Virtualizing Local Stores

by Henry M. Cook

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor D. Patterson
Research Advisor

(Date)

* * * * * * *

Professor K. Asanović
Second Reader

(Date)

Abstract

Software-managed local stores have proven to be more efficient than hardware-managed caches for some important applications, yet their use has been mostly confined to embedded systems that run a small set of applications in a limited runtime environment. Local stores are problematic in general-purpose systems because they add to process state on context switches, and because they require fast data memory close to the processor that might be better spent on cache for some applications. We introduce virtualized local stores as a mechanism to provide the benefits of a software-managed memory hierarchy in a general-purpose system. A *virtual local store* (VLS) is mapped into the virtual address space of a process and backed by physical main memory, but is stored in a partition of the hardware-managed cache when active. This reduces context switch cost, and allows VLSs to migrate with their process thread. The partition allocated to the VLS can be rapidly reconfigured without flushing the cache, allowing programmers to selectively use VLS in a library routine with low overhead.

1 Introduction

A memory hierarchy is the standard solution to the difficult tradeoffs between memory capacity, speed, and cost in a microprocessor memory subsystem. Performance is critically dependent on how well the hierarchy is managed. Hardware-managed *caches* (Fig. 1(a)) hide the memory hierarchy from software, automatically moving data between levels in response to memory requests by the processor. At the other extreme, software-managed *local stores* (Fig. 1(b)) expose the hierarchy, requiring software to issue explicit requests to a DMA engine to move data between levels. Although explicit management imposes an additional burden on software developers, application-specific software management of a local store can sometimes improve efficiency substantially compared to the fixed policy of a hardware-managed cache [31].

Software-managed local stores are widely used in embedded systems, where they help improve performance and performance predictability, as well as reducing cost and power dissipation. However, they are rarely employed in general-purpose systems. Whereas embedded processors often run only one or a few fixed applications tuned for the underlying hardware design, general-purpose processors usually run a multiprogrammed and time-varying mix of large, complex applications, often with large amounts of legacy code. One major challenge to the adoption of local stores in general-purpose computing is that the local store represents a large increase in user process state that must be saved and restored around operating system context switches. Another challenge is that a local store is only beneficial for some pieces of code, whereas for others, a cache would be more beneficial. As only a limited amount of fast memory can be placed close to the processor, any fixed partitioning between cache and local store (Fig. 1(c)) will inevitably be suboptimal for many use cases, as we show in Section 4. Existing schemes to support dynamic partitioning of on-chip memory between cache and local store have significant reconfiguration overhead [11, 26], and were primarily designed for embedded systems where a single long-lived partition is often adequate.

In this paper, we propose a new approach to software-managed memory hierarchies. Our goal is to provide the advantages of a software-managed local store where it is beneficial, without complicating software (both application and operating system) when it is not. We accomplish this by virtualizing the local stores: Instead of a dedicated memory structure, a *virtual local store* (VLS) is just a segment of the physical main memory cached in a dynamically created partition of the hardware-managed cache in the same way as any other piece of physical memory (Fig. 1(d)). Software management is achieved through user-level memory copy instructions, which provide a simple software interface to a high-throughput DMA engine that transfers data between main memory and the cached VLS segment. Like conventional software-managed local stores, the VLS model improves performance compared to conventional hardware-managed caches by reducing memory traffic, cache pollution, and cache coherence traffic. Unlike conventional local stores, the VLS model does not impact software that does not want to use software management and retains conventional hardware-managed caches' support for software-transparent migration of a process' data to physical main memory or to a different core after a context switch.

Given the trend towards providing larger numbers of simpler cores in chip-scale multiprocessors (CMPs), software-managed memory hierarchies may prove to be an important component to help improve per-core performance, while reducing energy consumption and improving utilization of scarce cross-chip and off-chip memory bandwidth. In addition, software-management can provide more predictable performance, which can be useful in future manycore systems where some cores are running real-time critical tasks, such as interfacing with I/O devices. Although there

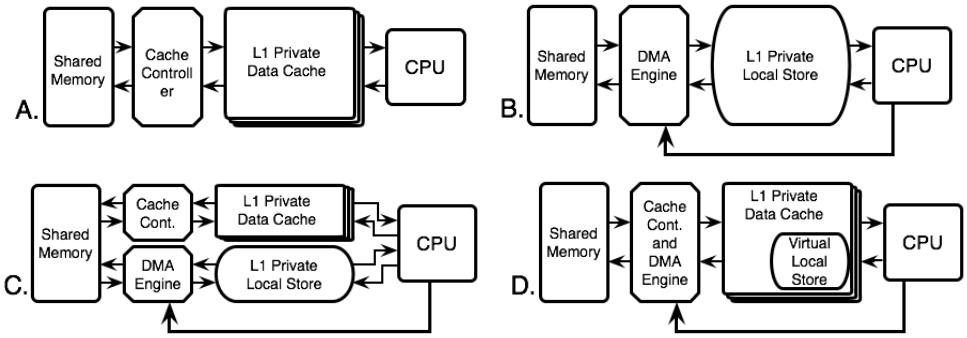


Figure 1: Alternative on-chip memory organizations: a) conventional hardware-managed data cache, b) pure local store architecture, c) separate cache and local store, d) virtual local store. The figure omits instruction caches and only shows alternatives for the innermost level of the memory hierarchy.

has been considerable earlier work in local stores and configurable memory hierarchies, we believe VLS is the first approach to make all the benefits of software-managed memory hierarchies available within a full-featured mainstream computing environment.

2 Background and Motivation

Hardware-managed data caches have been successful because they require no effort on the part of the programmer and are effective for many types of applications, but local stores can provide significant advantages over a cache for certain types of code when the programmer is prepared to expend the programming effort [30, 31]. We begin by reviewing the advantages of a local store architecture, and will later show how most of these are retained even if the local store is implemented as a VLS partition within a cache:

- **Simpler access:** A cache has to perform an associative search to locate the correct way within a set, whereas a local store simply uses the request address to directly access a single memory location, saving energy and potentially reducing latency.
- **Flexible placement:** Set-associative caches constrain the set in which data can reside based on its address in main memory, leading to conflict misses if many frequently accessed locations map to the same set, even when there is sufficient total capacity elsewhere in the cache. In addition, caches quantize storage in units of cache lines, and require that cache lines begin at naturally aligned memory addresses, leading to wasted capacity for small data objects that straddle cache line boundaries. Local stores allow data of any size from main memory to be placed anywhere in their data buffer, increasing the effective size of the data buffer.
- **Simpler high-bandwidth transfers:** To maintain full utilization of the main memory bandwidth, a non-blocking cache controller must be used to allow multiple cache line requests to be in progress simultaneously. These requests can be generated by a complex processor that can issue multiple independent requests before the first has been satisfied, or from an autonomous hardware prefetcher that predicts what future lines will be needed by the processor. For some applications, prefetchers are accurate and beneficial, but for other applications, prefetch mispredictions can reduce performance and increase energy consumption compared to no prefetching. Hardware prefetchers are also necessarily limited in scope; virtually no prefetchers will cross a page boundary, and they must generally experience a few misses before deciding to activate. A DMA engine is much simpler to implement, and allows even simple processor designs to fully utilize the main memory bandwidth. DMA engines can also move data in units less than a full cache line, providing greater efficiency in interconnect and memory (although many DRAMs have a minimum practical burst size).

- **Controllable replacement policy:** On a cache miss, an old line must be evicted from the cache to make room for the new line. A poor match between the fixed cache replacement heuristics and the application's memory access pattern can lead to a large increase in memory traffic compared to the ideal behavior. With a local store, software has complete control over replacement.
- **Predictable access time:** Because the cache is managed automatically by hardware, it is difficult for a programmer to determine the performance that will be achieved for a given piece of code, as this depends on the cache controller state and the operation of any hardware prefetchers. With a local store, software has full control over both the contents of the data buffer and when the buffer is accessed, making execution time much more predictable.

All of these characteristics give architectures that incorporate local stores a performance or energy advantage whenever it is productive for developers to engage in software memory management. Of course, there are many algorithms for which software memory management is challenging to program or inefficient at runtime. Furthermore, modern hardware prefetchers are complex enough to accurately capture and predict the behavior of many kernels. For these reasons, many architects are resistant to the idea that physical local stores should be included in any architecture for which they are not absolutely required.

VLS resolves this tension between efficiency and productivity by providing all of the desirable characteristics listed above, while requiring only extremely minimal changes to the baseline cache hardware and operating system. Since VLS consumes no resources when inactive, it does not interfere with the effectiveness of the hardware cache management, and when activated it is no less effective than a physical local store would have been. Giving the programmer control over the memory management model allows them to make the productivity/efficiency tradeoff on a per-routine basis, while allowing them to always fall back on robust hardware support.

3 Virtual Local Store Mechanisms

We begin by describing the mechanisms needed to support virtual local stores within a cached virtual memory architecture, and the implications that these mechanisms have for the programming model and OS behavior. To simplify our explanation, we focus only on the data portion of the memory hierarchy, and first assume a uniprocessor system before later expanding the design to a cache-coherent multiprocessor.

3.1 Address Space Mapping

Virtual local stores are implemented on top of traditional hardware-managed, physically-tagged caches. Figure 2 shows how the current thread running on a core may have an associated local VLS allocated in physical memory and mapped into its virtual address space. The local, physical cache may contain software-managed VLS *stored* data in addition to regular hardware-managed *cached* data. We refer to the underlying hardware cache memory structure as the *data buffer* to prevent confusion when different portions of the structure are allocated to different uses. The stored data in the VLS is accessed using regular loads and stores within the VLS virtual address range, avoiding modifications to the base ISA and allowing library code to be oblivious to whether it is operating on cached or stored data.

One approach to implementing the VLS address mapping is to simply add an extra VLS bit to page table entries in the TLB. For example, this approach was used in the XScale processor to indicate accesses should use the "spatial mini-cache" instead of the main cache [11]. The disadvantages of this approach are that every VLS access involves an associative lookup in the TLB which consumes energy (for example, 8% of StrongARM power dissipation was in the data TLB [19]), the replacement policy of the TLB might evict VLS entries, and all page table entries in memory and the TLB are made larger.

