Sidestep: Co-Designed Shiftable Memory & Software

T. Kelly, H. Kuno, M.D. Pickett, H. Boehm, A. Davis, W. Golab, G. Graefe, S. Harizopoulos, P. Joisha, A. Karp, N. Muralimanohar, F. Perner, G. Medeiros-Ribeiro, G. Seroussi, A. Simitsis, R. Tarjan, R.S. Williams

HP Laboratories
HPL-2012-235

Keyword(s):

memory

Abstract:

We have designed computer memories that can shift a long user-specified contiguous region a short, fixed distance in constant time. Such memories make possible complementary co-designed software that not only improves performance but also simplifies and unifies solutions to fundamental computing problems including sorting, searching, and data management.

External Posting Date: November 21, 2012 [Fulltext] Approved for External Publication
Internal Posting Date: November 21, 2012 [Fulltext]

Sidestep: Co-Designed Shiftable Memory & Software

T. Kelly, H. Kuno, M.D. Pickett, H. Boehm, A. Davis, W. Golab*, G. Graefe, S. Harizopoulos†, P. Joisha,
A. Karp, N. Muralimanohar, F. Perner, G. Medeiros-Ribeiro, G. Seroussi, A. Simitsis, R. Tarjan, R.S. Williams

Hewlett-Packard Laboratories, Palo Alto, California
*Department of Electrical and Computer Engineering, University of Waterloo †Nou Data Corporation

{terence.p.kelly, harumi.kuno, matthew.pickett}@hp.com

Abstract

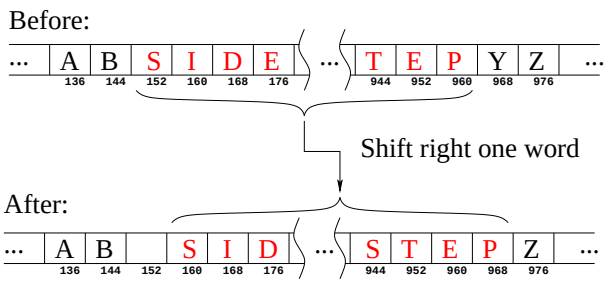
We have designed computer memories that can shift a long user-specified contiguous region a short, fixed distance in constant *time. Such memories make possible complementary co-designed software that not only improves performance but also simplifies and unifies solutions to fundamental computing problems including sorting, searching, and data management.*

1 Introduction

Computer hardware and software have adapted to one another since the inception of the computer industry. Sometimes adaptation resembles a tennis match in which software responds to changed hardware or vice versa; each strives to improve assuming that the other remains fixed. One advantage of such independent evolution is that it respects a clean separation of specialized responsibilities. Unfortunately, independent evolution drives overall systems toward *local* optima rather than the global optimum. Major advances often require coordinated, simultaneous changes that create value collectively even if they make no sense individually—a Model T can't burn hay and a horse can't drink petrol.

This paper considers discontinuous, simultaneous changes to both software and hardware. Memory is a promising area to consider for both technical and business reasons. Off-socket memory bandwidth struggles to keep pace as socket core counts increase. There is pressure to consider an alternative model where certain operations can be supported by the memory subsystem. Memory vendors appear to be open to innovation, covering more niche market segments (e.g., DDRx, GDDR, and RLDRAM). HP owns the platform but not the CPU, so an increased role for memory has strategic appeal. Finally, the disruptive arrival of non-volatile RAM may create openings for both memory component and memory hierarchy innovations. Our specific goal is to enable improved software via easily programmable, massively parallel logic in memory. We want to address general and fundamental computing problems, not just offbeat applications. We want to enable full realization of the performance benefits of multicore concurrency by pairing it with complementary parallelism in main memory. We see a way to exploit HP's advantages—particularly our end-to-end design competencies and proprietary nanoscale devices—to differentiate HP solutions.

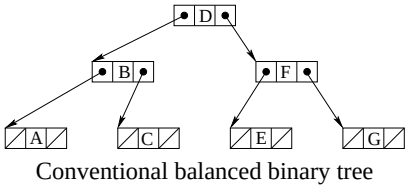
This paper describes "Sidestep," a computer memory that supports an internal shift operation in addition to the usual read/load and write/store operations. A shift displaces the contents of a long (e.g., multiple Mbyte) user-specified contiguous region of memory a short, fixed distance in either direction *in constant time*. The shift occurs *entirely within memory* and the time required is *not* proportional to the amount of data shifted. The distance of a Sidestep shift is fixed; a natural choice for the shift distance is the machine's word size, because software data structures are often multiples of this size. In the example at right, the contents of memory in byte addresses 152–967 are shifted one 64-bit word toward higher addresses in constant time. The remainder of this section explains the potential value of Sidestep by way of the co-designed software that it enables. Section 2 describes fast, compact, efficient Sidestep circuits based on both DRAM and SRAM and explains how emerging memory devices (e.g., memristor) can further improve these circuits.



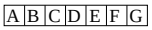
1.1 Sorting and Sorted Sets

How does Sidestep change software? Consider the most fundamental computing problem: sorting. The natural Sidestep sorting algorithm is *not* efficient without Sidestep: Insert items into a sorted array by first finding the proper location via binary search, then shifting to make room, and finally storing the item in the gap thus created. The worst-case time requirement is $O(N \log N)$. Uniformly distributed input allows us to employ interpolation search instead of binary search, which yields a practical $O(N \log \log N)$ sorting algorithm.

