Virtual Ways: Low-Cost Coherence for Instruction Set Extensions with Architecturally Visible Storage

THEO KLUTER, Bern Technical University of Applied Sciences
SAMUEL BURRI, Ecole Polytechnique Fédérale de Lausanne
PHILIP BRISK, University of California, Riverside
EDOARDO CHARBON, Technical University of Delft
PAOLO IENNE, Ecole Polytechnique Fédérale de Lausanne

Instruction set extensions (ISEs) improve the performance and energy consumption of application-specific processors. ISEs can use architecturally visible storage (AVS), localized compiler-controlled memories, to provide higher I/O bandwidth than reading data from the processor pipeline. AVS creates coherence and consistence problems with the data cache. Although a hardware coherence protocol could solve the problem, this approach is costly for a single-processor system. As a low-cost alternative, we introduce Virtual Ways, which ensures coherence through a reduced form of inclusion between the data cache and AVS. Virtual Ways achieve higher performance and lower energy consumption than using a hardware coherence protocol.

Categories and Subject Descriptors: B.3.2 [**Hardware**]: Memory Structures—*Design styles*; C.3 [**Computer Systems Organization**]: Special-Purpose and Application-Based Systems—*Real-time and embedded systems*

General Terms: Performance, Design, Experimentation

Additional Key Words and Phrases: Instruction set extension, architecturally visible storage, memory coherence, memory consistence, Virtual Ways

ACM Reference Format:
Theo Kluter, Samuel Burri, Philip Brisk, Edoardo Charbon, and Paolo Lenne. 2014. Virtual Ways: Low-cost coherence for instruction set extensions with architecturally visible storage. ACM Trans. Architec. Code Optim. 11, 2, Article 15 (June 2014), 26 pages.
DOI: http://dx.doi.org/10.1145/2576877

1. INTRODUCTION

Embedded systems often require application- or domain-specific acceleration to meet performance and energy requirements. One way to improve the performance and energy efficiency of an application-specific processor is through instruction set extensions (ISEs), which are added to the instruct set of architecture (ISA) of the processor [Gonzalez 2000, 2006; Halfhill 2003; Hameed et al. 2011]. Compiler techniques to automatically identify and synthesize application-specific ISEs are mature

Authors' addresses: Theo Kluter, Bern University of Applied Sciences, Engineering and Information Technology, Quellgasse 21, CH-2501 Biel, Switzerland; email: Theo.Kluter@bfh.ch; Samuel Burri, EPFL SCI STI EC, INF 131 (Bâtiment INF), Station 14, CH-1015 Lausanne, Switzerland; email: samuel.burri@epfl.ch; Philip Brisk, Winston Chung Hall Room 339, Department of Computer Science and Engineering, University of California, Riverside, Riverside CA 92521, USA; email: philip@cs.ucr.edu; Edoardo Charbon, HB 17.310, Department of Microelectronics, Delft University of Technology, Mekelweg 4, 2628 CD Delft, The Netherlands; email: e.charbon@tudelft.nl; Paolo Ienne, EPFL IC ISIM LAP, INF 137 (Bâtiment INF), Station 14, CH-1015 Lausanne, Switzerland; email: paolo.ienne@epfl.ch.

