Vector Lane Threading

Suzanne Rivoire, Rebecca Schultz, Tomofumi Okuda†, Christos Kozyrakis
Electrical Engineering Department
Stanford University
{rivoire,rschultz,tokuda,kozyraki}@stanford.edu

Abstract

Multi-lane vector processors achieve excellent computational throughput for programs with high data-level parallelism (DLP). However, application phases without significant DLP are unable to fully utilize the datapaths in the vector lanes. In this paper, we propose vector lane threading (VLT), an architectural enhancement that allows idle vector lanes to run short-vector or scalar threads. VLT-enhanced vector hardware can exploit both data-level and thread-level parallelism to achieve higher performance. We investigate implementation alternatives for VLT, focusing mostly on the instruction issue bandwidth requirements. We demonstrate that VLT's area overhead is small. For applications with short vectors, VLT leads to additional speedup of 1.4 to 2.3 over the base vector design. For scalar threads, VLT outperforms a 2-way CMP design by a factor of two. Overall, VLT allows vector processors to reach high computational throughput for a wider range of parallel programs and become a competitive alternative to CMP systems.

1 Introduction

Vector and data-parallel processors are making a comeback in several domains, including scientific computing and bioinformatics [9, 16, 30, 13, 29, 25], multimedia processing [24, 19, 17, 5, 22, 15], and telecommunications [1, 31, 4]. Abundant data-level parallelism (DLP) in these domains allows vector processors to provide higher peak and sustained performance than superscalar and chip multiprocessor (CMP) designs while running a single thread, issuing fewer instructions per cycle, and consuming less power [13, 19]. Furthermore, vector processors support a simple loop-based programming model, backed by mature compiler technology for automatic vectorization.

Vector processors use replicated *lanes* in the vector unit to exploit data-level parallelism [2]. Each vector lane contains a slice of the vector register file, a datapath from each vector functional unit, and one or more ports into the memory hierarchy. On each cycle, lanes receive identical control signals and execute multiple element operations for each vector instruction. Multiple lanes allow a vector unit to reach high computational throughput without using the complicated logic necessary for high instruction issue rates. Current vector processors use up to 16 lanes, and future designs are expected to use even more.

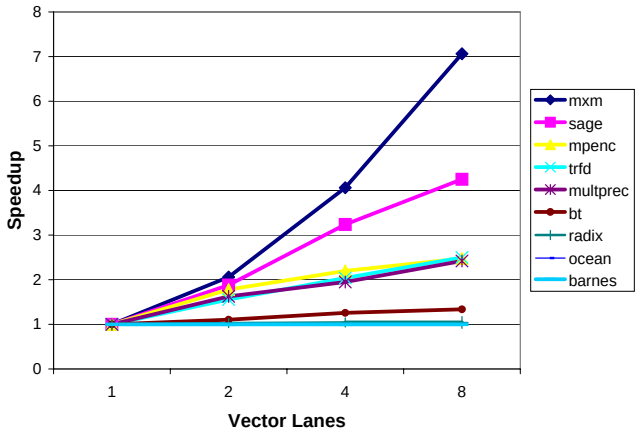

Figure 1. The effect of increasing the number of vector lanes on application performance.

Despite their efficiency for applications with long vectors (high DLP), multi-lane vector processors are not considered as general as CMP designs. This is because applications with medium-length vectors, short vectors, or no vectors at all cannot fully utilize the replicated lanes. Figure 1 shows the relative performance of a high-performance vector processor as we scale the number of lanes from 1 to 8. Applications with long vectors throughout their execution (`mxm`, `sage`) scale well with the number of lanes, which clearly demonstrates the scalability advantage of vector processors. Applications with short vectors (`mpenc`, `trfd`, `multprec`, `bt`) or no vectors at all (`radix`, `ocean`, `barnes`) in significant portions of their execution do not benefit from more lanes, since they underutilize vector hardware for long periods of time. For vector processors to be viable for a wider range of application domains, it is important to improve the performance of such applications on a large number of lanes.

In this paper, we propose multithreading the vector unit to increase the utilization of vector lanes when running low-DLP code. We partition the vector lanes across several threads, which execute in parallel. The number of lanes assigned to each thread corresponds to its amount of data-level parallelism. Even though each thread cannot utilize all lanes on its own, the combination of threads can saturate

†Currently with Sony Corporation (tomofumi@sldc.sony.co.jp). Research performed while a visiting scholar at Stanford University.

the available computational resources. We call this technique *vector lane threading (VLT)*. VLT allows idle DLP resources to be used to exploit the thread-level parallelism (TLP) available in such applications, analogous to the way a processor with simultaneous multithreading (SMT) allows idle resources for instruction-level parallelism (ILP) to exploit TLP. [10, 11]. A VLT-enhanced processor can thus achieve high computational efficiency with both high-DLP and low-DLP applications. The major architectural challenge for VLT is the increased instruction issue bandwidth required by the concurrent threads.

The major contributions of this paper are:

- We evaluate a set of alternative implementations that address the VLT requirements for instruction issue bandwidth. We show that the area requirements of most of these alternatives are low.
- We demonstrate that VLT improves the performance of an 8-lane vector processor for applications with medium and short vectors by factors of 1.4 to 2.3 on top of the speedup provided by vectorization.
- We show the potential of VLT with pure scalar threads for applications that can be parallelized but do not vectorize. For several applications, a VLT-enhanced vector processor provides twice the performance of a CMP with two multithreaded wide-issue processors.