The contrast between conventional and Sidestep-aware software is particularly pronounced in the case of *sorted containers*, which maintain items in sorted order under dynamic insertions and deletions. Sorted containers are widely used because they support range queries, making them ideal for in-memory indexing. Today's implementations (e.g., the C++ Standard Library's `map`) use balanced binary trees, but Sidestep allows us to use much more compact and efficient arrays. We maintain stored items in a sorted array using binary or interpolation search for lookups and constant-time shift for insertions and deletions. The figure at right shows two sorted containers filled with items "A" through "G". Sidestep's array is 2–4× smaller than a balanced tree holding the same items, depending on the details, because an array avoids the overheads of child pointers and per-node allocation. Array search is faster than tree search on today's Intel- and AMD-based servers: Binary search on an array is up to 5–6× faster than balanced tree lookup; interpolation search widens the gap to 15–20×. Arrays beat linked data structures by roughly *two orders of magnitude* on in-order traversal because sequential accesses are much faster than pointer chasing. Given the overwhelming advantages of arrays, why are trees used for sorted containers? Because array insertions/deletions on conventional memory require $O(N)$ time [3, pp. 82 and 426], [5, pp. 93, 95, and 502]. Sidestep's constant-time shift allows insertion in $O(1)$ time following a logarithmic-time lookup.



Conventional balanced binary tree



Sidestep array

It's worth noting that Sidestep not only simplifies but also *unifies* solutions to problems that are currently addressed by disparate mechanisms. Currently programmers use Quicksort for sorting, balanced trees for dynamic sorted sets, and heaps for priority queues. Sidestep makes it practical to use a single very simple data structure—a sorted array—for all three purposes.

Closely related to sorting is the problem of efficiently *merging* two adjacent sorted sub-arrays into a single sorted array. With Sidestep, such merging requires no more than a constant amount of extra memory, entails computation proportional to the length of the final sorted array, and preserves the relative order of items in the input sub-arrays. Sidestep-based merging therefore leads directly to a stable, in-place, and optimal comparison sort algorithm that is moreover very simple—a highly desirable and heretofore elusive combination of virtues [2].

1.2 Databases

Sidestep's benefits for sorting and searching are not limited to in-memory data but extend to databases on secondary storage. Data-management software focuses on moving data, both between memory and storage and also within memory. Data structures are designed to speed searches while minimizing the cost of accesses and movement, striking a balance between the cost of moving data to improve its organization versus the proposed move's promise to improve search/access performance. Sidestep changes the equation by asymptotically reducing the cost of an in-memory shift, which opens new possibilities for improving both conventional and emerging database systems. We have developed methods that use Sidestep for sorting large volumes of data, for database index optimization, and for efficiently updating read-optimized secondary storage back-ends (called read-optimized stores) such as those used by Vertica and ArcSight.

1.2.1 In-place Updates in Read-Optimized Stores

Some database systems (such as column-store databases like Vertica) use read-optimized stores in order to improve database performance. Read-optimized stores, which are built for high-performance querying, typically pack and/or compress attribute values of a single column together. This increases the amount of data that can be retrieved with a single disk access. However, this compressed format is not conducive to in-place updates (inserts, deletes or modifications).

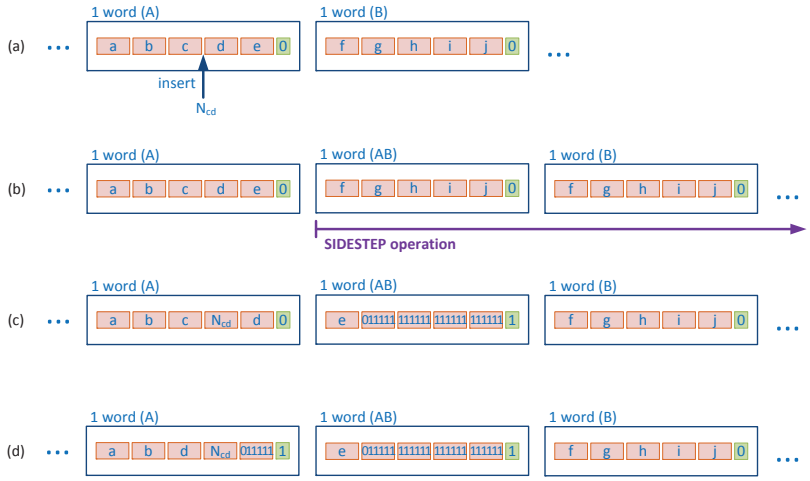


Figure 1: In-place insert/delete in read-optimized stores: the word-aligned case

To facilitate updates, such systems traditionally maintain, and when necessary periodically merge, two separate storage areas: one where all updates are recorded, and one for the read-only data. Operations (updates or queries) may need to process two different storage areas before computing the result. Separating read-only data from updated data may also affect the freshness of query results and performance, depending on how efficiently the results from processing these individual storage area can be merged together.

