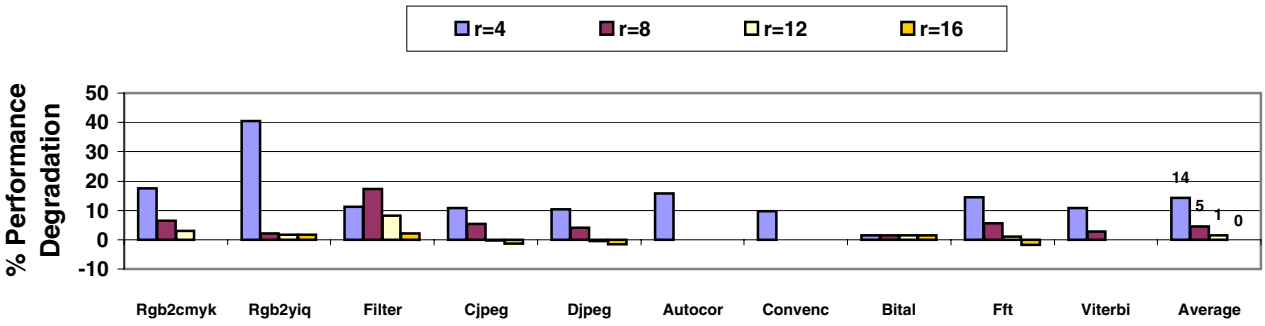

Figure 6. The performance loss due to supporting precise exceptions for vector instructions as a function of the number of local vector registers per cluster (r) in the default configuration. In all cases, we compare to the performance without precise exceptions for the same number of registers per cluster.

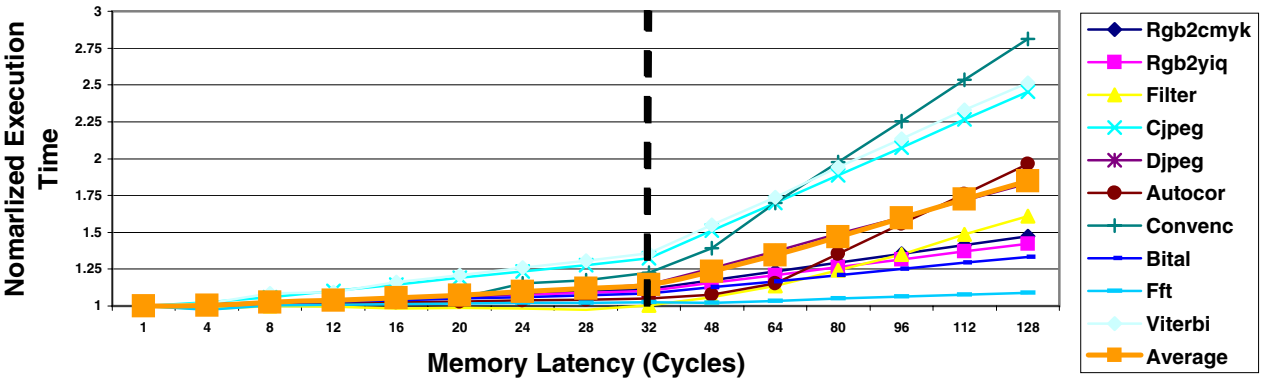

Figure 7. The effect of memory access latency on the execution time on CODE. Execution time is normalized to the case of single cycle main memory latency. Note that the x axis consists of two linear regions (1 to 32 and 32 to 128).

tency and cannot tolerate a slower memory system.

Figure 7 shows that, as long as sufficient bandwidth is available, it is reasonable to use an off-chip memory system with a vector processor. Even though both on-chip and off-chip bandwidth come at a cost, off-chip memory systems can be larger and more flexible. A vector processor operating at a modest clock frequency for power efficiency does not require on-chip caches or main memory. It can access directly an off-chip main memory system at a small performance loss.

