The Case for VOS: The Vector Operating System

Vijay Vasudevan, David G. Andersen, Michael Kaminsky[1]
Carnegie Mellon University, [1]Intel Labs

Abstract

Operating systems research for many-core systems has recently focused its efforts on supporting the scalability of OS-intensive applications running on increasingly parallel hardware. Lost amidst the march towards this parallel future is efficiency: Perfectly parallel software may saturate the parallel capabilities of the host system, but in doing so can waste hardware resources.

This paper describes our motivation for the Vector OS, a design inspired by vector processing systems that provides *efficient parallelism*. The Vector OS organizes and executes requests for operating system resources through "vector" interfaces that operate on vectors of objects. We argue that these interfaces allow the OS to capitalize on numerous chances to both eliminate redundant work found in OS-intensive systems and use the underlying parallel hardware to its full capability, opportunities that are missed by existing operating systems.

1 Introduction

Over the last decade, computing hardware has promised and delivered performance improvements by providing increasingly parallel hardware. Systems with multi-core CPUs, and GPUs with hundreds of cores have become the norm. The gauntlet has been firmly thrown down at the software community, who have been tasked to take advantage of this increased hardware parallelism.

The operating systems community has largely risen to the challenge, presenting new OS architectures and modifying existing operating systems that enable parallelism for the many-core era [2, 12, 3, 4]. Independently, application writers have rewritten or modified their applications to use novel parallel programming libraries and techniques. Both communities have improved software without changing how applications request OS resources.

In this paper, we argue that OS-intensive parallel applications, including many of today's driving datacenter applications, must be able to express requests to the operating system using a vector interface in order to give the OS the information it needs to execute these demands efficiently. To not do so will make "embarrassingly parallel" workloads embarrassingly wasteful. We outline a new design, the Vector OS (VOS), that is able to let OS-intensive applications be not just parallel, but *efficiently* parallel.

Consider a modern webserver that receives a new connection using the `accept()` system call. `accept()` returns exactly one connection from a list of pending connections. The application then performs a series of sequential operations to handle the connection—setup, application-specific processing, and teardown. A high-load webserver may be serving many requests in parallel, but each request follows similar execution paths, wasting resources executing redundant work that could be shared across concurrent requests.

In VOS, work is executed using vector operations that are specified through vector interfaces like `vec_accept()` (which returns multiple connections rather than just one), `vec_open()`, `vec_read()`, `vec_send()`, etc. Exposing vector interfaces between applications and the operating system improves efficiency by eliminating redundant work; moreover, vector interfaces open the door to new efficiency opportunities by allowing VOS to more effectively harness vector and parallel hardware capabilities already present in many machines, such as SSE vector instructions and GPUs.

This paper argues that vector interfaces are critical to achieving efficient parallelism, thus requiring changes to the OS interface that applications program to. But this departure from a decades-old interface raises several questions and challenges that are as exciting as they are difficult: Should developers explicitly specify work in terms of vectors of resources? If not, how should vector execution be hidden from the programmer without introducing excessive complexity into the OS? What are the semantics of vector system call completion? Should they be synchronous or asynchronous, and how should they handle exceptions? We address many of these questions in this paper, but do not yet have answers to all of them. Nonetheless, we believe that the Vector OS can enable OS-intensive applications to make the maximum use of today's increasingly vector and parallel hardware.

2 The Parallel Landscape

Hardware parallelism is growing. Processors continue to gain speed by adding cores; graphics engines improve rendering performance by adding shaders; solid state