By leveraging Sidestep, we allow in-place updates on column-store data stored in contiguous main-memory regions. For this, we propose a these storage method for attribute values of a single column. Depending on the nature of the stored values (word-aligned or not), a 1-bit flag/control bit is added to the program data structure for either each value or each memory word, with this control bit being used to indicate one of two modes of storage: (i.) unmodified (values are stored in the same way as in the read-only storage area) and (ii.) modified (a self-navigating storage region with a different storage format than unmodified values, which typically is the result of an update operation that may also involve a Sidestep operation).

To give an example, assume the case of word-aligned single-attribute entries (compressed/non-compressed). In-place updates may be realized as follows (see also Figure 1). Each word contains a control bit (last bit), which is zero (0) if the word has not been modified and one (1) otherwise. For inserting a new value N_{cd} between values c and d in A word (step (a)), we use Sidestep to empty a memory word, say AB, by shifting forward subsequent words (step (b)). It is not important what values are contained in AB at this point. Next, we re-write word (A) to shift the fields over to the right of the insertion point, insert N_{cd}, and move the last value of word A (i.e., e) to the next word, AB. The rest of AB is filled with dummy values starting with 0 and continuing with 1 until AB becomes full. For modified words (whose control bit equals 1) we start reading from the end and continue toward the start until we find the first 0. After this zero there is a valid value; i.e., e. The space captured by dummy values can be exploited for future inserts.

Then, if we want to delete a value, say c (step (d)), we shift forward all remaining values in the word and fill the space first with a zero bit and then with ones. We also denote the word as modified; i.e., the control bit is set to 1. The case of updates follows from insert and delete. Finally, we use a "compaction" process that runs periodically over the fragments of contiguous storage (e.g., pages, extends) and removes all "holes" from memory.

1.2.2 Search and Inserts

Sidestep enables a constant-time shift of data in memory. By thus supporting efficient insertion operations on arrays, it reduces a key cost of constructing and maintaining data in the form of large arrays. There are a number of advantages to being able to store data in a single very large sorted array, as opposed to in a number of individual sorted sub-arrays or other organizations. First, it provides a single object to be searched and managed, thereby

simplifying the task and reducing the cost of organization. Second, increasing the amount of data that can be organized in a single array improves the scalability of techniques that employ arrays.

However, Sidestep does not directly speed search operations. Unless data values are so uniformly distributed that interpolation search can be used (e.g., by padding with a judicious number of "ghost records" added to aid interpolation), Sidestep requires logarithmic time to find a data element in a very large array or to find an appropriate insertion point where a new element may be added. Sidestep may also be subject to certain physical limitations, e.g., on the amount of data that can be shifted efficiently in a single operation (§ 2).

B-tree structures and Sidestep complement each other. By combining the components of Sidestep, library sort, interpolation search, and ghost records, we can increase the number of elements that can be efficiently stored and managed in an array. Sidestep thus enables B-trees to have larger nodes, thereby facilitating shallower trees. In turn, B-tree structures enable robust, efficient search of the data, regardless of whether data is uniformly distributed and accommodating limitations on the amount of data that Sidestep can shift at a time. We merely require that Sidestep be able to perform shifts on regions as large as a B-tree node.

1.3 One-Step Wait-Free Synchronization

Another potential application of Sidestep pertains to multi-threaded software—an area of growing importance in the wake of the multi-core revolution. Harnessing parallelism in multi-core architectures necessitates synchronization among threads that access shared resources, which on today's hardware incurs significant overhead in terms of both computation and engineering effort. Conventional databases, for example, spend more time isolating parallel transactions than searching indexes and retrieving data records [6]. To make matters worse, multi-threaded programs are prone to subtle concurrency bugs that elude even the most skilled programmers, sometimes with disastrous consequences [4]. Decades of research have linked these fundamental problems with the legacy interface between hardware and software, namely the machine instruction set, which forces software to deal with synchronization using minimal hardware support. Sidestep advances the state of the art by enabling powerful software-directed memory shifts, paving the way for robust hardware-accelerated synchronization. Below we sketch how an extension of Sidestep's core functionality can facilitate this feature.

Non-blocking (e.g., "lock-free" and "wait-free") techniques are particularly desirable because they ensure progress even under adverse conditions, such as when a thread halts or pauses while applying an operation to a shared object. However, their added complexity makes them less popular in practice than simple but less robust lock-based mechanisms. In contrast, with Sidestep, the processor can naturally support atomic instructions that combine a fetch or store with a shift, enabling operations on shared stacks in only one step. For example, to push an element X onto a stack represented as an array with elements (Y, Z), a thread would execute an atomic "shift-and-store" that moves (Y, Z) to the right and inserts X in the space created. Similarly, to pop the stack, a thread would execute an atomic "fetch-and-shift" that returns X and then shifts (Y, Z) to the left, overwriting X with Y. (A sentinel element can be used to handle operations on an empty stack easily.) This type of shared stack has all the advantages of a "wait-free" shared object, which guarantees progress to each thread even in the face of crash failures, but without the overhead of complex "helping mechanisms" found in traditional wait-free constructions (e.g., [1]). Similar but slightly more elaborate techniques can be applied to construct FIFO queues, which can be used in high-performance server systems to distribute work to a pool of executor threads. Reducing the overhead of synchronization in such systems helps meet stringent latency and throughput requirements, delivering greater value to HP's customers.