Overall, VLT improves the efficiency of vector processors for applications that traditionally challenge their computational capabilities. Hence, it improves their applicability to a wider range of parallel applications and makes them a practical alternative to conventional CMP designs.

The rest of the paper is organized as follows. Section 2 reviews the architecture of a modern vector processor. Section 3 introduces VLT. Sections 4 and 5 present the VLT design alternatives for short-vector threads and scalar threads, respectively. Section 6 presents our methodology and Section 7 the experimental evaluation. Section 8 presents related work, and Section 9 concludes the paper.

2 Modern Vector Architecture

Figure 2 presents the block diagram of a modern vector processor. For practical reasons, we focus on a high-end vector design [13]. Nevertheless, VLT is equally applicable to the simpler in-order multi-lane vector designs used in embedded domains. The processor consists of three major components: the scalar unit (SU), the vector unit (VU), and the on-chip memory system [28]. The scalar unit is a superscalar processor with its own first-level caches. Like processors for PCs and servers, the SU supports wide-issue, out-of-order (OOO), and speculative execution; register renaming; and multiple functional units. The SU fetches both scalar and vector instructions but renames, schedules, and executes only scalar instructions. Its reorder buffer tracks both scalar and vector instructions for precise exceptions.

The vector unit consists of the parallel lanes and the vector control logic (VCL). Vector lanes provide register file and execution resources for vector instructions [2]. The vector register file may contain more vector registers than the number specified in the ISA in order to facilitate vector register renaming [12]. The elements of each vector register are

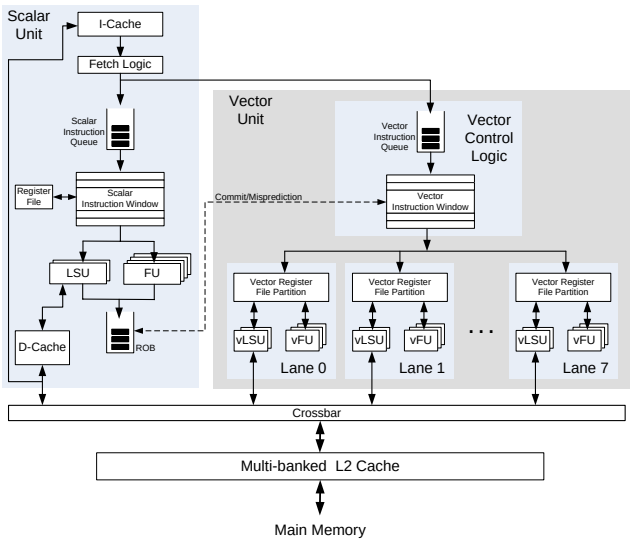

Figure 2. Block diagram of a modern vector processor.

distributed across lanes in a round-robin manner. Each lane contains an arithmetic datapath for each vector functional unit, as well as address generation, translation, and queuing resources for each vector memory port. To execute a vector instruction, the control logic applies identical control to all lanes so that they execute multiple element operations in parallel every cycle. For an arithmetic instruction with vector length of 64, an 8-lane VU takes 8 clock cycles to execute it on one of the vector functional units.

The vector control logic implements out-of-order execution of vector instructions using hardware structures similar to those in the SU (instruction queue, renaming, instruction window) [12]. Because vector instructions specify multiple element operations, each instruction occupies a vector functional unit for several cycles even in multi-lane implementations. Hence, the VCL structures are much simpler than their scalar counterparts because a lower instruction issue rate is sufficient in the vector unit (typically one or two instructions per cycle). The VCL also communicates with the SU to resolve vector-scalar dependencies, to retire instructions, and to recover from branch mispredictions.

The on-chip memory system includes a large L2 cache. Since the vector unit can tolerate memory latency, it accesses the L2 directly to avoid thrashing in the small L1 cache. The L2 is highly associative and highly banked to provide a large number of ports for strided and indexed vector accesses [13]. Coherence between the L1 and the L2 is maintained by hardware as described in [28]. Proper vector-vector and scalar-vector access ordering is maintained using compiler-generated memory barriers.

3 Vector Lane Threading (VLT)

Applications with long vectors utilize multiple lanes well, even with relatively low instruction issue rates in the VU. For example, Figure 2 shows an 8-lane vector unit with 5 vector functional units (3 arithmetic units and 2 load/store ports). If all vector instructions define more than 40 element operations, the datapaths in all lanes can be kept busy with

an issue rate of just 1 vector instruction per cycle. Unfortunately, applications with short or medium vector lengths cannot saturate the vector datapaths as easily. If the vector length is smaller than the number of lanes (1 to 7), a vector instruction can never utilize all datapaths across the 8 lanes. Applications with medium vector lengths (8 to 40) require a much higher instruction issue rate to fully utilize all vector lanes. For example, at a vector length of 16, each instruction executes in 2 cycles on 8 lanes, requiring an issue bandwidth of 2 to 3 instructions per cycle to fully utilize all 5 vector functional units. Finally, the vector lanes remain completely idle when running non-vectorizable code, regardless of the issue rate available in the vector unit.