DOI: http://dx.doi.org/10.1145/2576877

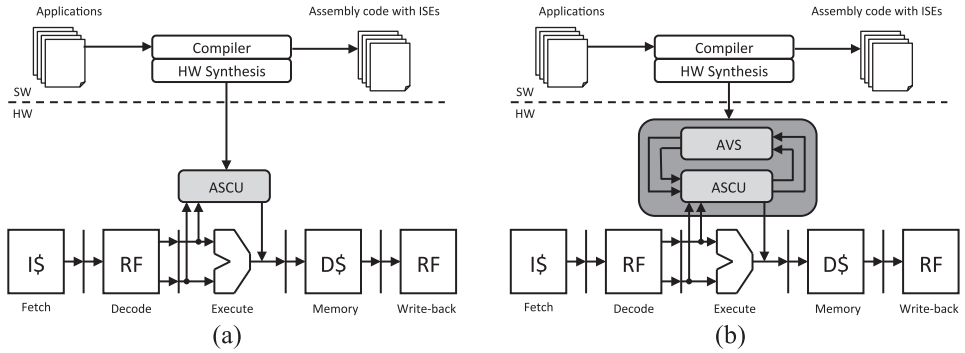


Fig. 1. (a) A compiler and hardware synthesis tool automatically configures a processor with application-specific extensions. The compiler identifies ISEs and generates assembly code that invokes them . The hardware synthesis tool adds the ISEs to the processor's ISA, generates HDL for the ASCU and integrates it into the pipeline. (b) Integrating AVS into the ASCU provides a more efficient data transfer mechanism than communicating directly with the processor pipeline.

[Pozzi et al. 2006; Verma et al. 2010; Atasu et al. 2012]. When integrated into a processor pipeline, the hardware implementation of an ISE is referred to as an application-specific custom unit (ASCU), as shown in Figure 1(a). A synthesis tool integrates the ASCUs for all of the ISEs into the hardware description language (HDL) implementation of the processor, which can then be laid out and fabricated.

Data transfers between the processor register file and the ASCU are a performance bottleneck [Pozzi and Ienne 2005]. I/O bandwidth can be improved by providing the ASCU with its own local memory, known as architecturally visible storage (AVS) [Biswas et al. 2007], as shown in Figure 1(b). The AVS is placed under compiler control, similar to a scratchpad memory [Banakar et al. 2002]; at each point during program execution, the compiler chooses which data structures (typically arrays) reside in the AVS. Direct memory access (DMA) transfers stream data between main memory and AVS, bypassing the processor and its caches [Biswas et al. 2007]. Unfortunately, this creates memory coherence and consistence problems between the AVS and the processor's data cache.

One solution is to employ a hardware coherence protocol [Kluter et al. 2008]. This is economical when designing a coherent multiprocessor system on a chip (MPSoC); however, coherence protocols are costly in terms of performance, area, and energy consumption, and many MPSoCs do not require coherence protocols because the designer partitions all exclusive (nonshared) data among private caches and AVS, while putting all shared data (if any) into a larger shared memory. In these types of systems, lower-cost solutions than hardware coherence protocols are desirable. Additionally, the area and energy overhead of introducing a coherence protocol to manage multiple parallel memories in a single-processor system would be prohibitive.

This article introduces Virtual Ways, a low-cost alternative to a hardware coherence protocol, which can maintain coherence between the data cache of a processor and one or more AVS memories. Virtual Ways alters the cache's hit detection circuitry and state machine controller to provide this capability. Virtual Ways is shown to achieve higher performance and lower energy consumption than traditional approaches to hardware coherence in application-specific processors with AVS-enhanced ISEs. Our article is an extension to a conference publication [Kluter et al. 2010] and includes much more detail on the AVS architecture and operating behavior (Section 3), a detailed cost model for communication overheads to transfer data to and from the AVS (Section 4), and far more extensive experimentation (Sections 5 and 6).

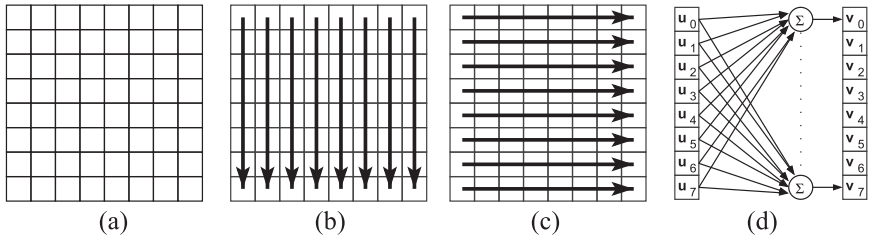


Fig. 2. The IDCT kernel of the JPEG decompression algorithm performs its operation on an 8 × 8 input array (a), by first performing eight 1D-DCTs (d) on the columns (b) and then on the rows (c).