5.5 Scalability

Figure 8 presents the performance of CODE as we scale the number of clusters and lanes in the default configuration. To increase the number of clusters, we first introduce extra integer clusters and then add one load-store cluster for every two integer clusters when possible. With the additional clusters, we need to increase the bandwidth available in the inter-cluster communication network. We allow for 2 additional 64-bit transfers per cycle for every 3 extra clusters in the system. In the case of 2, 4, or 8 lanes, we increase the width of the inter-cluster transfers from 64 bits to 128, 256, or 512 bits respectively. In all cases, we scale properly the number and width of the ports to the 8-bank main memory system.

Increasing the number of lanes allows the execution of multiple (data-parallel) element operations per cycle for each vector instruction in progress. Two lanes lead to a nearly perfect speedup of 2. Four and eight lanes lead to speedups of 3.1 and 4.8 respectively. In general, efficiency drops significantly as we move to a large number of lanes. Applications with very short vectors (10 16-bit elements) cannot use efficiently a large number of 64-bit datapaths. For programs with longer vectors, multiple lanes decrease the execution time of each vector instruction, so it becomes more difficult to hide the overhead of scalar instructions, inter-cluster communication, and memory latency.

Increasing the number of clusters allows the parallel ex-

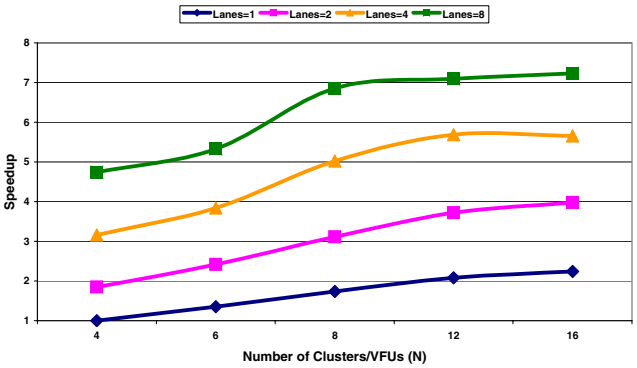

Figure 8. The performance of CODE as we scale the number of clusters and lanes. The speedup presented is the average across the ten benchmarks and is calculated over the default configuration with four clusters (2 integer, 1 load-store, 1 without VFU) and one lane.

ecution of multiple vector instructions. The use of 8 clusters provides 50% to 75% performance improvement over the 4-cluster case, depending on the number of lanes. Yet, there is little performance improvement with more than 8 clusters for two reasons. Primarily, it is not possible to efficiently use more than 8 clusters with single-instruction issue, especially in the case of multiple lanes. Furthermore, data dependencies and an increased number of conflicts in the memory system limit the possibilities for concurrent instruction execution.

Overall, the combination of the two scaling techniques allows CODE to improve its performance up by almost 7 times, using 8 clusters and 8 lanes, before it becomes instruction bandwidth limited.

6 Related Work

Both VIRAM [19] and Tarantula [7], the two most recently proposed vector designs, are based on a centralized vector register file and multiple lanes. They are both limited to 3 VFUs due to the VRF complexity. Asanovic̀ [2] analyzed the VLSI benefits of a bank-partitioned VRF and argued that there is no performance loss compared to a conventional centralized implementation. However, the scope of his work was also limited to a vector processor with a small number of VFUs.

Non-centralized register files have been thoroughly studied outside the scope of vector processors [21]. Imagine [15] uses a hierarchical register file to feed 6 functional units in each of its 8 lanes. Software has full control over execution and communication scheduling through microcoded, VLIW-style instructions. Hence, Imagine can tailor its register access pattern to the data-level or instruction-level parallelism of each application. The price for the flexibility is the need for a full crossbar for communicating data across functional units and lanes, and a complicated programming model. Several VLIW processors [10, 25] use a clustered register file to improve scalability. Their compilers must generate explicit move instructions. There is typically a performance loss associated with the use of a clustered register file in VLIW processors [12]. With CODE, the clustered VRF is transparent to software and decoupling eliminates the performance overhead.