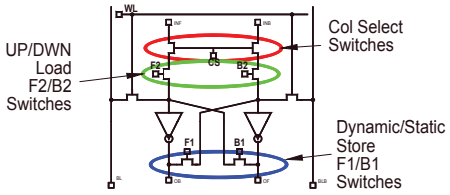
2 Sidestep Hardware Implementation

This section describes two different practical approaches to implementing Sidestep. The first provides cell-level shiftability in a modified SRAM cell (§ 2.1); the second provides block-level shiftability in DRAM sense amplifiers (§ 2.2). We explain how emerging memory technologies including non-volatile RAM can enhance both our SRAM and DRAM designs (§ 2.3). Finally, we describe how Sidestep memory may be integrated into a computer architecture and accessed by software (§ 2.4). The SRAM approach detailed here is logic CMOS compatible and could be realized in a standard CMOS foundry. In contrast, realization of the DRAM approach would require the partnership of a memory vendor since DRAM manufacturers have very specialized and closed processes.

2.1 Cell-Level SRAM Implementation

A shift performed with a CPU connected to conventional main memory will have a performance limited by the read and write properties of the memory. Specifically, a word shift operation will require each word in a selected block to be sequentially read into the processor from one memory location, and to be written by the processor into an adjacent memory location, with a performance proportional to length of the region that is shifted, and proportional to the processor-to-memory access time. To perform such shifts in constant time and without unnecessary transit to/from the processor, Sidestep employs novel components and circuits.

Of the many different types of memory blocks available for the Sidestep memory, a modified SRAM may be used to significantly improve the performance at a modest layout area cost. The basic properties of an SRAM include memory storage without regular refresh operations, fast read and write operations, and two-state storage per unit cell. The modified SRAM starts with the basic six transistor (CMOS) unit cell and adds three groups of switch transistors to momentarily convert the basic SRAM memory into a shift register to shift a selected group of words one position up or down in the memory block and then switch back to an SRAM mode to insert a word at the beginning of the group of words shifted to complete the Sidestep memory operation. The Sidestep shift is performed within a fixed time independent of the length of the shift and the shift operations are performed without SRAM read and re-write operations.



The modified SRAM unit cell illustrated at right takes advantage of the fact that it is physically adjacent to neighboring memory cells that are above and below, and that each SRAM unit cell stores two states (bit and bit-bar). Switches are added to each memory cell to select a group of words within a large memory block, and to convert the selected memory cells into a load/store shift register along the selected columns, and shift all the selected words up or down the selected columns, and then complete the operation with an SRAM write into a selected word location. Four novel concepts make the Word Shift SRAM (WSSRAM) possible: 1) the SRAM cell may switched from a static storage mode to a temporary dynamic storage mode, 2) in the dynamic storage mode, the circuit holding the state of the bits may be configured into an up/down shift register, 3) one or a group of words may be selected within long row lines to perform simultaneous word shifts along selected columns, and 4) long columns of words are folded in a serpentine manner to efficiently utilize the row and column control circuits in high density memory blocks.

Two storage mode switches (F1 and B1) are inserted in the cross coupled feedback loop of the six transistor SRAM unit cell to isolate the two states stored in the SRAM cell by switching the operation of the memory cell from a static storage mode to a dynamic storage mode. The states are held by charge stored in the gates of the SRAM inverter circuits. One of two shift register mode switches (F2 or B2) is used to connect the isolated dynamic store memory circuits into either an up or a down shift register. Note that, in normal word shift operations, adjacent serpentine columns of words shift in opposite directions. Finally, two word select switches are used to select columns of words for the shift operations.

Figure 2 is an example of how a block of four bits can be shifted in a 1×6 SRAM. Initially, the SRAM is in the storage mode with both F1 and B1 closed. An “insert” address and a “delete” address are applied to a modified row decoder that will determine the length and direction of the shift operation; the “insert” address is where a gap will be created by a shift, and the “delete” address is where data will be overwritten by the shift. The modified row decoder applies the shift clocks to all of the cells within the selected range. First, F1 and B1 are opened to cause the memory cells to switch to a dynamic storage mode. Then F2 is closed only in rows within the selected range of cells to load data from one memory cell to an adjacent cell. After the load, F2 is opened to revert back to the dynamic storage mode. Then F1 is closed to load the shifted data into of the SRAM inverters to prevent data corruption that may occur when B1 is closed to switch back to the static storage mode. The Sidestep shift is completed when new data is written into the “insert” address from the SRAM bit lines when the selected word line is asserted. In this example, data is inserted into address B1 and the original B1–B4 data is shifted up one address with the original data in B5 is being over written with B4 and essentially deleted.