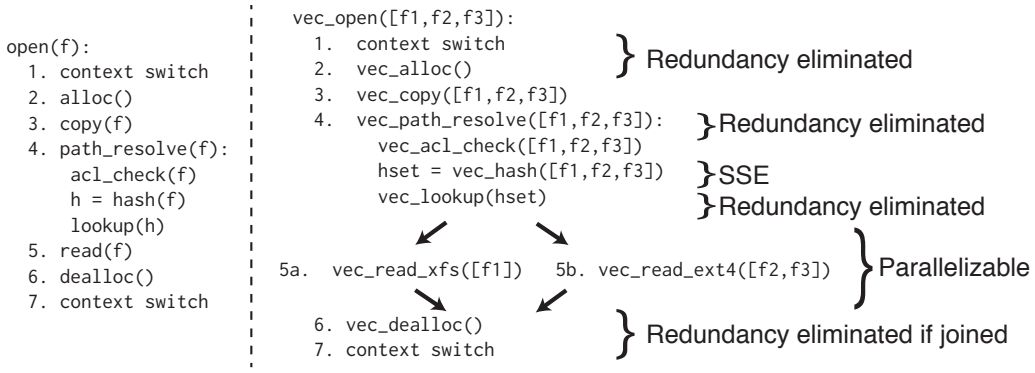


Figure 1: Pseudocode for open() and proposed vec_open(). vec_open() provides opportunities for eliminating redundant code execution, vector execution when possible, and parallel execution otherwise.

drives improve throughput by adding more flash chips; and CPUs increase throughput by making their vector instructions, such as SSE, even wider.[1] These trends are expected to continue into the foreseeable future, barring a revolution in device physics.

OS-intensive, parallel applications are adapting to parallel hardware. `memcached` moved from single-threaded to multi-threaded; event-based webservers such as `node.js` and Python Twisted are improving support for massive concurrency; languages and threading libraries such as OpenMP, Cilk, Intel Thread Building Blocks, and Apple's Grand Central Dispatch library all encourage applications to break work into smaller independent tasks, which the libraries can then assign to cores.

Parallel operating systems are making parallel execution possible. Recent work on improving operating system support for parallelism has made great strides in *enabling* parallel execution of code by removing locks, improving cache locality and utilization for data, carefully allocating work to processors to minimize overhead, and so on [4, 10, 12, 7].

Parallelism is necessary, but not necessarily efficient. While parallel execution on parallel hardware may sound ideal, it completely ignores *efficiency*, a metric that measures how much of a system's resources are necessary to complete some work [1]. We therefore turn our attention to "efficient parallelism," or making the best overall use of resources on a parallel system.

3 Vectors will be Victors

Providing efficient parallelism requires that operating systems do the following:

1. *Eliminate redundancy*: Identify what work is common or redundant across a set of tasks and execute that work exactly once.
2. *Vectorize* when possible: Work should be expressed as operations over vectors of objects so it can be both executed efficiently using hardware techniques such as SSE and better optimized by compilers.
3. *Parallelize* otherwise: If code paths diverge, continue to execute work in parallel.

Consider the example of the `open()` system call in Figure 1 (left) simplified from the Linux 2.6.37 source code. Opening a file requires context switches, memory operations (e.g., copying the filename into a local kernel buffer in `alloc()`), performing access control checks, hashing file names, directory entry lookup, and reads from a filesystem. Today, simultaneous `open()` calls can mostly be executed in parallel, but doing so would not be as efficient as it could be.

Figure 1 (right) shows the set of operations required for `vec_open()`, which provides a vector of filenames to the operating system. In this example, file `f1` exists in an XFS filesystem while the other two reside in an ext4 filesystem. The code path is similar to `open()`, with the exception that the interfaces are capable of handling vectors of objects, e.g, `vec_hash()` takes in several filenames and returns a vector of hashes corresponding to those filenames. These vector interfaces package together similar work and can improve efficiency by using the techniques described above.

Eliminating Redundancy: When provided vectors of objects, a vector system call can share the common work found across calls. Examples of common work include context switching, argument-independent memory allocations, and data structure lookups. `alloc()` is an argument-independent memory allocation that returns a page of kernel memory to hold a filename string; the `vec_alloc()` version need only fetch one page if the length of all filename arguments fits within that page, eliminating several additional page allocations necessary for each file if processed one by one. `vec_path_resolve()` requires traversing the directory tree, performing ACL checks and

[1] Intel's Advanced Vector Extensions support 256-bit wide register operations.

lookups along the way. If files share common parent directories, resolution of the common path prefix need only occur once for all files instead of once per file.

As yet another example, `lookup(hash)` must search a directory entry hash list to find the one entry that matches the input hash. In `vec_lookup(hset)`, the search algorithm need only traverse the list once to find all entries corresponding to the vector of hashes. While this search can be performed in parallel, doing so would needlessly parse the hash list once for each input hash, wasting resources that could be dedicated to other work.

Vector interfaces provide general redundancy-eliminating benefits (e.g., reducing context switches), some of which can be provided by other batching mechanisms [10]. But vector interfaces also can enable specialized algorithmic optimizations (e.g., hash lookup) because all operations in a batch are the same, even though the operands may differ.