3.1 VLT Overview

The idea behind VLT is to allow underutilized lanes to run multiple threads from a single parallel application. Even though a single thread may not have high DLP, we can create the effect of long vectors by running multiple threads in parallel. The number of vector lanes assigned to each thread depends on its DLP. VLT can run 2 threads of medium vector lengths with 4 lanes per thread, or 4 threads of short vector lengths with 2 lanes per thread. Alternatively, VLT can execute 8 scalar threads (no vectors) with 1 lane per thread. The result is fast execution of the application code on all available hardware resources.

VLT takes advantage of the fact that most parallel applications contain multiple levels of parallelism. In a loop nest, a vectorizing compiler can pick only one of the loops to vectorize for DLP. VLT exploits additional parallelism across other loops in the nest. VLT also allows vector resources to be used with loops that are parallel but not vectorizable or with code that has pure task-level parallelism (e.g., producer-consumer). In short, VLT enables the use of vector lanes to exploit both DLP and TLP in a flexible manner. Therefore, we can continue scaling the number of lanes in vector processors without incurring low performance efficiency for parallel applications without significant amounts of DLP.

Furthermore, VLT breaks the vector length vs. stride trade-off when vectorizing loop nests. Ideally, the compiler tries to select for vectorization a loop that provides both long vectors (maximum hardware utilization) and unit-stride accesses (minimum memory stalls). With loop nests, it is often the case that one loop provides long vectors and another provides unit-stride accesses. VLT allows the compiler to vectorize the loop that leads to unit-stride accesses and multithread across the other loop to get high utilization of the available vector lanes.

A vector processor with VLT is different from a simultaneous-multithreaded (SMT) vector processor [11]. An SMT-based vector design runs multiple threads to better utilize idle vector functional units to the lack of ILP in a single thread. For a processor with multiple vector functional units, low ILP can occur even in code with long vectors. VLT uses multiple threads to better utilize idle vector lanes to the lack of DLP in a single thread. VLT and SMT solve orthogonal problems in a vector processor and could be combined in a single design.

3.2 Hardware Requirements

VLT requires a single modification to the lane design. We must provide separate control signals to the lanes assigned to each thread. There is no need for additional registers, as each lane already contains a large register file to store vector register elements for applications with long vectors. With VLT, we use the otherwise idle lanes to execute multiple threads, so we can use the register files in these lanes to store the register values for the additional threads. When using 2 VLT threads, a 32-entry vector register file with 64 elements per register is used as two 32-entry vector register files with 32 elements per register. VLT requires no changes to the memory system. The memory hierarchy in a vector processor is already designed to support the large number of accesses generated by the lanes. VLT places additional pressure on non-sequential memory bandwidth, but such access patterns are already supported for large stride or indexed accesses with long vectors.

VLT does require additional support in the scalar unit and the vector control logic. For vector threads, the scalar unit must fetch instructions for all threads and execute any scalar instructions. Additionally, the vector control logic must schedule and issue instructions for all threads. There are two general ways to handle both challenges: *replication* and *multiplexing*. Replication suggests the use of multiple scalar units or multiple vector control logic blocks. Multiplexing suggests sharing the existing scalar unit or vector control resources between multiple threads. Replication provides higher performance at increased area overhead, while multiplexing offers the opposite trade-off. Our experiments indicate that a multiplexed VCL with statically partitioned resources across all threads performs as fast as a replicated one and introduces negligible area overhead (a few multiplexors). For the scalar unit, the trade-off is more interesting, and we analyze it in Section 4 for threads with short- or medium-length vectors and in Section 5 for scalar threads.

3.3 Software Requirements

VLT requires no significant ISA modifications to run multiple vector threads. Modern vector ISAs support parallel vector systems and include all instructions necessary for a multithreaded API. For the evaluation in Section 7, we extended a commercial vector ISA by a single configuration instruction that associates each thread with a subset of the available vector lanes.

VLT is applied selectively to sections of the program with low DLP. For high-DLP phases, the program can still use all lanes with a single vector thread. Hence, VLT does not have a negative impact on program phases that already execute efficiently on a vector processor. The program can also use a different number of VLT threads in different phases, depending on the DLP available in each phase. Switching the number of threads may require saving and restoring vector registers to and from memory. For the applications in Section 6, switching was performed at the boundaries of large parallel regions at which the vector registers did not contain live values.

To use VLT, a programmer must parallelize loops or tasks within the application code. In the case of loop nests, the

user can generate threads from an outer loop using OpenMP directives. Vectorization of the inner loops is automatically performed (when possible) by a vectorizing compiler. Overall, the combination of the familiar OpenMP model with automatic vectorization provides for a fairly intuitive parallel programming environment. Alternatively, the user can use a general thread package to parallelize the outer loop.

4 Vector Threads

When executing 2 or 4 vector threads with VLT, the hardware appears to the programmer as a system with 2 or 4 vector processors, each with a smaller vector unit (4 or 2 vector lanes). The original design has sufficient vector lanes to support this illusion. VLT must still support scalar processing and vector control for the additional vector threads.