We propose adding a more efficient mapping mechanism, which dedicates a large segment of virtual addresses (shown as "VLS" in Figure 2) for the current user thread's VLS and adds a separate VLS physical base address register, `pbase`, to each core to provide the base address of the current VLS in physical memory. The value of `pbase` is set by the operating system, and must be saved and restored on thread context switches. Figure 3 shows that a check of the high-order bits of the effective address early in the processor pipeline can determine whether any given access is targeting the VLS or regular cacheable memory. This technique is similar to how many systems determine

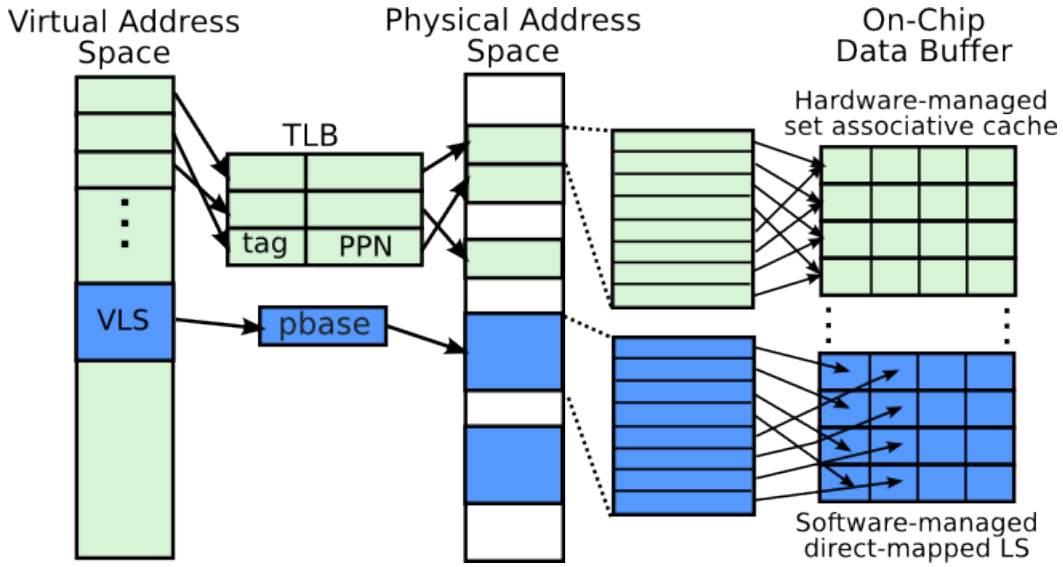


Figure 2: Mapping of virtual local stores from the virtual address space to physical pages, and how data in those pages is indexed in the on-chip memory hierarchy. PPN is a "physical page number". The pbase register is an optional optimization to remove the need to access the TLB on a local VLS access. To simplify diagram, set-based partitioning (see Section 3.2) is shown, but way-based partitioning is used in evaluated design.

translated versus untranslated memory ranges, so the full TLB lookup (and any TLB misses) can be avoided for VLS accesses. We also protect against VLS accesses outside the bound of currently allocated VLS physical memory by using a bound register, called `pbound` in Figure 3.

Every access to the VLS must check cache tags to see if the associated VLS line is present in the underlying hardware cache. The programming model for a local store assumes it will be a contiguous region of memory that fits in the local data buffer. Figure 2 show how we take advantage of this contiguity to reduce cache access energy by using a direct-mapped placement policy for VLS accesses. Because we recognize a VLS access early in the pipeline, we can disable the readout of tags and data from other ways in the set-associative cache to save energy (Figure 3). This approach is similar to Selective Direct-Mapping [2], but because we only place VLS data in the data buffer at the direct-mapped location, we do not have to make a way prediction and never have to search the other ways on a VLS cache miss. The VLS uses the same data path and pipeline timing as regular cache accesses to avoid complicating the hardware design.

3.2 Data Buffer Partitioning

The mechanisms presented so far do not treat stored versus cached data differently for purposes of replacement. This indifference would be adequate if all routines were either purely cache or purely local store-based. However, most software-managed code usually benefits from using a regular cache for some accesses, e.g., global variables or the stack. Requiring all of these to be first explicitly copied into local store would impose a significant performance and programmer productivity penalty. On the other hand, arbitrarily mixing cached and stored data would effectively revert back to a conventional cache design. We therefore desire to dynamically allocate a partition of the cache to exclusive VLS use, leaving it unaffected by any regular cache accesses.

One option is to use way-based partitioning [22] to allocate entire ways of the underlying set-associative structure to either the cache or local store aspects. Way-based partitioning only affects cache replacement decisions, and hence only adds logic to the non-critical victim selection path. A miss on cached data is not allowed to evict a line from the ways allocated to the VLS, but otherwise follows the conventional replacement policy. In contrast, a miss on VLS data will evict the current occupant of its direct-mapped location in the cache regardless of what type of data it holds. VLS accesses are always direct mapped, but we require cache accesses to search all ways, including the VLS partition, to

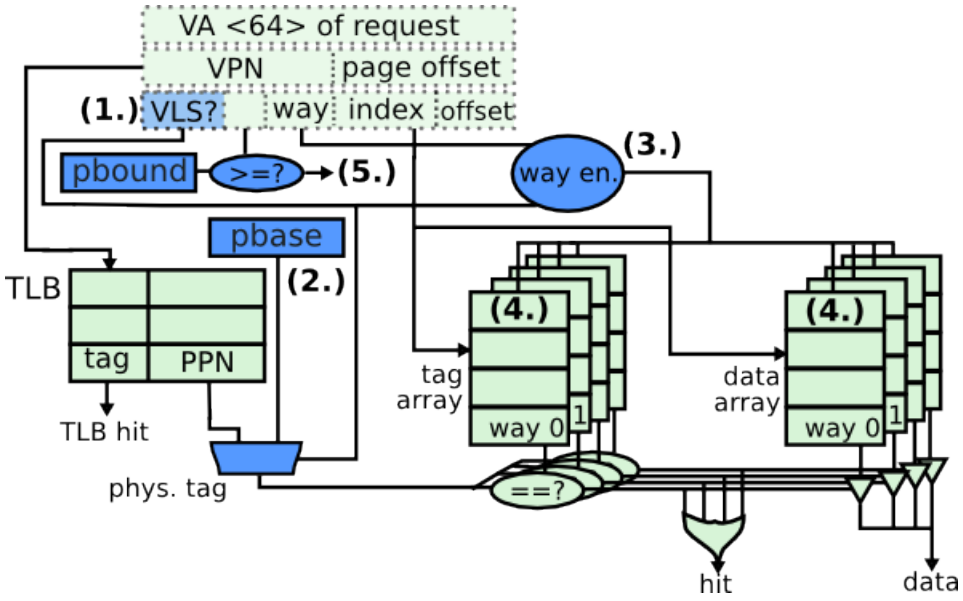


Figure 3: Overview of mechanisms used in virtualizing the local stores: 1.) Request is identified based on target address as being a VLS access; 2.) `pbase` holds physical base address, used instead of TLB lookup; 3.) direct-mapping means multiple way lookups can be avoided on a VLS access; 4.) Replacement policy respects partitioning; 5) `pbound` holds number of physical pages allocated to VLS, to provide protection.

reduce reconfiguration overhead. The main disadvantage of way-partitioning is the reduction in associativity of the cache partition.

The programmer configures the desired partition size using a special user-level instruction. This reconfiguration is a very light-weight operation as it simply sets a register in the cache controller to modify its replacement policy; current cache contents are unaffected. The typical usage would be for a library routine to enable the VLS partition, perform the computation, then disable the VLS partition. Because regular cache accesses still search all ways, any cached data that happens to already exist in a newly activated VLS partition will still be accessible during execution of the local store-using routine. Only the number of VLS lines dynamically required by the local store routine will be affected during the routine's execution, reducing the side effects of coarse granularity partitioning mechanisms. When the VLS partition is released at the end of the local store routine, any 'stored' data can freely be replaced by the regular cache replacement algorithm. If the local store routine is re-executed soon thereafter, it will likely find some or all of the data previously placed in the VLS still resident in data buffer. This light-weight reconfiguration is key to enabling selective and incremental use of software memory management in library routines.

It is possible to construct a similar scheme around set-based partitioning or per-line flags [7], which would have the advantage of preserving associativity of the cache partition and which could potentially allow a finer granularity of partitioning. However, set-based partitioning is complex to implement in practice as it inserts logic in the critical set index path. There is also a larger reconfiguration overhead as partitions have to be flushed when the set indexing scheme changes to maintain memory coherence. Partitioning based on per-line flags would allow for extremely fine-grain sharing, but requires mechanisms to flash clear the flags on VLS deactivations [7]. The evaluations in Section 4 use the simpler way-based partitioning mechanism as this worked well in practice.

3.3 Data movement mechanisms

The other important component of a software-managed architecture is an efficient mechanism to move data to and from memory. Regular loads and stores via processor registers could be used to load the VLS. However, it would be difficult for a simple processor to keep enough loads and stores in flight to saturate the bandwidth-latency product of the main memory system while simultaneously computing on the data. The Cray-2 was an early example of a system

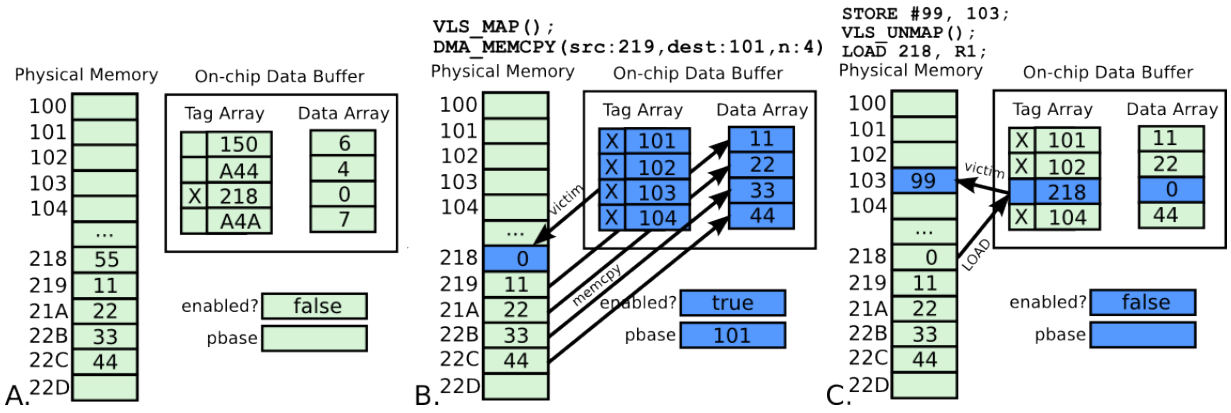


Figure 4: Interaction of the `memcpy` and VLS mechanisms. Initially the data buffer contains hardware-managed cache data. Then a VLS is mapped into the virtual address space. A `memcpy` targeting the VLS address range creates a new copy of the source data in the VLS, evicting the cached data. Changes made to the data in the VLS via store instructions are not reflected in the original location, since this data is a new copy. When the VLS is unmapped, the stored data may be evicted to its backing location in memory.

with a local store without DMA capability, but this machine instead had vector instructions to provide high bandwidth transfers to and from memory [3].