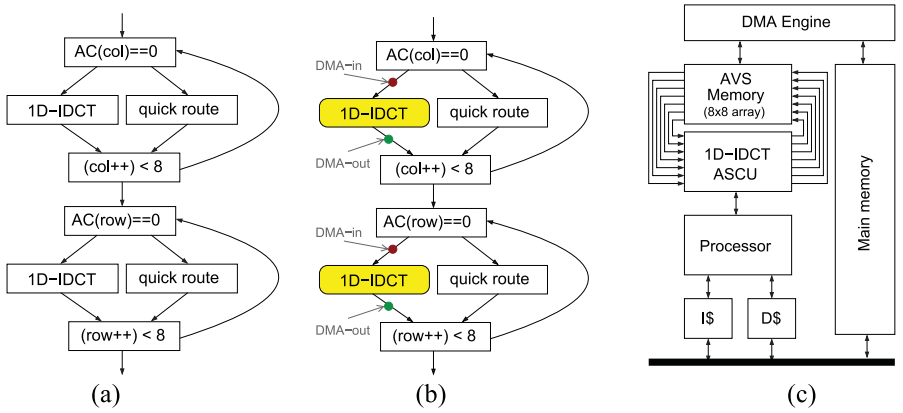


Fig. 3. (a) The IDCT kernel. (b) The algorithm of Biswas et al. [2007] identifies a 1D-IDCT kernel as an ISE with an AVS memory that holds an 8 × 8 array. (c) The complete system to accelerate the IDCT kernel, with the ISE implemented in hardware as an ASCU.

2. MOTIVATING EXAMPLE

The Inverse Discrete Cosine Transformation (IDCT) of the JPEG decompression algorithm [Halfhill 2000] is used as a motivating example. The IDCT kernel performs a fixed-point transformation on an 8 × 8 array of input values, shown in Figure 2(a). The kernel applies a 1D-IDCT to each of the eight columns (Figure 2(b)), followed by a 1D-IDCT to each of the eight rows (Figure 2(c)). Figure 2(d) graphically represents the 1D-IDCT.

Figure 3(a) depicts a software-optimized version of the IDCT. It includes a quick route that is invoked whenever the AC-component of a row or column is zero. In Figure 3(b), the full 1D-IDCT has been selected as an ISE, and DMA transfer instructions are inserted graphically into the program [Biswas et al. 2007]. Figure 3(c) shows the complete system with a 1D-IDCT AVS-enhanced ASCU.

2.1. The Memory Coherence Problem

Figure 4 depicts an execution sequence that leads to a memory coherence problem. The array is loaded into the processor's cache and then into the AVS memory by a DMA transfer. Several 1D-IDCT operations modify the array in the AVS memory, but these modifications are not propagated to the copy that resides in the cache. The DMA-out operation after the 1D-IDCT copies the updated array to main memory, bypassing the cache. A proper coherence mechanism would have invalidated the copy of the array residing in the data cache; however, no such mechanism is present in the system.