HW Vectorizing: Certain operations can be performed more efficiently when they map well to vector hardware already available but underutilized by existing operating systems. An implementation of `vec_hash()` might use SSE instructions to apply the same transformations to several different filenames, which may not be possible when dealing with precisely one filename at a time.

Parallelizing: Not all work will perfectly vectorize throughout the entire execution. In our `vec_open()` example, three files are stored on two different filesystems. While most of the code can be vectorized because they share common code paths, performing a `read()` from two different filesystems would diverge in code execution. Both calls can occur in parallel, however, which would use no more resources than three parallel `read()` calls; in this case it would probably use fewer resources because there will exist common work between files f2 and f3 within `vec_read_ext4()`.

When code paths diverge, the system should automatically determine whether to join the paths together. In Figure 1, there is an optional barrier before `vec_dealloc()`—should both forked paths complete together, joining the paths can save resources in executing deallocations and context switches. But should they diverge in time significantly, it may be better to let the paths complete independently.

3.1 Quantifying Redundant Execution

How much redundancy exists for OS-intensive parallel workloads running on highly parallel hardware? We explore this question by looking at the system call redundancy of Apache 2.2.17 server web requests between four different files in two different directories. We use `strace` to record the system calls executed when serving a single HTTP GET request for each file. Each request was traced in isolation and we show the trace for the request with the median response time out of five consecutive requests for each file.

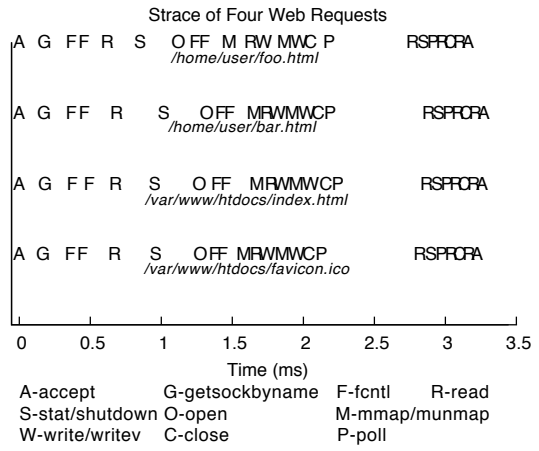


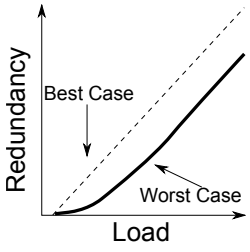
Figure 2: Trace of four different web requests serving static files shows the same system calls are always executed, and their execution paths in time are similar.

Figure 2 shows when each system call was executed for each request for these four different static files. Each request invoked the same set of system calls regardless of which file was served, and the timings between system calls were similar, with variances attributable to OS scheduling. This simple experiment demonstrates that real applications provide opportunities to take advantage of the benefits of vector interfaces.

3.2 Scaling Redundancy With Load

Efficiency matters most when parallel systems are operating at high load, so we must understand how redundancy scales as a function of incoming load. We argue that at high load, redundancy is abundantly available, making the Vector OS approach an appealing solution.

To illustrate this concept, the inline figure shows how offered load affects the redundancy available in a parallel system. Here, we define redundancy loosely as the amount of work that is identical to some unique work currently being done in the system. In the best case, each additional request is identical to existing requests in the system, and requests arrive simultaneously; redundancy therefore increases linearly with load.



In the worst case, requests are uncoordinated: they differ in both type and/or arrival time. At low load, redundancy is thus hard to find. As load increases, redundancy

increases for several fundamental reasons. First, a system has a finite number of different types of tasks it is likely to execute; a similar argument has been made for applications running on GPUs [6]. By the pigeonhole principle, redundancy must increase once a request of each type already exists in the system. In addition, high load systems are increasingly becoming specialized for one type of workload (databases and filesystems, caching, application-logic, etc.), further shrinking the number of distinct types of work. Distributed systems often partition work to improve data locality or further increase specialization within a pool of servers, again increasing the chance that a high-load system will see mostly homogeneous requests.