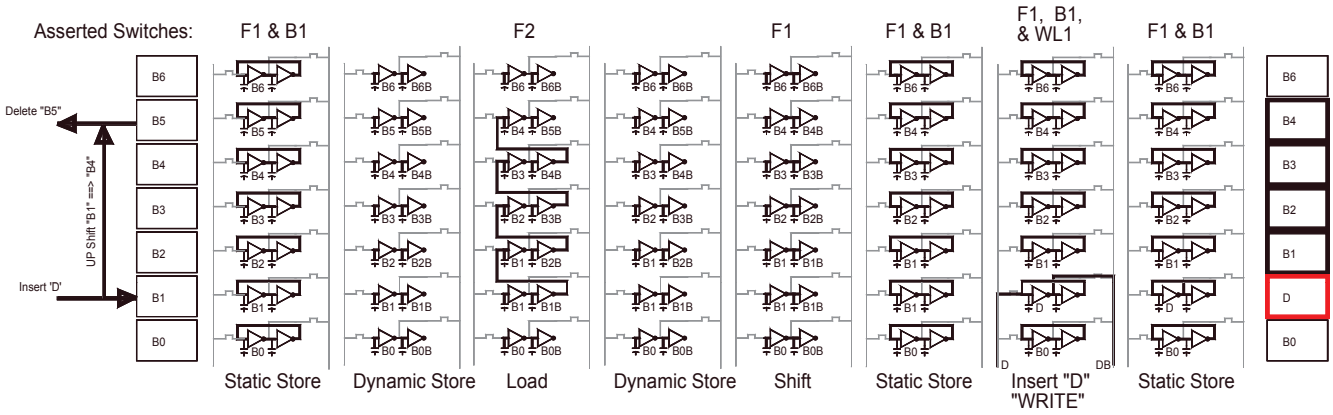


Figure 2: Example of a 1-bit block shift.

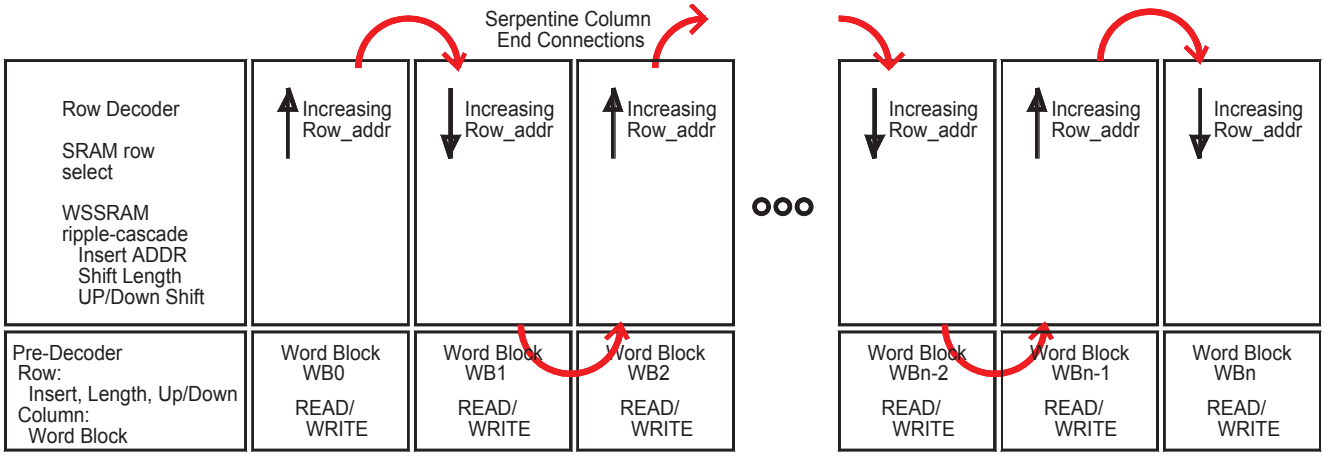


Figure 3: Word Shift SRAM (WSSRAM) block architecture.

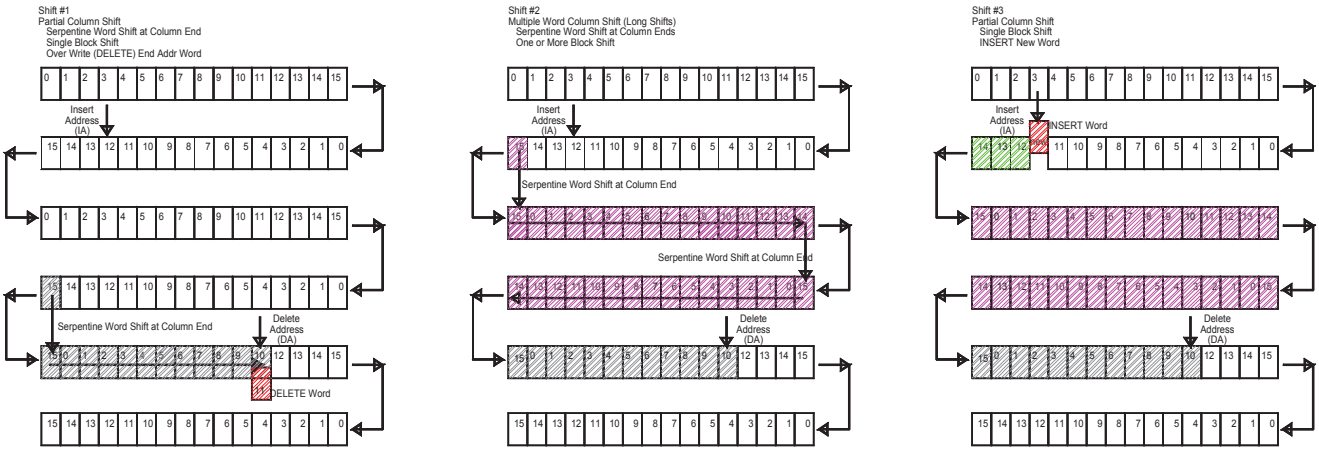


Figure 4: Example of a three-step (long) word shift.