VLS's use of the virtual address space lends itself to a natural `memcpy`-like interface for transmitting data to or from the virtual local stores. We therefore provide a user-level DMA engine as part of the cache controller that shares functionality with the regular cache miss datapath and prefetch logic. This engine provides basic functions for performing unit-stride, constant stride, and scatter-gather transfers between arbitrary virtual memory addresses. More extensive DMA capabilities could potentially be included to support more complicated patterns of data movement. DMA transfers can proceed in parallel with continued processor execution, and a fence mechanism is provided to synchronize the two units.

Unlike a conventional software prefetch or hardware-managed cache prefetch, the transfered data's location in the data buffer is explicitly specified by the DMA request's destination address, and is not tied to the source data's address in memory. This quality is important to software developers using VLS to efficiently pack data into the limited space available in the on-chip data buffers.

Each data item copied from memory by the DMA engine is given a new VLS-associated virtual and physical address, preserving the original source copy; Figure 4. This quality preserves the local store semantics expected by software developers familiar with software memory management. Writes made to the copy of the data in the VLS are not propagated back to the original copy in memory, unless the programmer explicitly copies the data back (using either stores or DMA transfers).

Numerous optimizations are possible in the interaction of the DMA engine with a memory hierarchy that supports VLS. The goal of such optimizations is to take advantage of programmer intentionality expressed in terms of DMA transfer requests by avoiding superfluous evictions and refills.

The DMA engine can avoid generating memory refills for any VLS destination data buffer lines that miss in the data buffer but will actually be entirely overwritten by the DMA transfer. This optimization prevents the underlying hardware-managed mechanisms from causing more refills than would occur in a purely software-managed system.

Other optimizations come in to play when a data buffer is partitioned between cache and VLS. For a memory-to-VLS transfer, the desired DMA behavior is to *not* place the source read data in the cache partition of the data buffer. Only the new VLS copy of the data is allocated space in the data buffer. This behavior is unlike how a `memcpy` operation would occur on a purely hardware-managed system, where both the source data and the new copy would be brought into the cache. By moving data into the VLS, the programmer is explicitly stating their intention to use that copy specifically, so there is no point in displacing data from the rest of the data buffer to make room for the original copy as well.

The same argument applies to transfers of data from VLS to memory. The destination data that is being written

back to should not be allocated space in the cache partition of the data buffer. Writing back the data from the VLS to memory does not evict the VLS copy, so if the updated data is useful it is already still present. If the programmer writes back the VLS data and then deactivates the VLS, they are clearly expressing knowledge that the data will not be reused in the short term. In other words, if it would be worthwhile to allocate space for the updated destination data, it would have been worthwhile to simply keep the VLS active.

It is worth noting that these allocation policies apply only to the level of the data buffer in which the VLS in question is resident; outer level shared buffers may choose to allocate space for these 'superfluous' source or destination copies. See Section 3.6.3.

VLS also enables new DMA commands that can improve the performance of software-managed memory implementations of certain pieces of code. For example, some update-in-place software-managed routines must always write back the data they operated on, even if the data is not always modified. A hardware-managed version would only write back dirty data. Since VLS is built on top of hardware which detects dirtiness, our integrated DMA engine might allow for conditional software-managed data transfers.

The integration of a user-level DMA engine with the cache controller logic, and potentially a hardware-managed prefetcher, is the most significant hardware change mandated by the VLS paradigm. However, in terms of hardware resource overheads, we contend that the similarities in datapath and functionality among these three units (cache refill logic, hardware prefetcher, DMA engine) mitigate some of the costs.

3.4 Software Usage Example

To clarify the way VLS is presented to the programmer, we provide the following simplified, unoptimized example of a vector-matrix multiply. The code assumes that at least the vector and one row of the matrix can fit in the physical local data buffer.

```
// A[M] = C[M][N] * B[N]
void vector_matrix_product(int M, int N, int* A, const int* B, const int* C) {
    int* vls_base = (int*) VLS_MAP(2*N*sizeof(int)); //Create a VLS partition
    int* vls_b = vls_base;                           //Assign space for data
    int* vls_c = vls_b + N;
    VLS_DMA_MEMCPY( B, vls_b, N*sizeof(int));        //Prefetch vector
    for( i=0; i < M; i++) {
        VLS_DMA_MEMCPY( &C[i][0], vls_c, N*sizeof(int));//Prefetch matrix row
        VLS_DMA_WAIT();                              //Wait for data
        int sum = 0;
        for( j=0; j < N; j++)
            sum += vls_b[j]*vls_c[j];                //Consume data
        A[i] = sum;
    }
    VLS_UNMAP();
}
```

In this example, the VLS provides a contiguous segment of storage in the local data buffer which the programmer can target using DMA prefetch commands. The programmer does not have to worry about whether the prefetches of rows of the C matrix will evict portions of the B vector, or whether B and C's address ranges conflict in the cache for certain sizes. The elements of the A matrix are accessed on-demand by the hardware management mechanisms, but cannot evict the prefetched elements of B and C.

Obviously, the programmer could now apply further optimizations such as software pipelining and double buffering in order to achieve optimal overlap of communication and computation. The size argument to the `VLS_MAP` call can either be used to allocate a variable-sized VLS partition if supported or simply checked for overflow to aid debugging in VLS systems with a fixed-size partition. Once the math routine is finished, it frees the data buffer space it had been using as a local store by re-enabling cache replacements in that region.

Although the math routine itself might require extensive tuning to make best use of the VLS, any programs which call this routine can remain completely oblivious to the use of VLS and will neither have to rewritten nor recompiled. In addition, if the routine was called for a small vector-matrix operation, only the cached data in the piece of the VLS specifically needed to hold active data would be evicted. The rest of the VLS partition would continue to hold

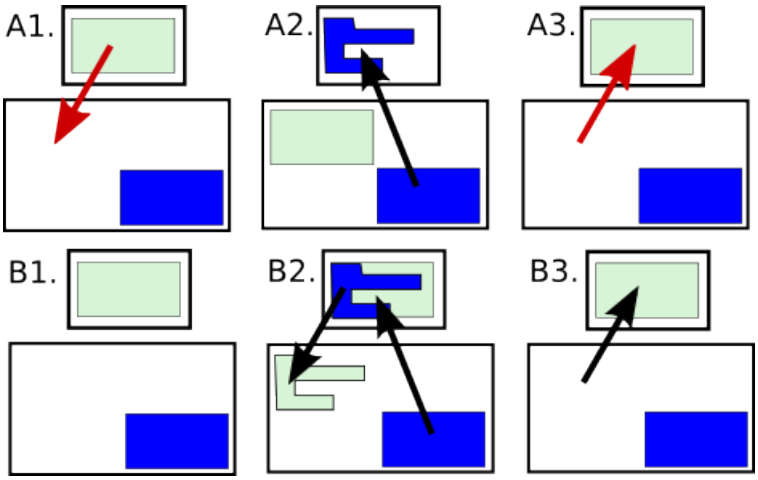


Figure 5: Alternate schemes for handling data placement on process context switches. Green data is in Process 1's VLS. Blue data belongs to Process 2. Red arrows indicate data movement cause by the OS, black arrows data movement caused by the normal hardware cache mechanisms. The smaller, upper box is a private data store and the larger, lower box is a backing memory.

previously cached data that would remain in the local data buffer after the VLS routine returned. This on-demand eviction reduces the performance impact on the caller when a callee uses VLS, as well as the overhead of VLS allocation.

When the operating system deschedules a process from a core, the VLS control registers (`enabled?`, `pbase` and `pbound`) must be saved and restored, but all of the process's physically tagged data may remain in the local data buffer. Previously stored data may be evicted by the incoming process depending on the new process's data access patterns. Evicted stored data is sent to its unique backing location in physical memory. Since different VLSs belonging to different processes are backed by different physical addresses, no OS effort is needed to manage the replacement of one by another.

Upon being restored to the same core, the original process may fortuitously find some of its cached or stored data remaining in the data buffer, and if not, then the data will be naturally restored on-demand by the regular hardware cache refill mechanisms. The virtual address used to access the previously stored-and-evicted data will indicate this data is associated with the process's VLS, and the refill mechanisms will place the data in the correct partition as it is brought back on chip.

3.5 Operating System Context Swaps

Figure 5 illustrates two possible ways to handle VLS data during context switches in a multiprogrammed system. In scheme A, the OS must flush all of Process 1's data (A1) before switching in process 2 (A2). Upon switching back to Process 1, the OS must restore all the data (A3). Scheme B uses only the hardware management mechanisms to evict and restore data on-demand. When switching to Process 2, all the OS has to do is deactivate the VLS partitioning (B1), allowing Process 2's data to evict some of the data left by Process 1 (B2). When Process 1 is restored (B3), the hardware mechanisms take care of bringing back any data that happened to be evicted and restore it to the VLS.