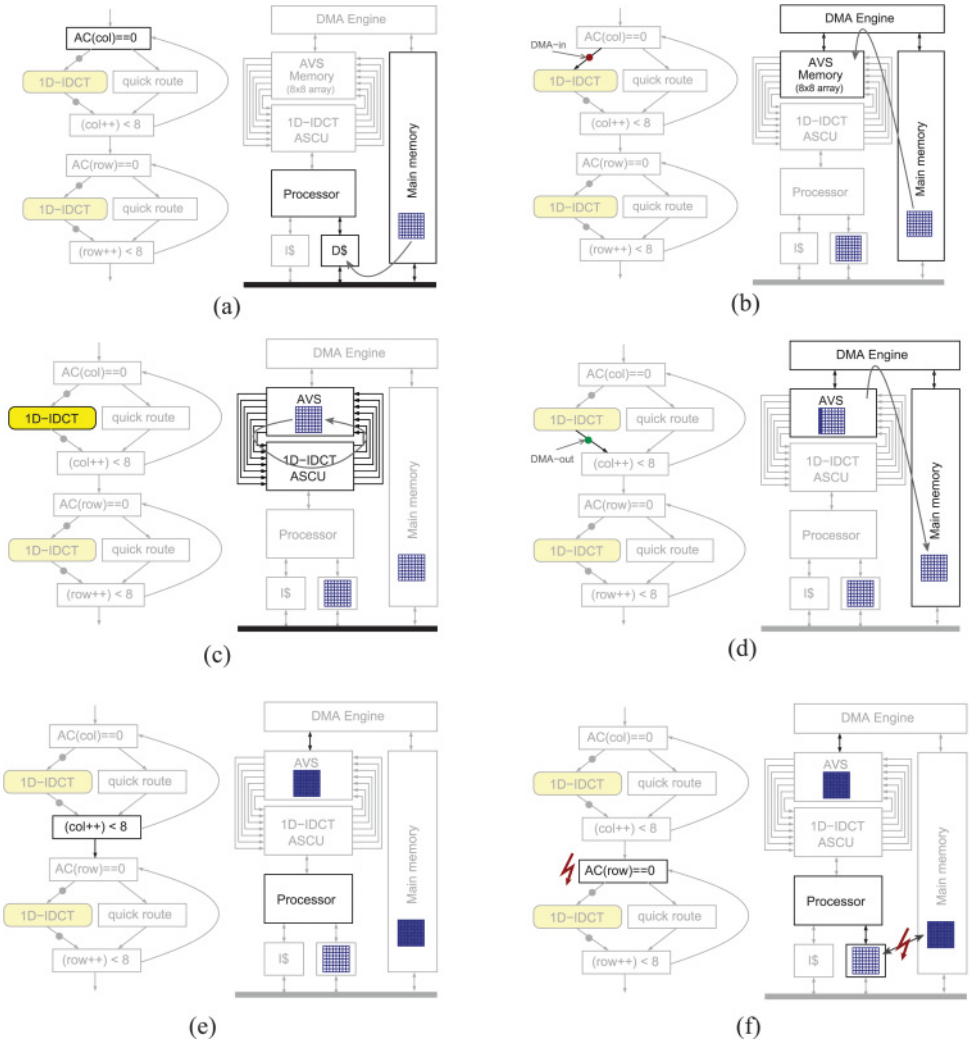


Fig. 4. A control sequence through the software-optimized DCT code that leads to a coherence problem between the AVS and cache in the processor. (a) The IF construct gradually loads the array into the data cache. (b) The DMA instruction copies the array from main memory into the AVS. (c) The ASCU potentially modifies one column of the array stored in the AVS. (d) The DMA instruction copies back the modified array to main memory; the data cache is unaware of these changes. (e) After having processed all columns, the program continues to process the rows. (f) A memory coherence problem occurs as the data cache still contains the old copy of the array loaded before the columns were processed.

In Figure 4(f), a software operation reads the array; the load operation hits in the cache, loading an invalid datum into a register. The if-statement could make an incorrect decision as a result of reading the invalid data; moreover, further memory activity in this code region could evict the invalid copy of the array from the cache. This would overwrite the valid copy of the array in main memory with an invalid copy.