The most challenging innovation in the SRAM Sidestep circuit involves decoder circuitry design. Novel memory block architecture modifications are made to the row decoder, the column decoder, and the block architecture. In the SRAM mode, the row decoder selects one row for read or write operations. For the word-shift mode, the row decoder is transformed into a ripple-cascade control circuit to multiplex the word-shift clock signals to all the rows in the selected shift range. In the word-shift mode, the column decoder is also converted to a ripple-cascade control circuit to select columns of cells for the word shift operation. The memory block architecture is configured as columns of relatively short words folded in a serpentine manner so that long word shifts may be simultaneously done with alternating columns doing up shifts or down shifts. Figure 3 illustrates the WSSRAM block architecture.

Long word shifts will require shifting data spanning multiple serpentine columns, and the long word shift operation must be done in a specific order. Refer to Figure 4, a three step, long word shift. Multiple shifts are necessary to allow partial column word shifts for columns containing the "insert" word and the "delete" word. The shift order is to first shift the data in the "delete" word column, a partial column shift with one word at the end of the column shifted from an adjacent column. Then all the columns between the "insert" and "delete" column are simultaneously shifted with words inserted at the top of the columns coming from adjacent columns. And a third partial shift in the "insert" word column followed by an SRAM write operation into the "insert" address location. The worst case shift delay for long word shift operations is three shift cycles. A well-designed circuit ensures that a shift cycle is the same as the SRAM read/write cycle (t_SRAM).

There are two key performance parameters for the WSSRAM, shift time and power consumption. The shift time for long word shifts is fixed at three t_SRAM delays. Short shifts could be done in one t_SRAM delay, but considering the serpentine block architecture, a safe assumption for a short shift is two t_SRAM delays. And shift lengths greater than the SRAM column length (1 to 4 Kwords) will require three t_SRAM delays. Even though the power consumed by the shift operations is dynamic power and is relatively small per unit cell, large groups of words moving in simultaneous shifts with many active control lines will result in a relatively high power consumption per shift. The power consumption can be managed by limiting the memory block size, reducing the Vdd power supply voltage, or by increasing the shift delay time. A mature CMOS technology with a 2.2 V power supply, 200 nm features, and a t_SRAM delay of 10 ns would require the active WSSRAM block to be 1 Mbit or smaller to limit the power consumption to less than 500 mW. CMOS technologies to be available around 2014 with 25 nm features for SRAM and CPUs would offer a 0.8 V Vdd, a t_SRAM of 2 ns, and a predicted power consumption of 63 mW for 4 Mbit block sizes. Larger WSSRAM memory arrays may be assembled with 1 Mbit or 4 Mbit sub-blocks to mitigate the power consumption issue.

Practical WSSRAM memory sizes are shown in Table 1. The WSSRAM unit cell is about 2.4× larger than the basic six transistor CMOS SRAM unit cell. Four CMOS technology nodes are considered to demonstrate the sizing of WSSRAM memory blocks. About one square centimeter of silicon chip area is assumed to be a practical limit

	Production Year				
	~2000	~2006	2009	2014	Units
1/2 Wire Pitch, nm (F)	200	100	50	25	nm
6T SRAM Cell (~$60F^2$)	2.40	0.60	0.15	0.04	μm^2
12T WSSRAM cell ($144F^2$)	5.76	1.44	0.36	0.09	μm^2
WSSRAM Area (0.7 Overhead Factor)					
1 Mbyte	**66**	**16**	**4**	**1**	mm^2
8 Mbyte	527	**132**	**33**	**8**	mm^2
16 Mbyte	1053	263	**66**	**16**	mm^2
32 Mbyte	2107	527	**132**	**33**	mm^2
64 Mbyte	4213	1053	263	**66**	mm^2
128 Mbyte	8426	2107	527	**132**	mm^2
256 Mbyte	16852	4213	1053	263	mm^2

Table 1: WSSRAM layout area. Practical designs shown in **green**.

for a commercial memory chip, and with this assumption a practical WSSRAM chip with today's mature CMOS technology is 1 to 8 Mbyte, and, with near future technologies, chip densities of 32 to 128 Mbyte are possible.

In summary, an SRAM using the novel WSSRAM unit cell, sophisticated new decoder circuitry, and a serpentine block architecture is a practical solution to build a fast, competitive Sidestep memory. The WSSRAM unit cell is a modified SRAM unit cell that allows columns of words to communicate to adjacent words by reconfiguring a portion of the SRAM to function as a load/store shift register and perform long word shifts without a need to read and re-write stored data. Long word shifts are done in a fixed delay time that is approximately equal to three times a fast SRAM read/write delay. The serpentine array structure allows the word shift memory to share many of the density advantages of large SRAM and DRAM memories and suggests large WSSRAM circuit blocks are practical to include in large systems.

2.2 Block-Level Sense Amp Implementation

The implementation discussed for SRAM in the previous section supports a large shiftable region, which is limited only by the size of a bank. Each bit is also transferred a minimum distance, just enough to the appropriate nearby row or column. Both these features are desirable; the former provides more flexibility for programmers and the latter keeps the shift energy low. However, this design has two drawbacks that limit its application: first, every SRAM cell is extended to include additional switching circuits to support row and column shifts. This reduces the array density and increases the cost per bit. Charge-based memory technologies are amenable to such changes, without increasing the amount of charge (and hence cell size). For example, a DRAM cell does not have sufficient charge to transfer and store it to a nearby cell without going through an amplifier. This section discusses an alternate design, which trades off flexibility and energy to reduce cost and support any byte addressable memory.