4.1 Design Space for Vector Threads

As explained in Section 3, the major design choice for supporting vector threads is whether to replicate or multiplex the scalar unit (SU). Replicating the scalar unit creates a CMP-like system with 2 SUs that share the vector unit and the L2 cache. Multiplexing the scalar unit is similar to an SMT processor that can support 2 contexts [10]. For 4 vector threads, the design space includes an additional, hybrid point: two SMT processors sharing the vector unit.

The choice of scalar unit used in the case of replication adds an additional dimension to the design space. The SU in the base vector processor for this study is a 4-way superscalar design. When introducing an additional SU, one may choose to use an extra 4-way core or a smaller 2-way processor. Again, the trade-off is between area overhead and performance benefit. For VLT, the smaller SU will only be used when two threads are available. Hence, using a smaller second SU does not impact the non-VLT portions of the application (pure scalar or long vectors). If the second scalar unit is not identical to the first one, we essentially have a heterogeneous CMP as the base of the VLT system [21].

4.2 Area Considerations

The performance differences between replicated and multiplexed SUs are analyzed in Section 7. Here, we quantify the area trade-off using a first-order approximation based on published area data for the Alpha architecture processors. We study the Alpha family because it includes three generations of superscalar processors (21064, 21164, 21264) [21] and the Tarantula vector extension for the Alpha 21464 [13]. To produce the area estimates for VLT, we studied the die photos and area breakdowns for the Alpha processors, adjusted their areas to account for any differences in cache sizes and functional unit mixes, and scaled them to 0.10μm CMOS technology.

Table 2 presents the area estimates for the major components of a vector processor. The overall die area is typically dominated by the on-chip L2 cache and the vector lanes. These resources make up approximately 86% of the base vector design used in Section 7. Table 2 shows the estimated percent increase in area of various VLT configurations over the base vector design. All presented configurations use a single, multiplexed VCL. The notation

	Area (mm^2)
2-way scalar unit + L1 caches	5.7
4-way scalar unit + L1 caches	20.9
2-way VCL	2.1
Vector lane	6.1
L2 cache (4MB)	98.4
Base vector processor (4-way SU, 8 vector lanes)	170.2

Table 1. Area breakdown for vector processor components.

Configuration		**% Area Increase**
V2-SMT	(2 VLT threads, 1 SMT SU)	0.8%
V4-SMT	(4 VLT threads, 1 SMT SU)	1.3%
V2-CMP	(2 VLT threads, 2 SUs)	12.3%
V2-CMP-h	(2 VLT threads, 2 heter. SUs)	3.4%
V4-CMP	(4 VLT threads, 4 SUs)	26.9%
V4-CMP-h	(4 VLT threads, 4 heter. SUs)	10.1%
V4-CMT	(4 VLT threads, 2 STM SUs)	13.8%

Table 2. Percentage area increase over the base vector processor for various VLT configurations.

Vn-$\{SMT, CMP, CMT\}${-h}$\}$ implies a VLT vector processor that supports n vector threads using a multiplexed (SMT), replicated (CMP), or hybrid (CMT) scalar unit. The optional $-h$ suffix implies that heterogeneous scalar units are used, with the first one being a 4-way processor and all the others being 2-way processors. The area estimates assume a 6% and 10% area penalty for 2-way and 4-way multithreading within a scalar processor [26].

The overhead of VLT is less than 2% for two or four vector threads ($V2$-SMT or $V4$-SMT) when the SU is multiplexed. Replicated configurations for two vector threads with identical ($V2$-CMP) or heterogeneous ($V2$-CMP-h) scalar units lead to area overheads of 13% and 4%, respectively. Supporting 4 vector threads is expensive if four 4-way SUs are used (37% for $V4$-CMP). However, four vector threads become practical if we use heterogeneous scalar units (10% overhead for $V4$-CMP-h) or two multithreaded scalar units (4% for $V4$-CMT). Overall, the area estimates show that several VLT configurations for both 2 and 4 vector threads are possible at an area overhead of less than 5%. The VLT area overhead decreases further as the on-chip L2 cache becomes larger, a common trend with modern processors.

5 Scalar Threads

For code that can be parallelized but does not vectorize, it is tempting to use the 8 vector lanes to run 8 scalar threads. The vector lanes have several of the resources that one would find in a simple scalar unit, such as registers, functional units, and memory ports. Nevertheless, executing scalar threads on the vector lanes is challenging, as it creates an instruction issue bandwidth problem. If each

Scalar Unit	Superscalar out-of-order processor 4-way instruction fetch/issue/retire 64-entry instruction window and ROB 4 arithmetic units, 2 memory ports 16-KByte, 2-way associative, L1 caches
Vector Control	2-way issue, 32-entry VIQ 32-entry vector instruction window
Vector Lane (8 replicas)	3 arithmetic units, 2 memory ports 64 physical vector registers (8 elements/lane)
Memory System	4-MByte L2 cache 4-way associative, 16-way banked 10 cycles hit, 100 cycles miss penalty

Table 3. The base vector processor parameters.

scalar thread consumes instructions at the rate of two per cycle, the total instruction issue bandwidth required in the vector unit is 16 instructions per cycle. Such a high rate is impossible to meet, since the vector unit is fed instructions by a 4-way scalar unit.