2.2. The Memory Consistence Problem

A naïve solution to the memory coherence problem is to flush the data cache prior to the DMA-in actions, as shown in Figure 5(b). Flushing the cache invalidates all of its

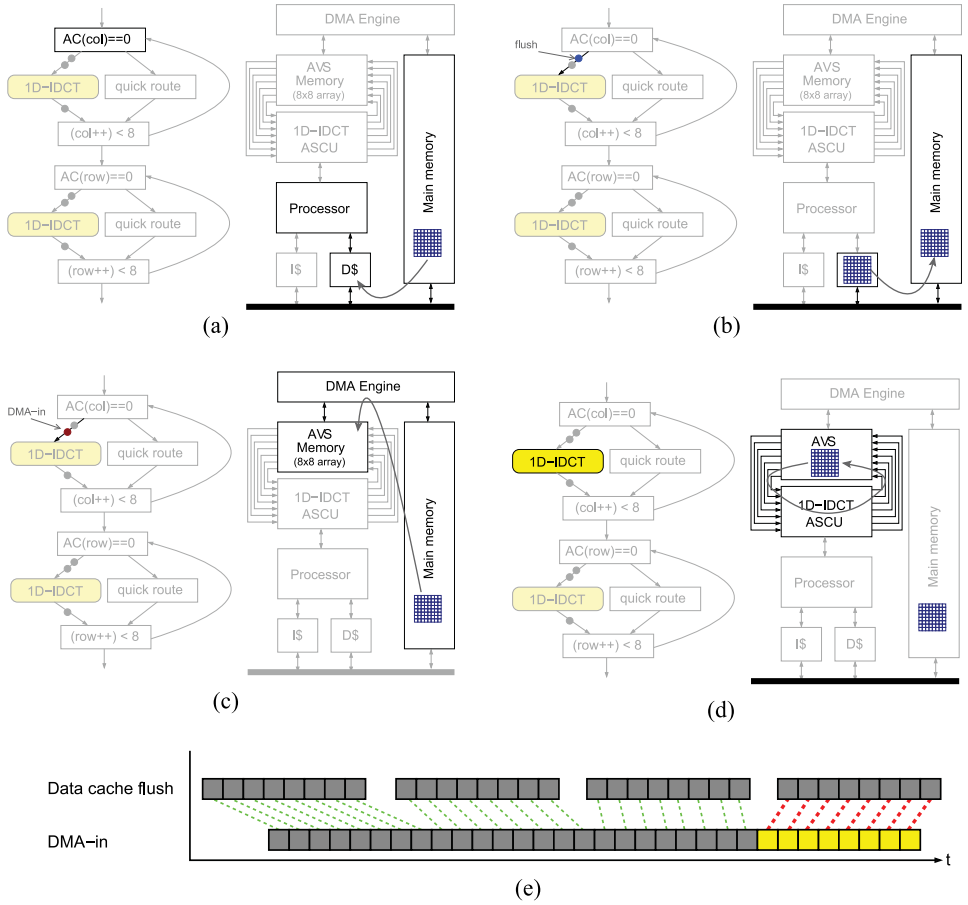


Fig. 5. A sequence of operations in the software-optimized DCT code that leads to a memory consistence violation; a cache flush operation preserves coherence but cannot ensure consistence. The IF construct gradually loads the array into the data cache (a); the data cache flush instruction evicts all data from the data cache back to main memory (b); the DMA instruction copies the array directly from main memory into the AVS (c); normal operation continues; however, correct execution is impeded by a consistence problem (d). Although the DMA instruction starts after the cache flushing, it executes faster; due to faster execution, its consumption rate exceeds the rate of the flush operation. The program correctly specified the order of operations; however, the hardware does not respect this order, potentially resulting in incorrect execution.

copies of the data structure and updates the copy in main memory. This guarantees that no copy of the array remains in the data cache, and thereby ensures coherence because the IDCT begins with main memory holding the only copy of the data. Figure 5(e) illustrates the memory consistence problem: assuming that the cache flush is a nonblocking operation, the DMA-in operation may copy data into the AVS memory before it is flushed into main memory; the AVS receives invalid data corrupting the IDCT. Using a blocking flush corrects the problem, but at an inordinate cost in terms of performance.