Obviously, a tradeoff exists between proactively flushing the stored data (as would have to be done in a system with a traditional physical local store) and allowing the data to be evicted and restored on-demand. There are three main reasons why on-demand is the best choice for use with VLS.

First, using the already present on-demand refill mechanisms exemplifies the way VLS is meant to fit on top of existing systems. Flushing the local store partition requires additional OS and hardware support.

Second, leaving data in place rather than proactively moving it around can potentially have significant energy savings, depending on the size of the local store partition, the frequency and type of context switches, what will happen to the data in the VLS once it is restored, and how likely the data is to remain in the data buffer in the interim swapped

out period. If most of the data will definitely be evicted, but most will definitely be read again after restoration, then amortizing overhead with a proactive save and restore may be most efficient. However, if the data is unlikely to be evicted, or is not going to be read again or only written over, supporting proactive save and restores is unnecessary.

Third, the other downside of on-demand replacement is the unpredictability in access latencies that will occur when a suspended process is resumed after its stored data has been evicted. While such non-determinism is antithetical to how local stores are traditionally expected to behave, it only occurs after OS-mandated context switches; real-time applications that cannot tolerate such unpredictability should not be being time-multiplexed by the operating system in the first place.

3.6 Multiprocessor Implications and Extensions

In this section, we examine how the VLS mechanism can be straightforwardly extended to multicore processors and multithreaded applications.

3.6.1 Cache Coherence

Data stored in the VLS is physically tagged and participates in the regular hardware-managed cache coherence protocol. One of the advantages often quoted for conventional local stores is that they do not suffer from interference from coherence traffic. Because a VLS contains data backed by a unique location in physical memory, interference would be eliminated in our system as well, provided the underlying coherence mechanism is directory-based (i.e., not snoopy). A directory-based coherence mechanism will not disturb a cache controller unless said cache is actively caching the needed line. Since stored data is always private and uniquely addressed, its presence in an on-chip data buffer will not result in any additional coherence misses or tag checks, other than in one special case.

The special case occurs when a process with a private VLS is migrated to a new core. After the process has been migrated, data in the VLS is moved to the new core's data buffer by the hardware coherence mechanism. If the data has already been evicted from the old data buffer, the coherence mechanism will automatically fetch it from backing memory. We note that only the needed data will be transferred over, and that this will usually represent much less data movement than in a conventional local store system where the entire contents must be transferred to the new core's buffer. As when dealing with context switches, VLS uses already present hardware mechanisms to provide on-demand data movement, at the small price of some slight unpredictability in access times after an already chaotic event.

3.6.2 Multithreaded Applications

If an application is multithreaded, we can allow each thread to have its own private VLS that will track the thread as it moves between cores. If the threads share an address space, supporting multiple VLSs is simply a matter of ensuring that they each have a unique region of the virtual address space (as well as of the physical address space) associated with their VLS. Whichever thread is resident on a given core sets that core's `pbase` register to ensure that the right accesses are identified as being targeted at the VLS. More sophisticated implementations of VLS can provide a richer environment for multi-threaded applications that can benefit from software-management of shared data, as described in the next two sections.

3.6.3 Shared Data Buffers

Virtual local stores can be implemented at multiple levels of the cache hierarchy simultaneously, with each level mapped to a different area of the virtual address space. This flexibility allows for explicit data movement between the different levels of the memory hierarchy, and furthermore allows for *shared* virtual local stores to be created in shared data buffers.

The additional capacity of outer-level shared caches may allow for better management of temporal locality. Some examples of how this capability might be used:

- If each data buffer has its own DMA engine, it is possible to increase the effectiveness of overlapping communication and computation by fetching and compressing large blocks of data into the shared cache and then smaller sub-blocks into the private data buffer.

- In a program that iterates over a large data structure repeatedly, we cannot keep the entire structure in the small private data buffer. However, we can ensure that the structure remains on chip by placing it in a VLS in the shared data buffer. This control improves the efficiency of macroscopic temporal data reuse.

- If we want to explicitly manage a large block of shared data that will be randomly updated by many threads, we can use a VLS to keep the block in the shared cache but allow hardware management to handle the many random updates and coherence among higher level caches.

Another implication that VLS has for shared data buffer design is that the hardware mechanisms of the shared data buffer must implement a policy for handling requests made as a result of DMA transfers moving data into a higher level private VLS. In our current implementation, the outer level data buffer is exclusive with regards to the private data buffers, meaning that data currently in the VLS will never be present in the shared data buffer. However, the shared data buffer may contain data in a currently non-resident VLS that has been evicted by recent activity in the private data buffer in which that VLS was formerly resident.

The shared data buffer will also cache the 'superfluous' source or destination data of a DMA transfer to a private VLS (i.e. the original copy). If the shared data buffer currently contains a shared VLS, then any orignal copies of private VLS data *or* hardware-managed data will be kept in the hardware-managed partition. However, the shared data buffer will not cache original copies of data in the shared VLS. This selective caching of 'superfluous' copies is dependent on each data buffer having its own personal DMA engine, and a control register that tracks which VLS (i.e. range of virtual memory) is currently resident in it.

3.6.4 Direct VLS-to-VLS communication

Another extension to VLS for multithreaded applications is to make the VLS of one thread "visible" to another, in that the other thread can read from or write to it. The mechanisms supporting this visibility allow direct core-to-core data transfers without involving the outer levels of the memory system. Figure 6 shows how we do this by expanding the VLS virtual address map to contain a separate address range (VLS 0, ..., VLS N-1) for each participating thread. These ranges are in addition to the original `local VLS` virtual address range, which always maps to the thread's local VLS (meaning one of the new VLS ranges will be a shadow backed by the same physical memory location).

A new set of registers is required to map which virtual address ranges correspond to active VLSs belonging to threads currently running on the processor. Each core has a register for every data buffer in which a VLS can be resident; each register implicitly corresponds to a specific data buffer. The VLS virtual address range to physical data buffer mappings must be kept coherent by the operating system as it migrates threads across cores in the system, or suspends and resumes them. This coherence requirement is similar to the demands of TLB coherence). This mechanism makes use of the virtual address space to abstract away from the programmer the number of physical data buffers in the system and the current placement of threads onto cores and their private data buffers.

A load or store to a remote thread's VLS is recognized by its target virtual address, and is then routed directly to the remote core's cache controller, bypassing the regular cache coherence scheme . The memory access is then handled by the remote cache controller as if it had originated from the local core's running thread. Figure 7 shows this sequence of events. .

The DMA engine can be used for higher performance, for example, reading from the local VLS and writing to a remote VLS to facilitate producer-consumer access patterns without excessive coherence traffic. The local read happens as usual, but the write request is now sent directly to the remote controller, not the outer layers of the memory hierarchy. The DMA engine could also read from a shared memory buffer and write to a remote VLS to allow one thread to prefetch data for another. The same mechanism can also be extended to allow external devices or application accelerators to be closely coupled with a thread. We provide a logically separate network to carry this cross-VLS request traffic, but this network can be multiplexed onto existing memory system physical networks to reduce cost.

When an active thread attempts to send data to a non-resident thread's VLS, the regular TLB and page tables are consulted to find the backing physical copy to which they will direct the updates. As an alternative, the operating system might choose to always gang-schedule all threads in a process onto physical cores, avoiding the need to handle partially resident processes. However, while a program doing partially-resident communication will suffer from additional coherence traffic (relative to one that is gang-scheduled), both cases will still execute correctly. This is

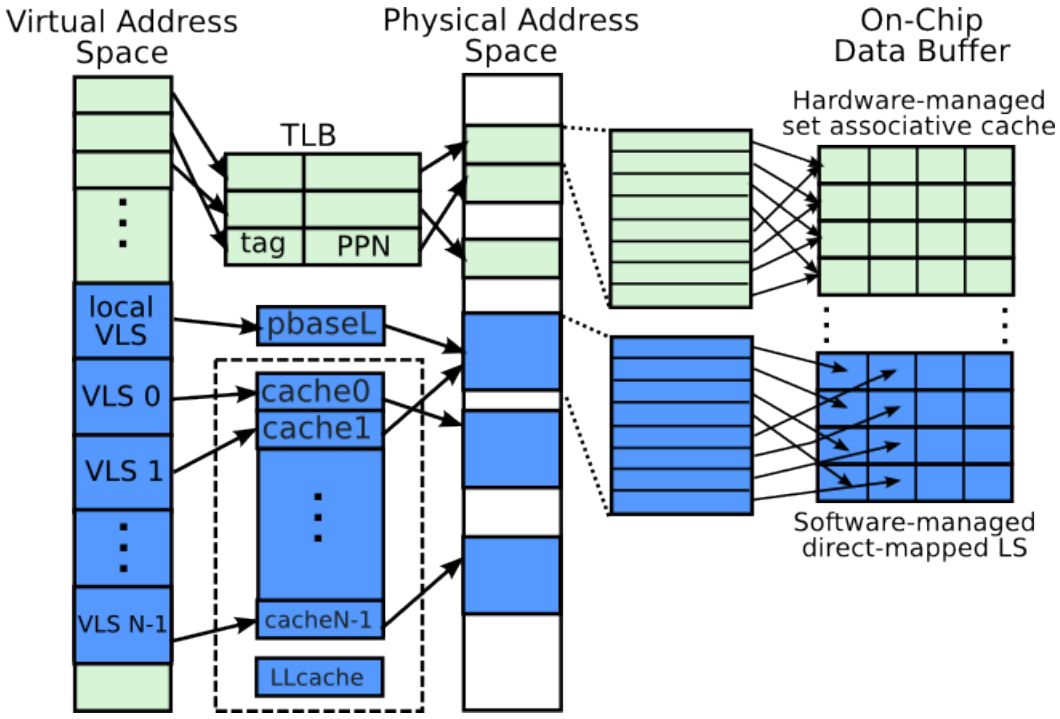


Figure 6: Mapping of multiple, mutually visible virtual local stores from the virtual address space to physical page. Multiple registers are used to track the current mappings of VLSs to data buffers. Changes in mappings must be kept coherent across all cores (indicated by dotted line). In this example the process is running on the core associated with `cache1`, so `local VLS` and `VLS 1` are shadows that map to the same physical memory location.