To enable scalar execution, we re-engineer the vector lanes to include a small instruction cache and sequencing logic. With these enhancements, each lane can operate independently as a 2-way in-order processor. Out-of-order execution within a lane is not possible without introducing additional structures, such as an instruction window and a reorder buffer. Since each lane can still access the L2 cache, per-lane data caches are not necessary. The latency of the L2 cache is not a major performance issue, since the lanes already include queuing resources for access decoupling [14].

We studied the case of scalar thread execution on the vector lanes using a 4-KByte instruction cache in each lane. The small cache is appropriate for threads generated from tight nested loops. The cache has the same capacity as the vector register file partition in the lane, but is significantly smaller, as it only requires a single access port. Instruction cache misses in the lanes are forwarded to the scalar unit for service as L1 instruction cache misses. Exceptions in each scalar thread are precise and are reported to the operating system by interrupting the scalar unit.

An alternative to executing scalar threads with VLT is to build a CMP system out of multiple scalar units. For example, the $V4$-CMT *without* the vector unit is a 2-core CMP system that can execute up to two scalar threads per core. Its area is 13% smaller than that of the base vector design and 26% smaller than the VLT version of $V4$-CMT. However, such a CMP design offers fewer execution resources than running the threads on the vector lanes. Two scalar cores in our evaluation presented in Section 7 include a total of 8 arithmetic datapaths. On the other hand, the 8 vector lanes combined include 24 arithmetic datapaths, up to 16 of which can be utilized with 2-way instruction fetch per lane. Similarly, the 8 lanes contain twice as many memory ports as two 4-way scalar units.

6 Methodology

Table 3 presents the parameters of the simulated base vector design. The processor executes the Cray X1 instruction set [8], which defines 32 vector registers with 64 64-bit elements per register. The base scalar unit is a 4-way superscalar processor with first-level caches. The vector control logic supports 2-way instruction issue (out-of-order) on the 8 vector lanes. The peak arithmetic performance of the vector unit is 24 64-bit operations per cycle (3 functional units times 8 lanes). A base processor with 16 vector lanes [13] would increase the usefulness of VLT for low-DLP applications. For a 2-way SU, we use identical caches but half the resources of the 4-way unit.

Table 4 presents the applications used in this study. For each application we list the percentage of vectorization, the average vector length in operations, the most common vector lengths, and the percentage of execution time on the base vector processor that is amenable to VLT (*opportunity*). The top two applications (`mxm` and `sage`) have long vectors and fully utilize the 8 vector lanes as shown in Figure 1. We do not evaluate VLT with these two applications, since there is no opportunity for optimization and the VLT system performs exactly as the base system. The next set includes applications that are vectorizable but have medium or short vectors (`mpenc`, `trfd`, `multprec` and `bt`). These applications can be accelerated using 2 to 4 vector threads. The final three applications are parallel but not vectorizable by an automated compiler (`radix`, `ocean`, and `barnes`). For these applications, We evaluate scalar threads executing on the vector lanes.

We compiled all benchmarks using the production C and Fortran compilers for the Cray X1 system with all optimizations for both scalar and vector code. Vectorization was automatic without any manual guidance. On several occasions, the compiler was able to perform significant loop nest restructuring, which led to longer vectors than initially expected. In other words, the baseline code is highly optimized. Threads were identified manually in low-DLP regions of each benchmark. The threads are coarse-grained, typically in the millions of instructions. All processor configurations, base and VLT, were simulated on a detailed execution-driven simulator for X1-based parallel vector systems. The simulator accurately models the behavior and timing of all components in a vector system: scalar unit(s), vector control, vector lanes, and caches.

7 Evaluation

This section evaluates the performance potential of VLT first for vector threads and then for pure scalar threads.

7.1 Vector Thread Analysis

Figure 3 presents the speedup of VLT over the base vector processor for the four applications with vector threads. We use the $V2$-CMP configuration for 2 threads (two 4-way scalar units) and the $V4$-CMP configuration for 4 threads (four 4-way scalar units). These configurations represent the maximum performance potential, as fully replicated resources best address the instruction issue requirements of VLT. We discuss area-efficient configurations in the following paragraphs. With 2 threads, the VLT speedup ranges from 1.14 to 2.15. With 4 threads, the VLT speedup ranges from 1.40 to 2.3. These speedups are in addition to the performance improvements already achieved by vectorization

Name	Description	% Vect	Avg VL	Common VLs	% Opportunity
mxm	dense matrix multiply	96	64.0	64	–
sage	hydrodynamics modeling	94	63.8	64	–
mpenc	video encoding	76	11.2	8, 16, 64	78
trfd [6]	two-electron integral transformation	73	22.7	4, 20, 30 ,35	99
multprec	multiprecision array arithmetic	71	25.2	23, 24, 64	81
bt [3]	block tridiagonal benchmark	46	7.0	5, 10, 12	70
radix	radix sort	6	62.3	24, 52, 64	90
ocean [32]	eddy currents in ocean basin	-	-	-	96
barnes [32]	galaxy system simulation	-	-	-	98

Table 4. Characteristics of the applications studied. "*% Vect*" is the percentage of vectorization measured in operations. "*Avg VL*" is the average vector length and "*Common VLs*'" lists the most common vector length values. "*% Opportunity*" is the percentage of execution time on the base processor that VLT can potentially accelerate with multiple threads.