3. VIRTUAL WAYS

The most straightforward solution to the memory coherence and consistence problems outlined in the preceding section is to use a hardware coherence protocol, as shown in Figure 6(a); this was the approach taken by Speculative DMA [Kluter et al. 2008],

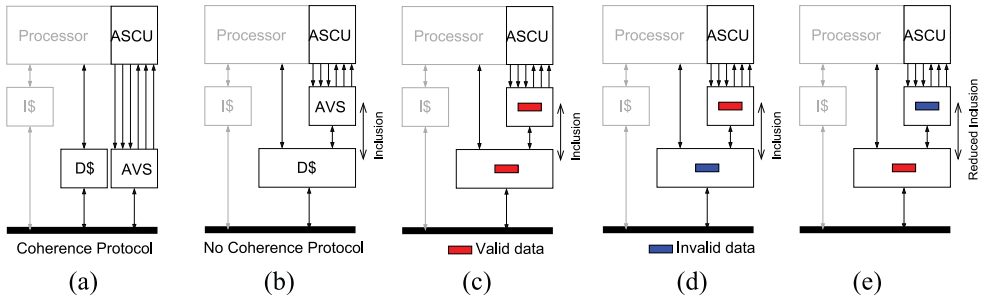


Fig. 6. (a) The AVS is placed in parallel with the data cache; a hardware coherence protocol maintains coherence and consistence between the data cache (D$) and AVS. (b) Virtual Ways eliminates the hardware coherence protocol by placing the AVS memory above the D$ in the memory hierarchy; an AVS-aware cache controller provides coherence and consistence. (c) Under the principle of inclusion, all cache lines held in the AVS must be a subset of those held in the D$. Initially, a line loaded into the AVS from the D$ is valid. (d) If the ASCU modifies a cache line in the AVS, then the copy in the data cache becomes invalid; Virtual Ways ensures that the processor receives a valid copy of the data if it issues a load to the data cache. (e) If the processor modifies a cache line in the D$, then the copy in AVS becomes invalid. This policy is semi-inclusive, because the cache lines in the AVS remain a subset of those in the D$, but with no guarantee to be valid. Under Virtual Ways, future ASCU loads from the AVS will read data from a valid cache line.

which is representative of the memory architecture shown in Figures 3 through 5. Virtual Ways, in contrast, as shown in Figure 6(b), places the AVS above the data cache in the memory hierarchy and modifies the cache controller to provide coherence and consistence. In a traditional multilevel cache hierarchy, the processor communicates directly with the highest level of the hierarchy. In Virtual Ways, the ASCU communicates with the highest level (AVS), whereas the processor communicates with the data cache, one level lower than AVS.

To use Virtual Ways, the compiler or programmer introduces a software prefetch instruction called AVS-in into the program to load a complete data structure (array) into the cache; it then copies each cache line containing the data structure into the AVS. The AVS-in operation is blocking, which ensures that the complete data structure is brought into both the cache and AVS before the ISE start to execute. Virtual Ways extends the data cache state machine controller to become cognizant of this prefetching mechanism. The choice to simultaneously bring data into the data cache and AVS enables them to share the controller, as long as they share the same cache line granularity.

Virtual Ways does not employ an AVS-out operation to evict data; instead, it uses a lazy write-back scheme, which evicts data from AVS upon request. In a single-processor system, the only request could be a read or write issued by the processor that goes to the data cache; in a multiprocessor system, a coherence protocol could also trigger an action if another processor issues a read or write to a different copy of the same cache line.

The name “Virtual Ways” is based on the observation that the tag and state information of the AVS is already present in the data cache; in this respect, the AVS is like a way of the cache; however, only the ASCU, not the processor, can access data in the AVS, unlike a cache. The AVS therefore acts like “virtual ways” of the cache.