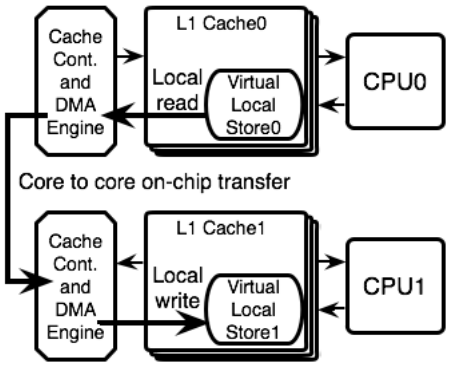


Figure 7: Mechanisms supporting direct VLS-to-VLS communication.

another example of how the way VLS builds on top of a hardware cache mechanisms makes it easier to implement software-managed memory in a general purpose system.

4 Evaluation

In this section, we quantify the benefits of adding VLS to a system. We compare systems with purely hardware-managed caches or with a fixed partition between hardware-managed cache and software-managed local store against the more flexible partitioning possible with VLS. In general-purpose systems, a fixed partition will be sub-optimal globally since not all applications and libraries can be converted to effectively use the specialized resources.

We simulate the performance of multiple versions of a set of microbenchmarks and multiple implementations of more complex speech recognition and image contour detection applications, all running on a simulated machine with virtualized local stores. We find that, in the case of the benchmarks, VLS is able to provide each version with the best possible configuration, while no static configuration is optimal for all benchmarks. In the case of the multi-phase applications, VLS is able to provide each phase with a unique configuration, resulting in "best of both worlds" performance. VLS also significantly reduces the amount of data that must be saved and restored on context switches.

4.1 Microbenchmarks

We select a group of microbenchmarks from the set of multithreaded benchmarks used by Leverich et al. in their study of memory management models [14]. The benchmarks were originally chosen to represent a variety of general-purpose and media applications. The benchmarks are parallelized using POSIX threads and use the associated synchronization mechanisms [15]. Not all benchmarks had source code that was available to us. Table 1 describes some characteristics of these benchmarks. We consider two versions of each microbenchmark: one that assumes hardware-managed local memory, and one that assumes a mix of hardware- and software-managed local memory (in the original study this later version ran on a system with a physically separate local store, in our study it runs on a system with an appropriately configured VLS).

4.2 Multi-phase Applications

As a case study in applying VLS to larger, more complex applications with varying memory usage patterns, we evaluate several implementations of two different multi-phase applications.

4.2.1 Image Contour Detection

The first application is an implementation of the *gPb* approach to image contour detection [5]. Image contour detection is an important part of many computer vision tasks, including image segmentation, object recognition and object

Name	Input set	Parallelism
Merge Sort	2^{19} 32-bit keys (2 MB)	Across sub-arrays, decreasing with time
Bitonic Sort	2^{19} 32-bit keys (2 MB)	Across sub-arrays, no decrease over time
FIR filter	2^{20} 32-bit samples	Across sample strips
Stereo Depth Extraction	3 CIF image pairs	Input frames are divided and statically allocated
MPEG-2 Encoder [24]	10 CIF frames	Macroblock task queue, no dependencies
H.264 Encoder	10 CIF frames (Foreman)	Macroblock task queue, interdependencies

Table 1: Microbenchmark descriptions. See [14] for details

classification. The *pGb* detector [17] is the highest quality image contour detector currently known, as measured by the Berkeley Segmentation Dataset. The detector is composed of many modules that can be grouped into two sub-detectors, one based on local image analysis on multiple scales, and the second on the Normalized Cuts criterion.

The local detector uses brightness, color and texture cues at multiple scales. For each cue, the detector estimates the probability of a contour existing for a given channel, scale, pixel and orientation by measuring the difference between two halves of a scaled disk centered at that pixel and with that orientation. The detector is constructed as a linear combination of the local cues, with weights learned from training on an image database.

The local detector is used to create an affinity matrix whose elements correspond to the similarity between two pixels. This affinity matrix is used in the Normalized Cuts approach [25], which approximates the normalized cuts graph partitioning problem by solving a generalized eigensystem. This approach results in oriented contour signals that are linearly combined with the local cue information based on learned weights in order to form the final *pGb* detector. For more information on the algorithms used in the various detectors, please refer to [5].

Two of the many modules that make up the *gPb* detector take up the vast majority (90%) of computation time. One is the module responsible for computing the local cues, the other for the solution to the generalized normalized cuts eigensystem. Local cue computation involves building two histograms per pixel. The eigensolver used here is based on the Lanczos algorithm using the Cullum-Willoughby test without reorthogonalization, as this was found to provide the best performance on the eigenproblems generated by the Normalized Cuts approach [5]. The Lanczos algorithm requires repeatedly performing a sparse matrix-vector multiply (SpMV) on large matrices; the size of the matrix is the square of the number of pixels in the image.

Converting the local cues calculation and the eigensolver to use software memory management took less than 50 lines of additional code (the entire application had 3953), not counting the previously implemented DMA library. For the local cues calculation, the entirety of both histograms used per pixel fit in the VLS. Placing the histograms there ensured that they were kept in the L1 data buffer throughout the computation. Sections of the image were also software prefetched. For the eigensolver, adding macroscopic software prefetching to some sections of code was trivial, whereas for other pieces of code (such as SpMV) the optimizations required more effort.

VLS allowed us to leave the insignificant portions of the code unmodified, without paying an energy or performance overhead.

We used a 50 x 50 pixel image for this experiment, as larger datasets resulted in extraordinarily lengthy simulation run times. Future work on faster simulators will explore how the performance benefits scale with increasing image size.

4.2.2 Speech Recognition

The second application is a Hidden-Markov-Model (HMM) based inference algorithm that is part of a large-vocabulary continuous-speech-recognition (LVCSR) application [6, 10].

LVCSR applications analyze a set of audio waveforms and attempt to distinguish and interpret the human utterances contained within them. First, a speech feature extractor extracts feature vectors from the input waveforms. Then an inference engine computes the most likely word sequence based on the extracted speech features and a language recognition network. The graph-based recognition network represents a model of human language that is compiled ofine, and trained using statistical learning techniques. The recognition network we use here models a vocabulary of over 60,000 words and consists of millions of states and arcs.

Phase	Name	Description	Data Access
1	Cluster	Observation prob. computation, step 1	Up to 6 MB data read, 800KB written
2	Gaussian	Observation prob. computation, step 2	Up to 800KB read, 40KB written
3	Update	Non-epsilon arc transitions	Small blocks, dependent on graph connectivity
4	Pruning	Pruning states	Small blocks, dependent on graph connectivity
5	Epsilon	Epsilon arc transitions	Small blocks, dependent on graph connectivity

Table 2: LVSCR application phase descriptions. See [6] for details

The algorithm traverses the graph of the network repeatedly, using the Viterbi search algorithm to select the most likely words sequence from out of tens of thousands of alternative interpretations [20]. Because the graph structure of underlying network is based on natural language it is very irregular, and when combined with an input dependent transversal pattern this irregularity results in a highly variable working set of active states over the course of the algorithm runtime. The inference process is divided into a series of five phases, and the algorithm iterates through the sequence of phases repeatedly with one iteration for each input frame. Table 2 lists the characteristics of each phase of the application.

Converting the inference engine to use software memory management took only about 50 lines of additional code (the entire application had 2969), again not counting the previously implemented DMA library. Most of the modifications had to do with software pipelining loops to allow for effective software prefetching. The transition process was eased by the fact that VLS guarantees correct execution even in the event of misconfiguration, and that the conversion could be implemented incrementally. All phases were converted to use software memory management to illustrate the contrast in the effectiveness of the different management types for different memory access patterns.

While the inference process can exploit fine-grained parallelism within each phase [6,21], we have initially chosen to work with sequential phase implementations for simplicity. The presence of multiple, repeated phases, each with different memory access patterns, is sufficient to demonstrate the applicability of VLS. We are working to port a parallel implementation of the LVCSR application application, as well as additional complex applications with diverse memory management needs.

4.3 Simulator

We use Virtutech Simics 3.0 [28] and Wisconsin GEMS [18] to simulate a sixteen core CMP system with a two level on-chip memory hierarchy. Table 3 summarizes the parameters of this system. We choose these specific hardware parameters in an attempt to match the memory hierarchy design space explored by Leverich et al. [14]. Our goal in doing so is to give our conclusions a concrete basis in the context provided by their examination of several chip multiprocessor memory management models and configurations.

While both studies have memory hierarchy parameters that are similar in scope, the actual simulation infrastructures used are quite different. In our case, Simics provides the functional simulation of the processor cores, while GEMS Ruby provides timing and functional models of the cache hierarchy, on-chip network, and off-chip DRAM memory modules. The cores modeled by Simics are simple in-order processors that support the x86-32 instruction set. Simics is a full system simulator, and we run OpenSolaris (microbenchmarks) or Fedora Linux (applications) installations on our simulated machine. We have modified GEMS Ruby to support VLS, and added a hardware prefetch engine (prefetching is turned on only where explicitly noted).

Simics' speed is sufficient to allow us to simulate the microbenchmarks to completion, or to at least one billion instructions per core, whichever comes first. We are able to run the single-threaded LVSCR application to completion, which can take over 50 billion cycles in some instances.

4.4 VLS Implementation

The most pertinent feature of the simulated machine is that each core is associated with a local, private on-chip memory, all of which are backed by a global, shared on-chip memory. We have modified GEMS Ruby's functionality such that each of the local memories can be dynamically partitioned into a VLS and a hardware-managed cache.