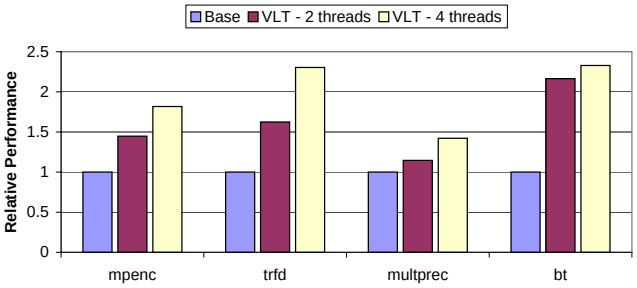

Figure 3. The VLT speedup for vector threads over the base vector processor.

in the base design. The exact speedup for each application depends on its average vector length and the opportunity for VLT. The average vector length indicates how many lanes are available for execution of additional threads. The opportunity indicates the portion of the original execution time that could be accelerated by multithreaded execution in the lanes. For example, `mpenc` has an average vector length of 11, which indicates that only 2 to 4 vector lanes are efficiently used in the original configuration. With the potential for 1 to 3 additional threads and a 78% opportunity, `mpenc` should achieve an overall speedup of 1.6 to 2.3. Our results indicate that `mpenc` reaches a speedup of 1.8. Secondary factors that affect the observed speedup include caching effects and the thread API overhead.

Figure 4 provides insight into the performance advantage of VLT. It presents the normalized utilization of the arithmetic datapaths in the vector lanes during the execution of each application. There are 24 arithmetic datapaths, three per lane. On every clock cycle, a datapath may be busy with an element operation; stalled because of dependencies or insufficient vector instruction issue bandwidth; or idle. We separate two cases of idle datapaths. First, all 8 datapaths in a vector functional unit may be idling due to a complete lack of vector instructions. Second, some of the 8 datapaths may be idling due to very small vector lengths. Figure 4 shows that VLT compresses the program execution by allowing multiple vector operations to issue and execute in parallel. The number of stall and idle cycles is reduced and speedups of up to 2.3 are achieved. Nevertheless, a significant number of stall and idle cycles remains, due to pure sequential portions in the application and imbalances in the functional unit mix.

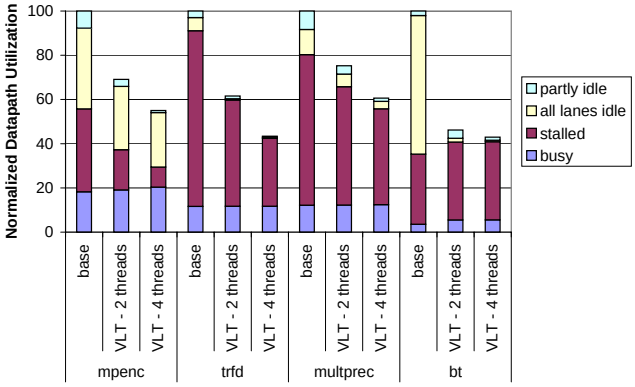

Figure 4. The datapath utilization in the 8 vector lanes for the base and VLT designs. The utilization is normalized over the execution on the base configuration. A lower bar implies faster execution time. Each lane contains 3 arithmetic datapaths.

Figure 5 shows the performance of VLT with two and four vector threads for all scalar unit configurations. The notation for each configuration is explained in Section 4. All performance numbers represent speedup of different implementations of VLT over the base vector design. For two vector threads, there is no significant difference between the replicated ($V2$-CMP) and the smaller multithreaded ($V2$-SMT) scalar unit. With four threads, on the other hand, the scalar unit configuration becomes a significant factor. A single multithreaded SU ($V4$-SMT) is no longer sufficient for best performance because 4 instructions per cycle are not sufficient for 4 vector threads. Fortunately, the hybrid scalar unit configuration $V4$-CMT (2 SUs, each 2-way threaded) performs as well as the expensive in area, fully replicated $V4$-CMP (4 scalar units). This result indicates that an issue rate of 8 instructions per cycle is sufficient for 4 vector threads and that full performance can be achieved at a reasonable area overhead (13% area increase for $V4$-CMT for up to 2.3x performance improvement).

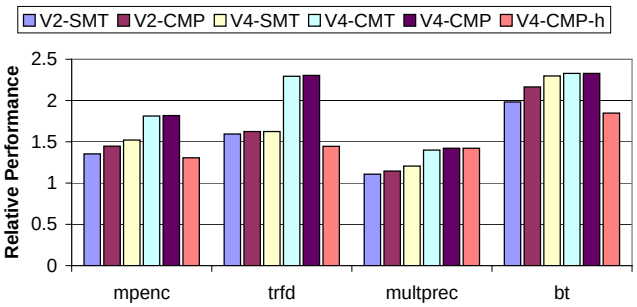

Figure 5. Performance evaluation of the design space for vector threads.

Figure 5 also presents the performance achieved with the heterogeneous $V4$-CMP-h configuration for the scalar unit (one 4-way processor and three 2-way processors). It provides speedup over the base design (1.3 to 1.8) but performs worse than all other VLT configurations. Combined, the 4 cores in $V4$-CMP-h provide sufficient instruction issue bandwidth for 4 vector threads ($4 * 1 + 3 * 2$). However, a thread running on one of the 2-way scalar units is always restricted to an issue rate of 2 instructions per cycle, which may be insufficient for program phases with very short vectors or a low degree of vectorization. In addition, for SPMD applications that use barrier synchronization, like `radix`, `ocean`, and `barnes`, performance is determined by the slowest thread. In contrast, VLT configurations like $V4$-CMT allow two threads to flexibly share the instruction issue rate of a 4-way scalar unit.