3.1. Inclusion Policies vis-à-vis Virtual Ways

Virtual Ways supports a nontraditional hybrid inclusion/exclusion policy, where the data cache may or may not contain a copy of the data in the AVS. Figures 6(c)–(e) illustrate different aspects of the policy. The situation shown in Figure 6(e) is nontraditional because the highest level of the memory hierarchy (AVS) has an invalid copy of a cache line, whereas a lower level (L1 data cache) has a valid copy. This is allowed because CPU writes to the L1 data cache invalidate, but do not evict, copies of the same

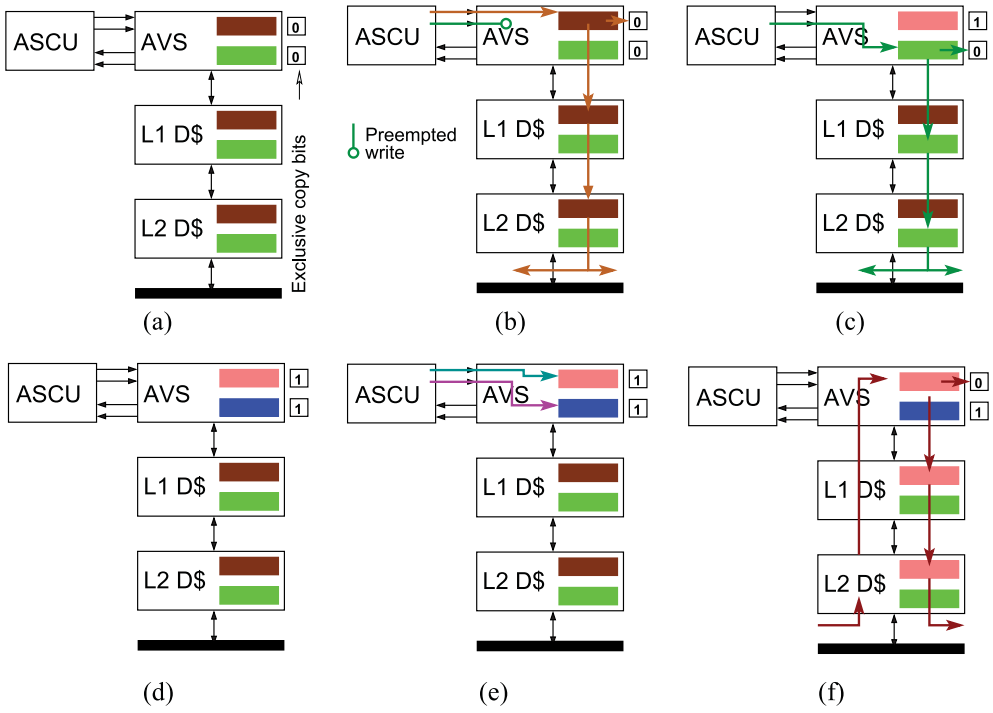


Fig. 7. (a) Initially, both the AVS and L1 data cache store valid copies of a cache line. (b) As the AVS does not contain an exclusive copy of the cache line containing the data structure, parallel writes issued by the ASCU need to be serialized to satisfy inclusion requirements. (c) The sequential write sets the exclusive copy bit of the cache line invalidating the lines in the data cache. (d) Afterward, the AVS contains exclusive copies of the cache lines. (e) All subsequent writes issued by the ASCU can be performed in parallel, as the AVS contains exclusive copies of the cache lines. (f) Exposing the AVS exclusive copy bits to the cache controller allows an AVS write-back request to update the invalid copies of the cache line within the memory hierarchy. For example, if the processor tries to read an invalid cache line in the L1 data cache, this controller will copy the valid cache line from the AVS memory into the cache hierarchy in accordance with the write policy (write-through or write-back).

cache line in AVS. The policy is also exclusive because evicting the cache line from the L1 data cache does not evict the copy from AVS.

Virtual Ways is compatible with both write-through and write-back policies for data caches. In a write-through cache, the updated cache line propagates directly to the lower levels. In a write-back cache, the updated cache line propagates to the lower levels only when it is evicted. As noted earlier, the AVS employs a lazy write-back scheme in our implementation of Virtual Ways, similar in principle to a write-back policy.