Attribute	Setting
CPUs	800MHz, in-order execution, 16 cores for microbenchmarks, 1 for LVSCR
L1 Instruction Caches	16KB, 2-way set associative, 32-byte blocks, 1 port
L1 Data Caches	32 KB, 4-way set associative, 32-byte blocks, 1 port, VLS enabled
L2 Unified Cache	512KB, 16-way set associative, 32-byte blocks, 2 banks, non-inclusive
Coherence Protocol	GEMS MOESI-CMP-Directory-m [18]
Hardware Prefetcher	1 per data cache, tagged, tracks sequential accesses, 8 access history, run-ahead of 4 lines
DMA Engine	1 per core, 32 outstanding accesses
On-chip Network	Hierarchical Crossbar, 2 cycle latency local, 5 cycle latency non-local, plus arbitration
Off-Chip 1GB DRAM	One channel at 3.2 GHz
Virtual Local Stores	One per core, may be 0 to 24 KB in size, incremented by 8 KB (way-based)

Table 3: Simulated system parameters

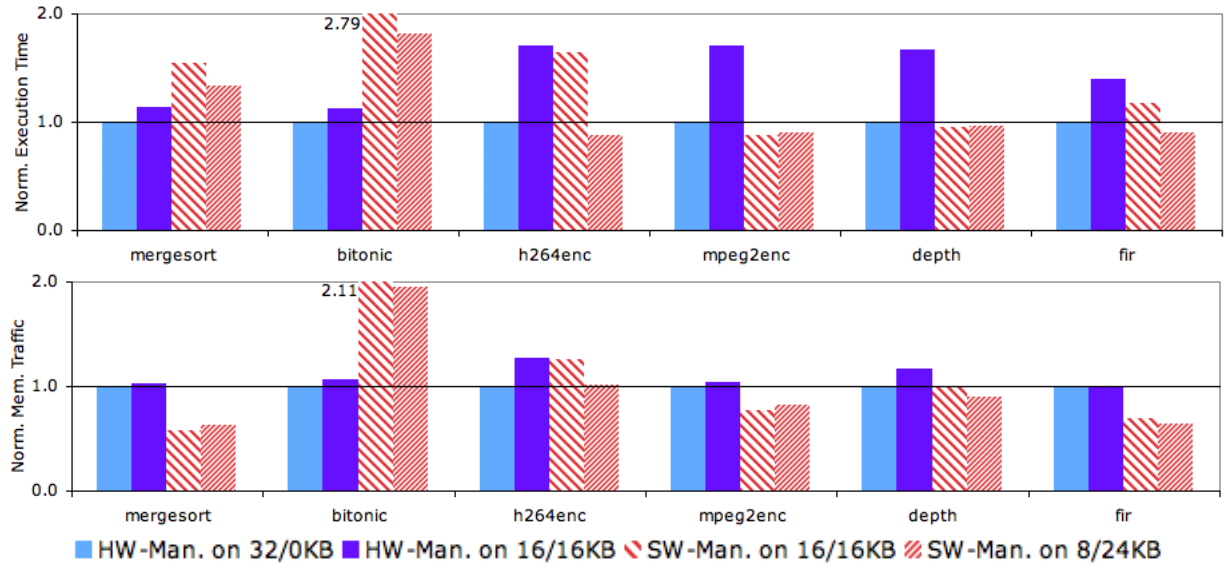


Figure 8: Normalized effect of hardware and software management on performance and aggregate off-chip memory traffic. The y-axis shows the normalized number of cycles or off-chip read and write requests issued by the application, while each bar is a different combination of management style and hardware configuration. The hardware configurations are specified as X/Y, where X is the number of KB allocated to cache and Y is the number of KB allocated to local store.

Adding this capability required modifying the cache replacement protocol and the indexing mechanism for accesses identified as mapping to the VLS. In our experiments partitioning is done on a *per-way* basis. The virtual address space region that is to be treated by the hardware as mapping to the VLS is communicated to the simulator by Simics magic instructions; which are functionally equivalent to writing to the `pbase` and `pbound` control registers.

We created a DMA library backend to support the software management of the virtual local stores in our simulated machine. We took the Smart Memories [16] DMA library already used by the software-managed versions of the microbenchmarks [14] and ported it to generate data movement requests on our Simics-based simulator.

4.5 Microbenchmark Analysis

The microbenchmarks we use have been converted to be able to run using either caches or local stores, but this conversion process was not a simple one [14]. Architects cannot expect all software that will ever run on the system to be rewritten; programmer effort aside, the software-managed memory model is untenable for programs with complex memory access patterns. In the face of application heterogeneity, general-purpose systems without virtualization or reconfiguration capabilities must statically allocate their physical resources between different use cases.

The graphs in Figure 8 expose the various effects that different static partitionings between cache and local store

Phase	Name	HW-managed	Software-managed
1	fileio	0.078	0.079
2	lib	0.005	0.005
3	textons	0.668	0.668
4	localcues	7.581	5.375
5	mcombine	0.008	0.008
6	nmax	0.001	0.001
7	intervene	0.307	0.307
8	eigsolver	3.305	1.381
9	oefilter	0.167	0.167
10	gcombine	0.002	0.002
11	postprocess	0.006	0.006

Table 4: Performance of vision phases under different memory management styles. Performance number units are billions of cycles.

have on each of our microbenchmarks' performance and aggregate off-chip bandwidth, which is a significant contributor to energy used by the memory system. While software management is slightly more effective for certain benchmarks, hardware management is more efficient in some cases. An even split of the underlying hardware between the two use cases is generally detrimental to cache miss rates and local store efficiency, relative to configurations that maximize the resources available to the memory management used by the code. Hardware-managed versions running on limited cache sizes demonstrate the performance penalty that unmodified applications will pay for having a permanently allocated physical local store. Software-managed configurations with limited local store sizes demonstrate the limits placed on software management's effectiveness by having some local memory configured to be hardware-managed cache for the benefit of legacy applications or libraries.

The results presented here correspond with those published in [14], even though the underlying simulation infrastructure and local store implementation are completely different. This independent verification helps to validate both sets of results. A few comments on specific benchmarks: bitonic sort's high traffic and poor performance under the software-managed model is due to the fact that sublists being sorted may already be partially in order, and hence do not actually need to be modified and written back. The hardware-managed systems discover this automatically, whereas the software-managed systems write back all the data regardless. FIR and Mergesort see the most memory traffic improvement under the software-managed model. Mergesort makes significant use of store-only lists, which a write-allocate cache will first refill in the cache before immediately overwriting it. Explicit management of the output lists alleviates this problem, reducing memory traffic. However, the software management requires extra comparisons within the inner loop, resulting in extra instructions and increased runtime. FIR's performance is benefiting particularly from macroscopic software prefetching using DMAs.

These benchmarks represent computational kernels, many of which might be composed together in a real application. Performance suffers when a compromise solution is chosen that statically allocates less on-chip memory to each type of management in order to provide for both types. Given a mixed set of kernels or libraries running on a statically partitioned machine, we must apply the same allocation across all applications (i.e. choose a single bar across all benchmarks in Figure 8), and this inevitably results in suboptimal performance for some programs. In contrast, VLS can dynamically repurpose memory resources on a per routine basis, allowing programmers to always choose the configuration most suited to their kernel's memory access pattern.

4.6 Vision Application Analysis

Based on the runtime breakdown of the different tasks involved in *gPb* image contour detection, we deemed that it was only worthwhile to provide software memory management for two of the many steps (local cue calculation and the eigensolver). VLS made this selective deployment of software memory management as an optimization possible. Using VLS and software memory management resulted in significant speedups for the two tasks we did target, resulting in an overall 32% reduction in runtime. The speedup for local cues is due to a combination of placement of certain

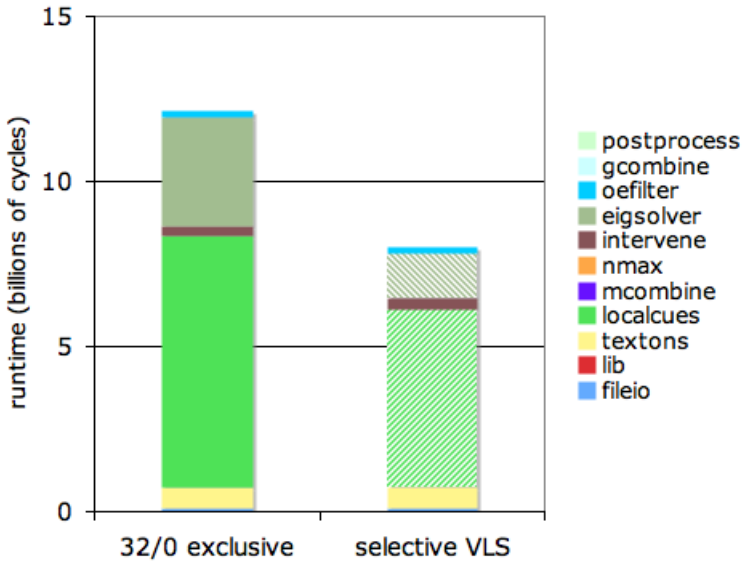


Figure 9: Effect of different memory management schemes on per-phase performance in cycles. The first bar represents hardware management on a full cache system, and the second is a dynamically partitioned system with VLS. Solid segments represent hardware-managed phases, while striped segments are software-managed.

data structures inside the direct-mapped VLS partition, and some software prefetching. The improvement in the eigensolver performance is due primarily to macroscopic software prefetching (since the matrices and vectors involved are generally too large to fit entirely within the local data buffer, and span multiple pages). The results presented in Figure 9 and Table 9 are for a single-threaded version of the code operating on a 50x50 pixel image. A larger image might increase the relative benefit of the software-management, however we have not yet evaluated any due to the performance limitations of our software simulator.