7.2 Scalar Thread Analysis

Figure 6 evaluates the potential of VLT with scalar threads running on the vector lanes. Essentially, each vector lane operates as a 2-way in-order processor. The system operates like an 8-processor CMP with very simple cores [18]. We do not use the scalar unit to run a 9th scalar thread, since our thread library requires that the number of threads be a power of two. This is not a fundamental limitation.

We compare the execution of scalar threads in the vector lanes using VLT to the execution of scalar threads on the replicated scalar units that were already introduced to support vector threads. Specifically, we compare the $V4$-CMT configuration against the same configuration without the vector unit and the VLT support (CMT). This is essentially a system with two 4-way superscalar cores, each capable of simultaneously executing up to two threads. As shown in Figure 6, VLT provides twice the performance of the purely scalar design for `radix` and `ocean`. Since each thread in these applications includes only a limited amount of instruction-level parallelism, it is better to run 8 threads on 8 very simple processors, rather than 4 threads on 2 wide-issue processors. VLT thus provides a significant performance advantage. On the other hand, each scalar thread for `barnes` underperforms in the vector lanes, so despite VLT's higher number of threads, VLT and parallel execution on the CMT system have equal performance.

We should note that for applications with long vectors (`mxm` and `sage`), the performance advantage of the vector design over the CMT is nearly proportional to the ratio of their peak throughputs (8 to 1). VLT retains the vector performance advantages over CMP for long vector applications and allows the vector design to reach or exceed the CMP performance for non-vectorizable applications. Overall, VLT provides good performance for vector threads with medium and short vectors at reasonable area overheads. The $V4$-CMT VLT configuration provides speedup of 1.4 to 2.3 over an aggressive vector processor. For scalar threads, VLT exhibits significant performance potential for certain applications, but additional work is necessary to achieve significant benefits in all cases.

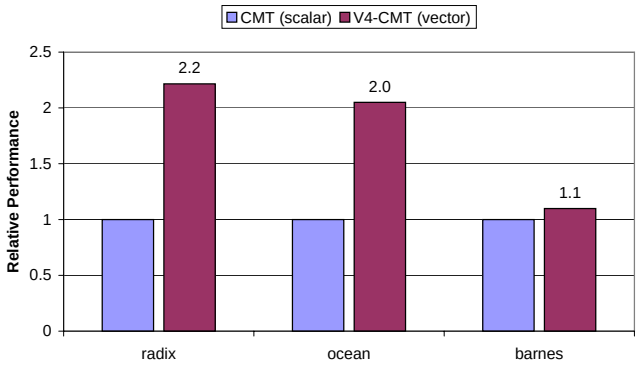

Figure 6. Performance for 8 VLT scalar threads running on the vector lanes over 4 scalar threads running on the scalar units of the V4-CMT configuration (2 4-way superscalar processors, 2-way threaded).

8 Related work

Mateo Valero, Roger Espasa and their colleagues developed and evaluated several concepts for advanced vector processors: decoupled execution of vector instructions [14], out-of-order execution of vector instructions [12], SMT vector processors [11], and vector processors with wide-issue scalar units [28]. Our work builds on techniques in [14, 12, 28] and is orthogonal to [11], as explained in Section 3. Asanović quantified the design complexity and performance benefits of multi-lane vector processors [2]. These techniques are currently used in advanced vector processors such as the Cray X1 [9], the NEC SX-6 [16, 30], and the proposed Alpha Tarantula [13]. Our work utilizes multiple lanes for applications with limited DLP.

The idea of running threaded code on vector resources was first introduced by Chiueh in [7], but his work did not consider multiple lanes, the central concept in our study. The Scale architecture can also facilitate vector-like or threaded execution on an array of tightly coupled processors with register-to-register communication [20]. The Scale philosophy is to enhance a tightly coupled CMP to run vector code efficiently, while we approach the same issue by enhancing a vector processor to run threaded code efficiently. The evaluation in [20] uses application code with assembly for the Scale instruction-set, while we use binaries produced by a vectorizing compiler from high-level code. The IBM Cell processor can execute statically scheduled SIMD threads on its 8 SPUs, orchestrated by control

code running on a superscalar processor [27]. If all SPUs operate in lock-step, the 8 SPUs behave as a vector unit for long vectors.

This work also draws on techniques developed for chip multiprocessors [23] and simultaneous-multithreaded processors [10] for scalar instruction sets.

9 Conclusions

We presented vector lane multi-threading, a technique that uses idle vector resources in a vector processor to execute additional threads with short vectors or no vectors at all. VLT improves the utilization of DLP resources similarly to the way SMT improves the utilization of ILP resources. In other words, VLT allows a vector processor to efficiently exploit thread-level parallelism with existing DLP hardware. VLT is practical to implement and provides significant performance advantages for applications that have phases with short vectors or no vectors.