These homogeneous requests may also arrive closely in time: Many systems already batch work together at the interrupt level to cope with interrupt and context switching overheads, creating an under-appreciated opportunity. Today's operating systems make limited use of this interrupt coalescing, but this mechanism plays a more fundamental role in the Vector OS: It provides the basis for nearly automatic batching of I/O completions, which are delivered to the application and processed in lockstep using convenient vector programming abstractions. These "waves" of requests incur additional latency only once during entry into the system.

4 Towards the Vector OS

The Vector OS must address several difficult interface and implementation challenges. The design must expose an appropriate set of vector interfaces to allow application writers to both easily write and understand software running on parallel hardware. VOS must also organize and schedule computation and I/O to retain vectorization when possible and parallel execution otherwise.

Batching imposes a fundamental tradeoff between efficiency gains by having a larger batch, and adding latency by waiting for more requests to batch. However, at higher loads, this tradeoff becomes a strict win in favor of vectorization, as the efficiency gains enable *all* requests to execute faster because they spend less time waiting for earlier requests to complete. VOS must therefore make it easy for applications to dynamically adjust their batching times, much as has been done in the past for specific mechanisms such as network interrupt coalescing. Fortunately, even at lower load, Figure 2 shows that calls that arrive together (e.g., subsequent requests for embedded objects after an initial Web page load) present a promising avenue for vectorization because their system call timings are already nearly identical.

4.1 Interface Options

Today's system call interface destroys opportunities for easy vectorization inside the kernel—system calls are synchronous and typically specify one resource at a time. VOS must be able to package similar work (system calls, internal vector function calls) together to be efficiently parallel. We enumerate several interface possibilities below and describe their impact on the efficiency versus latency tradeoff.

1. Application-agnostic changes: One way to provide opportunities for making vector calls without application support is to introduce system call queues to coalesce similar requests. An application issues a system call through libc, which inserts the call into a syscall queue while the application waits for its return. Upon a particular trigger (a timeout, or a number threshold), VOS would collect the set of requests in a particular queue and execute the set using a vector interface call—this implementation can build upon the asynchronous shared page interface provided by FlexSC [10]. Another approach is to rewrite program binaries to submit multiple independent system calls as one multi-call by using compiler assistance [9, 8]. Both approaches transparently provide VOS with collections of system calls which it could vectorize, but these approaches have several drawbacks: First, applications do not decide when to issue a vector call, so they cannot override the timing logic built into the OS or compiler, leading to a frustrating tension between application writers and OS implementers. Second, a single thread that wishes to issue a set of similar synchronous system calls (e.g., performing a bunch of `read()` calls in a for loop), will still execute all reads serially even if there exists no dependence between them.

2. Explicit vector interface: Applications can help VOS decide how to coalesce vector calls by explicitly preparing batches of similar work using the vector interface to system calls, such as `vec_open()`, `vec_read()`, etc. VOS can use this knowledge when scheduling work because it knows the application thread will be blocked until all submitted work completes. This assumes these vector calls are synchronous, though a non-synchronous completion interface (e.g., return partial results) may be useful for some applications.

As an example of how an application might use explicit vector interfaces, the core event-loop for an echo server may look as follows (in simplified pseudocode):

```
fds = vec_accept(listenSocket);
vec_recv(fds, buffers);
vec_send(fds, buffers);
```

As the application processing between `vec_recv()` and `vec_send()` becomes more complicated, raw vector interfaces may prove difficult to use. The benefit is that the OS is relieved of deciding how and when to vectorize, leaving the application as the arbiter for the efficiency versus latency tradeoff and eliminating that complexity from the OS.

3. Libraries and Languages: Although our focus is on the underlying system primitives, many applications may be better served by library and language support built atop those primitives. Several current event-based language frameworks appear suitable for near-automatic use of vector interfaces, including `node.js` and Python Twisted. Programs in these frameworks specify actions that are triggered by particular events (e.g., a new connection arrival). If the actions specified are side-effect free, they could be automatically executed in parallel or even vectorized as is done with some GPU programming frameworks such as CUDA. System plumbing frameworks such as SEDA [11] that use explicit queues between thread pools also present a logical match to underlying vector interfaces, with, of course, non-trivial adaptation.