Memory is typically organized as a set of arrays. Each array consists of a grid of memory cells interconnected by wordlines and bitlines, decoding logic to activate a particular wordline, input drivers to send data to bitlines (for write operations), and sensing logic to retrieve data from bitlines (for read operations). To access data from memory, a request carrying address is first sent from the controller to the memory. Decoding of address happens in multiple stages. The first stage decoder uses a part of the input address to identify the appropriate memory array and route the request to the array. Once the request reaches the array, the local decoder in the array uses a subset of the remaining address bits to activate the appropriate wordline. The memory cells connected to the activated wordline transfer content to their bitlines, which is amplified by sense amplifiers. The final decoder selects a subset of the amplified value and routes it back to the controller. For writes, the decoding process is similar but bitlines are used to modify the contents of cells using an input write driver.

The latency and energy of a read or a write is typically dominated by long wire traversals to exchange data between memory array and the controller. To support Sidestep, we propose modifying the peripheral logic in each array such that, after a sense-amp amplifies the bit read from its bitline, it automatically routes it through a multiplexor to the input driver of another bitline separated by n columns. Since all the sense-amps can do this shift in parallel, the

entire Sidestep operation can be done in constant time. However, a shift operation is limited to a particular row in a bank. In a modern DRAM, a row size in a DRAM chip is 1 Kbyte.

To allow memory to be reconfigured among a small range of different shift widths without increasing the complexity of multiplexor and control logic in the memory array, we propose interleaving data blocks at different granularity. This also increases the effective shiftable region size; for instance, a block interleaved across four banks will have a shiftable region of 4 Kbyte. Since an x4 DIMM will activate 16 DRAM chips per access, the shiftable region can easily be 16 Kbytes with interleaving.

2.3 Impacts of Emerging Memory Technology

As demonstrated above, Sidestep-enabled memory can be implemented with a straightforward modification of the two common volatile memory blocks, SRAM and DRAM. Emerging memory devices including memristors, PCM and STT-MRAM, have the potential to revolutionize the memory ecosystem by acting as "universal" memory: dense and low-latency (like DRAM), logic CMOS compatible and fast (like SRAM) and non-volatile (like SSD or HD). Given the fact that these emerging technologies may eventually supersede both SRAM and DRAM, it is important to consider how their arrival will impact and/or improve the Sidestep implementations discussed above.

Before discussing how Sidestep could play into emerging memory technologies, let us look at the proposed implementations from a high level perspective. The shift operation of the DRAM approach is achieved with a *block-level* shift in the logic at the periphery of the block. Since the added logic required for a shift is located at the periphery of the block, the added overhead per bit is amortized over a large number of bits and is effectively negligible. However, the length of the shiftable region is constrained to the block width, generally on the order of 1 Kbit. In contrast, the SRAM approach utilizes six extra transistors per bit in order to enable a *cell-level* shift operation. This allows for shifts that are as long as the entire memory (on the order of 100 Mbyte/chip in 2014) but comes, of course, at the expense of decreased density (12 transistors/bit).

For all of the emerging memories, a major driving force for their development is the density improvement that is gained by fabrication of the bits in multiple layers above CMOS. In the context of the two Sidestep approaches, cell-level and block-level, the emerging memory scaling paradigm is thus more closely matched to the block-level approach because it is driven by the removal of transistors from the memory block. Under the block-level Sidestep approach, regardless of whether or not the particular devices in the memory block are DRAM or memristor, the peripheral circuitry can be optimized to read, write and shift bits in the block. It is easy to imagine, then, that the DRAM approach outlined above could be extended to new memories in a straightforward manner as they become available.

As an added feature, the effective line lengths of stacked emerging memory blocks are shorter than 2D DRAM blocks because the third dimension allows for better geometric block mappings. Since line lengths limit of block sizes due to RC delay, it is conceivable that block widths could improve by a factor roughly equal to the number of layers. Thus, the availability of a four layer emerging memory could improve the memory density and length of the shiftable region by a factor of four.

Without getting into the details of specific circuits or a specific emerging technology, it is clear that the density of the SRAM cell-level approach cannot be improved by more than roughly a factor of two with the addition of a new device to hold memory. The cell-level approach will always require local switching circuitry at each bit to enable, direct and drive the shift. The possibility that this could be done with an emerging memory device with six transistors, half the number in the WSSRAM design above, is optimistic but not inconceivable. In this case the bit would be located in the layers above the CMOS and would not contribute to the area overhead. Thus the total memory density would improve by a factor of two and the entire memory would still be shiftable.