VLT helps manufacturers of vector systems to continue increasing the number of lanes to scale vector performance. This scaling technique works well for applications with high amounts of DLP, as it provides greater performance without increasing the instruction issue bandwidth requirements. With the addition of VLT, the new vector lanes can be efficiently used for parallel applications that do not have long vectors over their entire execution. In practice, VLT allows vector processors to be an competitive alternative to CMP designs for a wide range of parallel applications.

Acknowledgments

This work was supported by the DARPA HPCS program through Cray Inc. Suzanne Rivoire and Rebecca Schultz were also supported through a Stanford Graduate Fellowship and an NSF Graduate Fellowship, respectively.

References

[1] B. An et al. Implementation of MPEG-4 on Philips Co-Vector Processor. In *14th Workshop on Circuits, Systems and Signal Processing*, Nov. 2003.

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

[3] D. Bailey et al. NAS Parallel Benchmark Results. Technical report, NASA Ames Research Center, 1993.

[4] M. Baron. Intrinsity Arrays 2GHz Adaptive Matrix. *Microprocessor Report*, May 2002.

[5] M. Baron. Sandbridge Blasts Off at MPF. *Microprocessor Report*, Nov. 2002.

[6] M. Berry et al. The PERFECT Club Benchmarks: Effective Performance Evaluation of Supercomputers. *Intl. Journal on Supercomputing Applications*, 1989.

[7] T. Chiueh. Multi-Threaded Vectorization. In *Proceedings of the 18th Intl. Symp. on Computer Architecture*, May 1991.

[8] Cray Inc., Seattle, WA. *Cray Assembly Language (CAL) for Cray X1TM Systems Reference Manual*, 2003.

[9] T. Dunigan et al. Early Evaluation of the Cray X1. In *Proceedings of the Intl. Conf. on Supercomputing*, Nov. 2003.

[10] S. Eggers et al. Simultaneous Multithreading: A Platform for Next-Generation Processors. *IEEE Micro*, 17(5):12–19, Sept. 1997.

[11] R. Espasa et al. Multithreaded Vector Architectures. In *Proceedings of the 2nd Intl. Symp. on High-Performance Computer Architecture*, Feb. 1996.

[12] R. Espasa et al. Out-of-order Vector Architectures. In *Proceedings of the 30th Intl. Symp. on Microarchitecture*, Dec. 1997.

[13] R. Espasa et al. Tarantula: A Vector Extension to the Alpha Architecture. In *Proceedings of the 29th Intl. Symp. on Computer Architecture*, May 2002.

[14] R. Espasa and M. Valero. Decoupled Vector Architecture. In *Proceedings of the 2nd Intl. Symp. on High-Performance Computer Architecture*, Feb. 1996.

[15] T. Halfhill. Floating Point Buoys Clearspeed. *Microprocessor Report*, Nov. 2003.

[16] Japan's Ministry of Science and Technology. GS40 - Earth Simulator Vector Supercomputer. http://www.es.jamstec.go.jp/esc/eng/.

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

[18] P. Kongetira. A 32-way Multithreaded SPARC Processor. In *Conf. Record of Hot Chips 16*, Aug. 2004.

[19] C. Kozyrakis et al. Vector vs. Superscalar and VLIW Architectures for Embedded Multimedia Benchmarks. In *Proceedings of the 35th Intl. Symp. on Microarchitecture*, Nov. 2002.

[20] R. Krashinsky et al. The Vector-Thread Architecture. In *Proceedings of the 31st Intl. Symp. on Computer Architecture*, June 2004.

[21] R. Kumar et al. Single-ISA Heterogeneous Multi-Core Architectures. In *Proceedings of the 36th Intl. Conf. on Microarchitecture*, Dec. 2003.

[22] M. Levy. Making the Chip Right for Imaging. *Microprocessor Report*, June 2002.

[23] B. Nayfeh et al. The Case for a Single-Chip Multiprocessor. In *Proceedings of the 7th Intl. Symp. on Architectural Support for Programming Languages and Operating Systems*, Oct. 1996.

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

[25] L. Oliker et al. Scientific Computations on Modern Parallel Vector Systems. In *Proceedings of the Intl. Conf. on Supercomputing*, Nov. 2004.

[26] P. Peston et al. Design of an 8-wide Superscalar RISC Microprocessor with Simultaneous Multithreading. In *Proceedings of the Intl. Solid-State Circuits Conf.*, Feb. 2002.

[27] D. Pham et al. The Design and Implementation of a First-Generation Cell Processor. In *Digest of Technical Papers of the Intl. Solid-State Circuits Conf.*, Feb. 2006.

[28] F. Quintana et al. Adding a Vector Unit to a Superscalar Processor. In *Proceedings of the Intl. Conf. on Supercomputing*, June 1999.

[29] S. Shungu et al. A 26.58 TFLOPS Global Atmospheric Simulation with the Spectral Transform Method on the Earth Simulator. In *Proceedings of the Intl. Conf. on Supercomputing*, Nov. 2002.

[30] H. Takahara et al. NEC SX Series and Its Applications to Weather and Climate Modeling. In *Proceedings of the 4th Intl. Workshop on Next Generation Climate Models*, Mar. 2002.

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

[32] S. Woo et al. The SPLASH2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd Intl. Symp. on Computer Architecture*, June 1995.