Finally, the progress in general-purpose GPU programming is one of the most promising signs that programming for vector abstractions is possible and rewarding. Both CUDA and OpenCL provide a "Single Instruction Multiple Thread" (SIMT) abstraction on top of multi-processor vector hardware that simplifies some aspects of programming these systems. Although programmers must still group objects into vectors, the abstraction allows them to write code as if the program were a stream of instructions to a single scalar processor. We believe that the amazing success of GPGPUs in high-performance computing is a telling sign that programmers who want performance are willing and able to "think vector" in order to get it.

5 Discussion

We believe that restructuring for a vector OS is needed to improve the efficiency of OS-intensive parallel software, but to do so brings challenges and accompanying opportunities for OS research:

Heterogeneity: As the multiple-filesystem example of `vec_open()` showed, VOS will need to move efficiently between single-threaded execution of shared operations, vector execution of SIMD-style operations, and parallel execution when code paths diverge. We believe it is clear that forcing the OS programmer to manually handle these many options is infeasible; VOS will require new library, language, or compiler techniques to render the system understandable. In addition to code divergence, VOS's code should also be appropriately specialized for the available hardware, be it a multi-core CPU, an SSE-style CPU vector instruction set, an integrated GPU or a high-performance discrete GPU.

Vector interfaces and parallel I/O: In contrast to the "bad news" of heterogeneity, I/O presents a more rosy picture for VOS. Today's I/O devices themselves increasingly require large numbers of outstanding requests and batched I/O issuing and completion—characteristics that make them perfectly tailored for VOS. Vector interfaces such as `vec_read()` can drastically reduce the overhead needed to send many requests to a high-performance solid-state drive, for example; recent research by the authors and others has observed that the system overhead for these requests can cripple the performance of fast SSDs [5]. Batched I/O completion and interrupt coalescing is necessary for both network and fast storage devices [7], and the batch arrivals create the basis for returning vectors of completed I/O operations to applications with substantially reduced overhead.

6 Conclusion

The Vector OS aims to improve the efficiency of OS-intensive parallel applications by exposing and using OS interfaces that operate on vectors of objects. Doing so builds upon existing natural opportunities for work coalescing to and from I/O devices; completing the loop by extending large batches of completed work to applications in turn gives the OS the ability to fully harness the underlying capabilities of vector and parallel hardware, as well as to substantially reduce the amount of redundant work performed in executing its duties. Deciding how to expose applications to these vector interfaces remains an open question: We hope that this work spurs further research in finding the best way to contain OS and application complexity while exploiting a powerful set of OS vector interfaces.

References

[1] E. Anderson, J. Tucek. Efficiency matters! In *Proc. HotStorage*. Oct. 2009.

[2] A. Baumann, et al. The multikernel: A new OS architecture for scalable multicore systems. In *Proc. 22nd ACM Symposium on Operating Systems Principles (SOSP)*. Oct. 2009.

[3] S. Boyd-Wickizer, et al. Corey: An operating system for many cores. In *Proc. 8th USENIX OSDI*. Dec. 2008.

[4] S. Boyd-Wickizer, et al. An analysis of linux scalability to many cores. In *Proc. 9th USENIX OSDI*. Oct. 2010.

[5] A. M. Caulfield, et al. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. In *IEEE Micro*. Dec. 2010.

[6] M. Garland, D. B. Kirk. Understanding throughput-oriented architectures. *Communications of the ACM*, 53(11):58–66, Nov. 2010.

[7] S. Han, et al. PacketShader: a GPU-accelerated software router. In *Proc. ACM SIGCOMM*. Aug. 2010.

[8] A. Purohit, et al. Cosy: Develop in user-land, run in kernel-mode. In *Proc. HotOS IX*. May 2003.

[9] M. Rajagopalan, et al. Cassyopia: Compiler assisted system optimization. In *Proc. HotOS IX*. May 2003.

[10] L. Soares, M. Stumm. FlexSC: Flexible system call scheduling with exception-less system calls. In *Proc. 9th USENIX OSDI*. Oct. 2010.

[11] M. Welsh, D. Culler, E. Brewer. SEDA: An architecture for well-conditioned, scalable Internet services. In *Proc. 18th ACM Symposium on Operating Systems Principles (SOSP)*. Oct. 2001.

[12] D. Wentzlaff, et al. An operating system for multicore and clouds: Mechanisms and implementation. In *Proc. 1st ACM Symposium on Cloud Computing (SOCC)*. Jun. 2010.