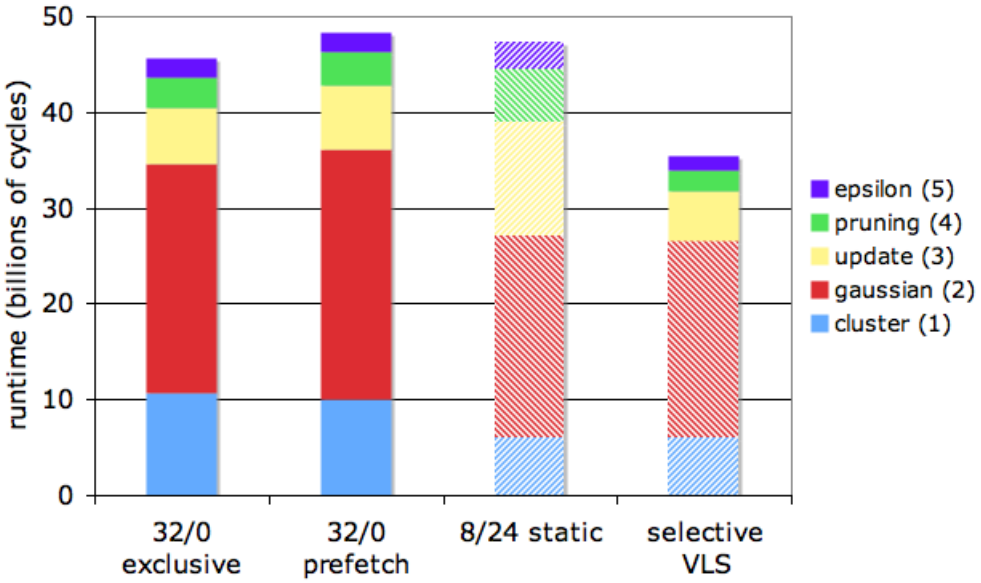
4.7 Speech Application Analysis

Phase	Name	32KB HW-man.	32KB HW-man. w/ prefetch	8KB HW-man. and 24KB SW-man.
1	Cluster	10.73	10.05	5.98
2	Gaussian	23.87	26.08	21.11
3	Update	5.84	6.64	11.86
4	Pruning	3.22	3.46	5.57
5	Epsilon	1.92	2.17	2.87

Table 5: Performance of speech recognition phases under different memory management styles. Performance number units are billions of cycles.

The speech recognition inference engine has different memory access patterns in each of its phases. This diversity is reflected in Figure 10 and Table 10, which show the performance of each of the phases under different memory management models and hardware configurations. The hardware configurations considered are all-phase hardware management on a full cache system, all-phase hardware management on a full cache system with a hardware prefetch engine, all-phase software management on a system with physically partitioned local stores, and both styles of management chosen per-phase on a dynamically partitioned system with VLS. While some phases of the application benefit more from one model or the other, VLS allows us to choose the best model and memory configuration for each phase.

It is important to note that choosing to use software-managed memory for a given routine is a choice akin to choosing to use software pipelining or software prefetching. During development, the programmer must decide whether



[h]

Figure 10: Effect of different memory management schemes on per-phase performance in cycles. Each stacked bar is a use case: hardware management on a full cache system, hardware management on a full cache system with prefetching, software management on a system with physically partitioned local stores, and mixed management on a dynamically partitioned system with VLS. Solid segments represent hardware-managed phases, while striped segments are software-managed.

deploying software memory management for a given piece of code is worthwhile based on their understanding of the memory access pattern and the impact that the optimization is going to have on overall performance. The contribution of VLS is allowing the programmer to deploy such changes incrementally and selectively, as we see in this case study.

Phase 1 streams through over 6 MB of data, performing simple accumulations. Large (19 KB) chunks of data are accessed deterministically, making it easy to use macroscopic prefetching to hide the cost of fetching this data from memory. The effectiveness of the hardware prefetch engine is limited by the fact that it cannot fetch data across page boundaries, and the fact that it often generates extra memory accesses by prefetching past the end of 512B subblocks of data. Software memory management is very effective and easy to implement for this phase. However, while macroscopic prefetching does limit the latency cost of the many memory accesses, it cannot address the sheer volume of data required by this phase. The traffic caused by such large, repeated streaming reads dominates the off-chip traffic for the entire application.

Phase 2 processes only smaller (1 KB) chunks of data at a time, making it harder to amortize the latency any better than the hardware prefetch engine can do automatically. Software and hardware management are about equally effective in terms of performance for this phase, although the software model generates fewer off-chip memory accesses since it never has to speculate. This phase is primarily computation-bound in our current implementation.

Phases 3, 4, and 5 are all heavily dependent on the graph structure of the underlying language model. Small, non-consecutive chunks of arc and state data are accessed and modified depending on which states are active and depending on the graph's connectivity. This makes it hard to effectively prefetch any large amount of data. Furthermore, there is a degree of reuse within and between these phases, as certain state and arc information may be read or written repeatedly. The hardware management mechanisms fortuitously discover this resuse, whereas our software management must pessimistically write back and refetch these small structures repeatedly.

Overall, we see a mix of different behaviors within a single application. The structure of the algorithm makes both management styles difficult or inefficient in some phases, but beneficial in others. Using the lightweight reconfigurability of VLS, the programmer can pick a local memory configuration and management type on a per phase or per routine basis. This flexibility allows us to achieve the best possible overall performance.

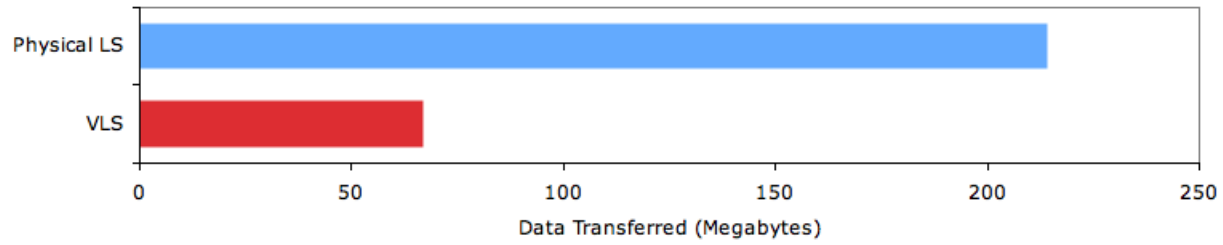


Figure 11: Effect of backing the VLS data in physical memory on context switch overhead for the speech inference engine. Both bars assume 24KB per-core of local store. The top bar assumes a separate physical local store that is virtually indexed. The bottom bar assumes a VLS with physically tagged data. Overhead is measured in terms of megabytes of data which are written back and restored due to context switches over the course of the application's runtime. A Linux 2.6.15 kernel with default settings is used.

4.8 Context Switch Overhead

VLS provides a way to seamlessly integrate local store semantics with a multiprogrammed operating system, but it should also provide improved efficiency over a physically partitioned local store in this environment. We quantify this improvement by comparing the amount of data that is added to the user process state by a physical local store (which must be saved and restored on context switches) with the amount of stored data that is evicted and refetched on-demand in a VLS system. Figure 11 plots this comparison for our LVSCR application on a machine with 24 KB local stores. On every context switch, the machine with the physical local store must save and restore the entire local store. In contrast, the machine with VLS only evicts stored data if there are conflicts with the new program's memory accesses, and only restores them on demand. In this example the only other code that runs are kernel routines, system daemons, and another user program running in an idle loop. If a competing program was able to use up the entire local memory during its time on the processor then the performance of both systems would be near equivalent. The reductions in data movement enabled by VLS, combined with its single tag lookup per access and single dual-use datapath, suggest that it may be more energy efficient than hybrid systems with split physical caches and local stores – this energy tradeoff is the subject of our future work.

5 Related Work

Albonesi [1] proposes a hardware-managed cache organization that allows for certain ways of the cache to be disabled for certain applications or phases of application execution. Ranganathan et al. [22] build on the way-based partitioning design by allowing different ways to be repurposed for a variety of uses (e.g. instruction reuse), rather than just being turned off. VLS makes use of these reconfigurable components.

A fundamental component of the Smart Memories architecture [16] is a reconfigurable memory substrate made up of 'mats' of small SRAM blocks and configurable logic. A set of mats can be configured to implement a wide variety of cache organizations, as well as local stores or vector register files. This reconfigurability can incur significant performance overheads, and the burden placed on the programmer is quite substantial as they must now program the wires and memory hierarchy as well as the processor. VLS is lightweight in that it provides support only for the local store use case, but does so without requiring the programming model to manage hardware behavior explicitly.

The TRIPS architecture as presented by Sankaralingam [23] allows memory tiles embedded throughout the architecture to behave as NUCA L2 cache banks, local stores, synchronization buffers, or stream register files. While different subsets of a TRIPS architecture may be in different morphs simultaneously, there is no clear way for a single process to simultaneously make use of the memory structures or policies associated with two or more morphs. Again, VLS serves as a lightweight solution compatible with standard architectures that provides only for the common local store use case.

Sasanka et al. [24] present ALP, an architecture that makes use of a DLP technique called *SIMD vectors and streams* on top an incrementally modified superscalar CMP/SMT architecture. Part of their L1 data cache can be reconfigured to provide a SIMD vector register file when needed. Only a single cache way is reconfigured into SVRs,

and this is done using the technique described by [22]. VLS uses address space mechanisms rather than special instructions to access the repurposed underlying hardware, enabling unmodified library code to operate on stored data.

The Texas Instruments TMS320C62xx series of digital signal processors have the capability to divide the L2 cache portion of their memory hierarchy into a hardware-managed cache and a software-managed RAM [26]. This is done on a per-way basis, similar to the technique used in [22]. Intel's Xscale processor architecture [11] has a virtually-tagged data cache that provides both per-cache-line locking of external memory addresses and allows certain ways of the data cache to be used as a local store. These hardware mechanisms are the most similar to the ones we propose, but no mechanisms are provided in support of multiprogramming and they have significant repartitioning overhead (for example, the Xscale requires a special instruction is issued per line locked in the cache).

Wen et al. [29] have developed a system design targeting applications with both regular and irregular streaming characteristics. They evaluate a variety of designs for the on-chip memory hierarchy, including physically partitioned [14] or hybrid systems [12], and discover that combining a streaming register file and a cache into a unified Syncretic Adaptive Memory module yields the best performance. Similarly, Gummaraju et al. [8,9] have proposed using portions of the L2 cache as a local memory with per-line locking flags, as well as additional hardware modifications in support of a stream programming model. Both projects focus on a specific type of streaming workload, and so naturally only addresses code compiled by a specialized compiler or operating in a special runtime environment. However, their key insights mirror our own, and SAM, Streamware, and VLS can be seen as addressing similar pressing issues for programmers attempting to write highly efficient code.