Table 2 summarizes and contrasts the different technology approaches and the impact that emerging memory technology is expected to have on each approach. Generally speaking, the block-level approach is about an order of magnitude denser than the cell-level approach while the cell-level approach allows for shiftable blocks that are four orders of magnitude larger. The software applications of a Sidestep co-memory will dictate how dense and how long the shiftable regions need to be for substantial improvement in performance. The market entry point as a co-memory (§ 2.4) allows for flexibility since both approaches could potentially be used in the same socket depending on the

Technology	Memory size (Gbyte/cm^2 at 25 nm)	Shiftable region (Mbyte)
DRAM	5	0.010
Emerging block-level	20	0.050
SRAM	0.15	150
Emerging cell-level	0.30	300

Table 2: Impact of emerging memory on Sidestep implementations.

desired application. As emerging memories become available, Sidestep is poised to take advantage of them in both the block-level and cell-level configurations. Several prominent concerns that will need to be worked out for the implementation of new memory technology will be how to deal with shifts if ECC is required and if multilevel cells are available.

2.4 Architectural Integration

Sidestep memory is unlikely to be deployed in all market segments, e.g., for mobile cost and power sensitive devices ranging from laptops through cell phones. Another issue is that it is unlikely that CPU vendors will create Sidestep-compatible caches in the near future. The result is that Sidestep memory will be off-socket and the remaining question is how will it be physically connected and accessed by application or systems programmers.

In a sense, Sidestep is similar to other domain-specific accelerators. The main difference is that Sidestep is an application-specific memory accelerator rather than the more conventional compute accelerator such as a GPU or the specialized IP blocks in a cell phone that perform video, audio, encrypt/decrypt, Viterbi encoding, or signal processing of various forms such as rake reception, etc. The benefits of specificity are improved performance per energy. Currently memory system architects debate whether non-volatile storage options such as FLASH, STT-MRAM, FeRAM, or memristive memories should be viewed as a level in the main memory system hierarchy or a separate and independently and separately controlled *co-memory*. Sidestep memory devices face a similar conundrum. We argue that the answer for Sidestep is simple. If Sidestep-compatible caches are unlikely, then the most compelling architectural view is that it should be viewed as a co-memory.

Given that off-processor signal pins are dedicated to either I/O or DDRx memory channels, there are essentially two choices for where to attach Sidestep memory components. Today's processors contain integrated memory controllers which drive DRAM specific channels that follow standard (commonly JEDEC) protocols. This trend is likely to be persistent. Sidestep memory, whether the devices are implemented with DRAM, SRAM, or other memory technologies, is unlikely to be compatible with commodity DRAM specific protocols due to differences in both timing and command sequences. We believe that initially, the right choice is to map Sidestep memory into the I/O space partition, much like graphics frame buffer memory has been attached in the past.

Since memory controllers are now on the processor die, chipset architecture has changed dramatically. All non-main memory functions are handled by a separate die. For example, the Intel Nehalem class processors interface with a separate I/O controller hub (ICH) such as the ICH10-1155 or -1156 chip. The ICH device brokers all of the I/O functions and provides specific support for numerous common I/O device protocols such as SATA, USB 2.0, digital audio, PCI, PCI express, etc., as well as less protocol specific general purpose I/O (GPIO). The current Intel GPIO interface is an 8-bit wide packet based approach. For Sidestep integration this is attractive since Sidestep commands, data, and addresses can be easily encapsulated in this format. The downside of this processor plus ICH approach is that there is an intermediate device that adds both latency and power consumption to a Sidestep memory access. However this current I/O model enables us to experiment and quantify the Sidestep capability benefit.

The other advantage of treating Sidestep memory accesses as an I/O transaction is that unlike normal memory transactions that may have a variety of consistency options, I/O events must be atomic, namely they are performed in-order and exactly once to prevent non-deterministic behavior. Hence even parallel writes must atomically perform a read-modify-write transaction. For Sidestep there are three types of writes, namely SHIFT-AND-WRITE, READ-AND-SHIFT, and ordinary WRITE. Each write type changes the content or the location of values. Guaranteeing atomicity is straightforward since there is only one ICH.

The basic functionality of Sidestep memory can be exposed to application-level software via standard portable interfaces (e.g., POSIX `mmap` and `memmove`) and via Sidestep-specific libraries. Under the hood the libraries can activate shifts by storing commands to special locations in Sidestep memory areas, eliminating the need for ISA extensions.

3 Conclusions & Prospects

We live in interesting times. Disruptions, including the emergence of non-volatile memory, may create opportunities for radically re-thinking the traditional division of responsibility between processor and memory as well as the traditional ways in which software interacts with hardware. HP is well positioned to contemplate disruptive changes because HP's technological knowledge spans nanodevices, circuits, architecture, software, databases, and theory. Sidestep is an example of massively parallel in-memory logic that enables the co-design of software for a range of commercially important computing problems.

References

[1] M. Herlihy, "Impossibility and Universality Results for Wait-Free Synchronization," *PODC* 1988.

[2] P.-S. Kim and A. Kutzner, "On Optimal and Efficient In Place Merging," *Proc. International Conference on Current Trends in Theory and Practice of Computer Science* (Jan. 2006), pp. 350–359.

[3] D.E. Knuth, *The Art of Computer Programming, V. 3: Sorting & Searching*, 2nd Ed., Addison-Wesley, 1998.

[4] N.G. Leveson and C.S. Turner, "Investigation of the Therac-25 Accidents," *Computer* 26(7), 1993.

[5] R. Sedgewick, *Algorithms in C*, 3rd Ed., Addison-Wesley, 1998.

[6] M. Stonebraker and R. Cattell, "Ten Rules for Scalable Performance in 'Simple Operation'," *CACM* 54(6), 2011.