The Alpha 21264 [14], the Multicluster processor [11], and the ILDP architecture [17] use clustering techniques to simplify the register file of an OOO wide-issue engine. The first two assign architectural registers to clusters in a static manner. ILDP and CODE use renaming to allow flexible assignment of registers to clusters. All clustered superscalar approaches require issue bandwidth of one instruction per datapath per cycle for efficient hardware use. By using vector instructions, CODE can scale to 64 datapaths (8 clusters and 8 lanes) with single instruction issue and efficiently hide the latency of inter-cluster communication.

Decoupled vector processors have been analyzed in terms of performance [8] and VLSI cost [2]. Previous proposals limited the use of decoupling to hiding memory latency and required separate data queues for its implementation. CODE relies on decoupling to hide additional high latency events, such as inter-cluster communication and suboptimal static scheduling. It also eliminates the need for separate queues by using the local vector registers in each cluster for data decoupling.

7 Conclusions and Future Work

Three basic limitations discourage the wide use of vector architectures despite their performance potential for a number emerging applications: the complexity of the vector register file, the difficulty of supporting precise exceptions, and the high cost of on-chip vector memory system. This paper introduces CODE, a scalable vector microarchitecture that addresses the three shortcomings. CODE uses a clustered vector register file that reduces complexity by separating the task of storing operands for each functional unit from the task of communicating data between functional units. It makes extensive use of decoupling to hide the latency of inter-cluster communication and the cost of accesses to off-chip memory. It applies renaming to hide its distributed nature from the instruction set and to implement precise exceptions for vector instructions.

For the EEMBC benchmarks, CODE is 26% faster than a vector processor with a centralized vector register file that occupies approximately the same die area. CODE places modest bandwidth requirements on inter-cluster communication and suffers less than 5% performance loss due to pre-

cise exception support. Most importantly, CODE scales efficiently to 64 datapaths (integer and load-store) by combining the ability to execute multiple data-parallel operations on eight vector lanes with the ability to execute multiple vector instructions concurrently on eight vector clusters.

Apart from tuning and evaluating CODE for additional applications, there are several interesting directions for future work. We want to study instruction set enhancements that will assist the vector issue logic with the tasks of cluster selection and register allocation. We also intend to evaluate the potential of CODE with more than one functional unit per cluster and with wider instruction issue capabilities (e.g. 2-way static issue). Overall, we want to investigate the optimal mix of hardware and software techniques for efficiently exploiting the data parallelism available in many compute-intensive applications.

Acknowledgments

Brian Gaeke developed the front-end trace parser for the CODE performance model. We would like to acknowledge all the members of the IRAM research group at U.C. Berkeley and in particular Kathy Yelick, Sam Williams, Joe Gebis, and David Martin. We also thank Suzanne Rivoire, Steve Scott, and the anonymous referees for their insightful comments on earlier versions of this paper. This work was supported by DARPA (DABT63-96-C-0056) and an IBM Ph.D. fellowship.

References

[1] D. Alpert. Scalable MicroSuperComputers. *Microprocessor Report*, pages 1–6, March 2003.

[2] K. Asanović. *Vector Microprocessors*. PhD thesis, Computer Science Division, University of California at Berkeley, 1998.

[3] J. Butts and G. Sohi. Characterizing and Predicting Value Degree of Use. In *the Proceedings of the 35th Intl. Symp. on Microarchitecture*, pages 15–26, Istanbul, Turkey, Nov. 2002.

[4] R. Crisp. Direct Rambus Technology: The Main Memory Standard. *IEEE Micro*, 17(6):18–28, December 1997.

[5] EEMBC Benchmarks. http://www.eembc.org.

[6] R. Espasa et al. Vector Architectures: Past, Present and Future. In *the Proceedings of the 2th Intl. Conf. on Supercomputing*, pages 425–432, Melbourne, Australia, July 1998.