Leverich et al. [14] provide a recent comparison study between pure hardware-managed caches and software-managed local stores in a chip multiprocessor context. They use a common set of architectural assumptions to evaluate two representative architectures under a set of hand-tuned benchmarks. They find that the stream programming model improves performance when used on top of either memory model, and that both organizations are near equally scalable. VLS gives the programmer flexibility to choose between their design points at a very fine granularity, and supports adoption of the stream programming model by interested programmers.

Software prefetching has been proposed as a way to mitigate the reactive disadvantage of hardware management mechanisms [4, 27]. However, correctly utilizing software prefetch can prove difficult for programmers and compilers due to the fact that its effectiveness depends on the specific timing and replacement policies of the underlying hardware. VLS provides a stable target for software prefetch mechanisms. Software management of a special repurposed partition has also been proposed for the instruction cache [13].

6 Future Work

Virtual local stores can be implemented at multiple levels of the cache hierarchy, and we are working to rewrite applications to take advantage of shared virtual local stores maintained in the outer levels of the cache hierarchy. The additional associativity and size of the outer caches may allow for better management of temporal locality in shared memory workloads. We are also investigating multi-threaded applications that can make use of fine-grained data movement directly between VLSs resident on separate cores, and will investigate the utility of providing additional hardware support for such movement. Further studies will also incorporate analyses of the energy saving provided by on-demand software memory management.

Another area of future research is integrating VLS into software productivity frameworks. Such frameworks might include performance-tuned libraries, code-generating autotuners, or scripting language constructs that JIT into highly efficient parallel code. All these frameworks mesh well with the VLS paradigm, in that they are intended to allow programmers to selectively optimize their code in domain specific ways. It remains to be seen which frameworks will be able to make efficient use of the software memory management capabilities provided by VLS.

There are other mechanisms that could be 'virtualized' by repurposing on-chip data buffers in order to allocate specialized resources only when they are needed. One example is virtual transaction buffers that provide hardware support for transactional memory when it is useful, but serve as normal caches otherwise. Another example might be ring buffers used for communicating messages in a partitioned exokernel operating system. Mechanisms of interest to us are ones that can make use of the implicit mapping between virtual address space regions and some underlying specialized semantics. Using the virtual address space is a composable, flexible way to provide programmers with on-demand special purpose memory structures without damaging the performance of other programs that will be running

on the system. For now, it is unclear how many additional specialized use cases (if any) are worth supporting in the underlying hardware.

7 Conclusion

The power wall and the memory wall are increasing interest in software-managed local stores because expert programmers writing code for all types of systems are seeking to reduce power and improve memory performance. Hardware-managed caches are still needed for programs that do not lend themselves to such optimizations, as well as for programs that either are run too infrequently to merit heavy tuning or are written by programmers without the skills needed to take advantage of local stores. VLS gives expert programmers concerned with efficiency the software management tools they need without inflicting a performance penalty on all noncompliant code.

VLS is a set of architectural mechanisms that provides the advantages of software-managed memory without changing the ISA or substantially increasing the state of the user processes. The composable nature of VLS allows procedures written using software memory management optimizations and procedures that assume hardware management to work synergistically together in a single program, which is likely a requirement for the success of local stores in non-embedded applications. Building on underlying hardware management mechanisms ensures that execution will always be correct, even in the face of programmer error. At the same time, the local store semantics provided by VLS are easier for programmers to reason about than software prefetching into a hardware-managed cache.

With VLS, programmers do not have to pick a ‘mode’ at boot time or as an application begins to execute and then live with that setting throughout the entirety of the application’s run time. While some applications could be wholly converted to software management, partial conversions that only optimize critical sections are supported effectively with VLS. Virtual local stores only exist when the programmer can make use of them, and are seamlessly absorbed into the hardware-managed memory hierarchy otherwise. VLS provides all the benets of software-managed memory hierarchies within the context of a full-featured mainstream computing environment.

References

[1] D. H. Albonesi. Selective cache ways: on-demand cache resource allocation. In *MICRO 32: Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture*, pages 248–259, Washington, DC, USA, 1999. IEEE Computer Society.

[2] B. Batson and T. N. Vijaykumar. Reactive-associative caches. In *PACT '01: Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, pages 49–60, Washington, DC, USA, 2001. IEEE Computer Society.

[3] D. A. Calahan. Performance evaluation of static and dynamic memory systems on the Cray-2. In *ICS '88: Proceedings of the 2nd international conference on Supercomputing*, pages 519–524, New York, NY, USA, 1988. ACM.

[4] D. Callahan, K. Kennedy, and A. Porterfield. Software prefetching. In *ASPLOS-IV: Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, pages 40–52, New York, NY, USA, 1991. ACM.

[5] B. Catanazaro, B.-Y. Su, N. Sundaram, Y. Lee, M. Murphy, and K. Keutzer. Efficient, high-quality image contour detection. In *IEEE International Conference on Computer Vision*, 2009.

[6] J. Chong, Y. Yi, N. R. Satish, A. Faria, and K. Keutzer. Data- parallel large vocabulary continuous speech recognition on graphics processors. In *Intl. Workshop on Emerging Applications and Manycore Architectures*, 2008.

[7] L. T. Clark and et al. An embedded 32-b microprocessor core for low-power and high-performance applications. *IEEE JSSC*, 36(11):1599–1608, November 2001.

[8] J. Gummaraju, J. Coburn, Y. Turner, and M. Rosenblum. Streamware: programming general-purpose multicore processors using streams. In *ASPLOS XIII: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, pages 297–307, New York, NY, USA, 2008. ACM.

[9] J. Gummaraju, M. Erez, J. Coburn, M. Rosenblum, and W. J. Dally. Architectural support for the stream execution model on general-purpose processors. In *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, pages 3–12, Washington, DC, USA, 2007. IEEE Computer Society.

[10] X. Huang, A. Acero, and H.-W. Hon. *Spoken Language Processing: A Guide to Theory, Algorithm and System Development*. Prentice Hall, 2001.

[11] Intel. 3rd Generation Intel XScale(R) Microarchitecture Developer's Manual. http://www.intel.com/design/intelxscale/, May 2007.

[12] N. Jayasena, M. Erez, J. Ahn, and W. Dally. Stream register files with indexed access. In *Tenth International Symposium on High Performance Computer Architecture (HPCA-2004.*, 2004.

[13] T. M. Jones, S. Bartolini, B. D. Bus, J. Cavazos, and M. F. P. O'Boyle. Instruction cache energy saving through compiler way-placement. In *DATE '08: Proceedings of the conference on Design, automation and test in Europe*, pages 1196–1201, New York, NY, USA, 2008. ACM.

[14] J. Leverich, H. Arakida, A. Solomatnikov, A. Firoozshahian, M. Horowitz, and C. Kozyrakis. Comparing memory systems for chip multiprocessors. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, pages 358–368, New York, NY, USA, 2007. ACM.

[15] B. Lewis and D. J. Berg. *Multithreaded Programming with Pthreads*. Prentice Hall, 1998.

[16] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. J. Dally, and M. Horowitz. Smart memories: a modular reconfigurable architecture. In *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*, pages 161–171, New York, NY, USA, 2000. ACM.

[17] M. Maire, P. Arbelaez, C. Fowlkes, and J. Malik. Using contours to detect and localize junctions in natural images. In *Computer Vision and Pattern Recognition*, pages 1–8, June 2008.

[18] M. M. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, , and D. A. Wood. Multifacet's general execution-driven multiprocessor simulator (gems) toolset. *Computer Architecture News (CAN)*, September 2005.

[19] J. Montanaro, R. T. Witek, K. Anne, A. J. Black, E. M. Cooper, D. W. Dobberpuhl, P. M. Donahue, J. Eno, A. Fatell, G. W. Hoeppner, D. Kruckmeyer, T. H. Lee, P. Lin, L. Madden, D. Murray, M. Pearce, S. Santhanam, K. J. Snyder, R. Stephany, and S. C. Thierauf. A 160MHz 32b 0.5W CMOS RISC microprocessor. In *Proc. International Solid-State Circuits Conference, Slide Supplement*, February 1996.

[20] H. Ney and S. Ortmanns. Dynamic programming search for continuous speech recognition. *IEEE Signal Processing Magazine*, 16:64–83, 1999.

[21] S. Phillips and A. Rogers. Parallel speech recognition. *Int. J. Parallel Program.*, 27(4):257–288, 1999.

[22] P. Ranganathan, S. Adve, and N. P. Jouppi. Reconfigurable caches and their application to media processing. In *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*, pages 214–224, New York, NY, USA, 2000. ACM.

[23] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, N. Ranganathan, D. Burger, S. W. Keckler, R. G. McDonald, and C. R. Moore. Trips: A polymorphous architecture for exploiting ilp, tlp, and dlp. *ACM Transactions on Architecture and Code Optimization(TACO)*, 1(1):62–93, March 2004.

[24] R. Sasanka, M.-L. Li, S. V. Adve, Y.-K. Chen, and E. Debes. Alp: Efficient support for all levels of parallelism for complex media applications. *ACM Transactions on Architecture and Code Optimization*, 4(1):3, 2007.

[25] J. Shi and J. Malik. Normalized cuts and image segmentation. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 22(8):888–905, August 2000.

[26] Texas Instruments. TMS320C6202/C6211 Peripherals Addendum to the TMS320C6201/C6701 Peripherals Reference Guide (SPRU290), August 1998.

[27] S. P. Vanderwiel and D. J. Lilja. Data prefetch mechanisms. *ACM Computing Surveys*, 32(2):174–199, 2000.

[28] Virtutech. Simics ISA Simulator. www.simics.net, 2008.

[29] M. Wen, N. Wu, C. Zhang, Q. Yang, J. Ren, Y. He, W. Wu, J. Chai, M. Guan, and C. Xun. On-chip memory system optimization design for the ft64 scientific stream accelerator. *IEEE Micro*, 28(4):51–70, 2008.

[30] S. Williams, J. Carter, L. Oliker, J. Shalf, and K. Yelick. Lattice boltzmann simulation optimization on leading multicore platforms. In *International Parallel and Distributed Processing Symposium (IPDPS)*, April 2008.

[31] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick. Scientific computing kernels on the cell processor. *International Journal of Parallel Programming (IJPP)*, June 2007.