[7] R. Espasa et al. Tarantula: a Vector Extension to the Alpha Architecture. In *the Proceedings of the 29th Intl. Symp. on Computer Architecture*, pages 281–291, Anchorage, AL, May 2002.

[8] R. Espasa and M. Valero. Decoupled Vector Architecture. In *the Proceedings of the 2nd Intl. Symp. on High-Performance Computer Architecture*, pages 281–90, San Jose, CA, Feb. 1996.

[9] R. Espasa, M. Valero, and J. Smith. Out-of-order Vector Architectures. In *the Proceedings of the 30th Intl. Symp. on Microarchitecture*, pages 160–70, Research Triangle Park, NC, Dec. 1997.

[10] P. Faraboschi et al. Lx: a Technology Platform for Customizable VLIW Embedded Processing. In *the Proceedings of the 27th Intl. Symp. on Computer Architecture*, pages 203–13, Vancouver, BC, Canada, June 2000.

[11] K. Farkas et al. The Multicluster Architecture: Reducing Processor Cycle Time Through Partitioning. In *the Proceedings of the 30th Intl. Symp. on Microarchitecture*, pages 327–56, Research Triangle Park, NC, Dec. 1997.

[12] J. Fisher et al. Clustered Instruction-Level Parallel Processors. Technical Report HPL-98-204, HP Labs, Dec. 1998.

[13] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Francisco, CA, third edition, 2002.

[14] R. Kessler. The Alpha 21264 Microprocessor. *IEEE Micro*, 19(2):24–36, March 1999.

[15] B. Khailany et al. Imagine: Media Processing with Streams. *IEEE Micro*, 21(2):35–46, March 2001.

[16] B. Khailany et al. Exploring the VLSI Scalability of Stream Processors. In *the Proceedings of the 9th Symp. on High Performance Computer Architecture*, pages 153–164, Anaheim, CA, Feb. 2003.

[17] H. Kim et al. An Instruction Set and Microarchitecture for Instruction Level Distributed Processing. In *the Proceedings of the 29th Intl. Symp. on Computer Architecture*, pages 71–81, Anchorage, AL, May 2002.

[18] C. Kozyrakis. *Scalable Vector Media-processors for Embedded Systems*. PhD thesis, Computer Science Division, University of California at Berkeley, 2002.

[19] C. Kozyrakis and D. Patterson. Vector Vs Superscalar and VLIW Architectures for Embedded Multimedia Benchmarks. In *the Proceedings of the 35th Intl. Symp. on Microarchitecture*, pages 283–293, Istanbul, Turkey, Nov. 2002.

[20] J. Nickolls et al. Broadcom Calisto: A Multi-Channel Multi-Service Communication Platform. In *the Conf. Record of Hot Chips XIV*, Palo Alto, CA, Aug. 2002.

[21] S. Rixner et al. Register Organization for Media Processing. In *the Proceedings of the 6th Intl. Symp. on High-Performance Computer Architecture*, pages 375–86, Touluse, France, Jan. 2000.

[22] J. Smith. Decoupled Access/Execute Computer Architecture. *ACM Transactions on Computer Systems*, 2(4):289–308, Nov. 1984.

[23] J. Smith and A. Pleszkun. Implementing Precise Interrupts in Pipelined Processors. *IEEE Transactions on Computers*, 37(5):562–73, May 1988.

[24] H. Tahakara and D. Parks. NEC SX Series and Its Applications to Weather and Climate Modeling. In *the Proceedings of the 4th Intl. Workshop on Next Generation Climate Models*, Boulder, CO, March 2002.

[25] M. Tremblay et al. The MAJC Architecture: a Synthesis of Parallelism and Scalability. *IEEE Micro*, pages 12–25, Nov. 2000.

[26] E. Tsui et al. A New Distributed DSP Architecture Based on the Intel IXS. In *the Conf. Record of Hot Chips XIV*, Palo Alto, CA, Aug. 2002.