Figures 7 through 9 provide two further examples with additional details. These examples are not meant to be exhaustive; they simply illustrate commonly occurring scenarios that pertain to Virtual Ways. Each line in the AVS is augmented with an exclusive copy bit, which pertains to the scenarios shown in Figure 6(d) and 6(e).

An AVS may contain several distinct memories, each of which holds a different data structure; as such, the ASCU may try to write to several memories within the AVS in parallel, as shown in Figure 7(b). In a traditional memory hierarchy, in which the load/store unit of a processor connects to a cache with one read and one write port, an update at the higher level of the hierarchy is propagated to the lower levels in accordance with the write policy. Performing parallel writes is impractical, as it requires a high-cost interconnect at all levels of the hierarchy. An alternative is to

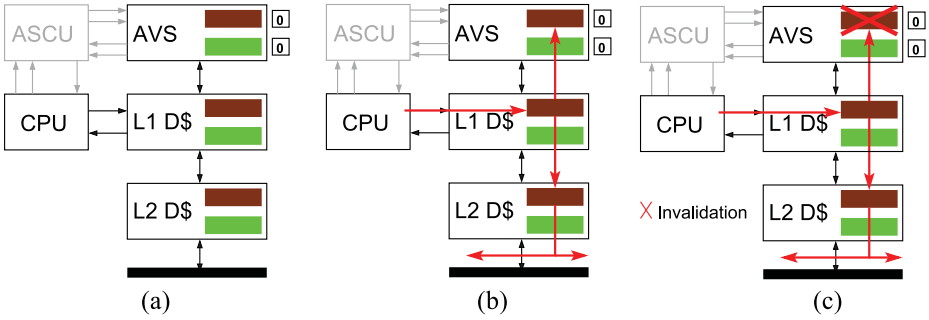


Fig. 8. The sequence of actions that occurs when the processor updates a cache line that is valid in the data cache and AVS. (a) Initially, all levels of the memory hierarchy hold valid copies of the cache line, as mandated by the inclusion policy; the AVS copy is not exclusive. (b) A processor write updates all lower levels of the hierarchy, typical for inclusion. The processor can update the AVS to keep coherence. (c) The processor write, however, can also invalidate the copy in the AVS. Invalidation is preferable as it avoids inter processor consistence problems.

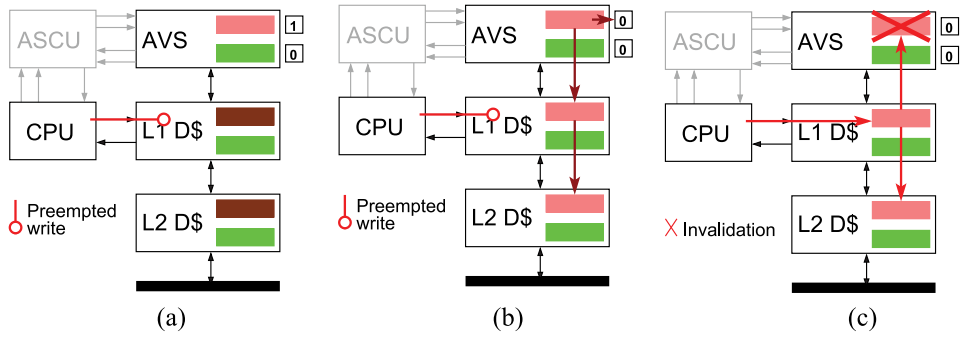


Fig. 9. The sequence of actions that occurs when the processor accesses a cache line that resides in the exclusive state in the AVS. (a) Initially, the AVS contains an exclusive copy of the cache line and the cache(s) contain a dead copy; the access needs to be stalled. (b) The invalid copies of the cache line are restored throughout the memory hierarchy; the AVS copy is no longer exclusive. (c) If the access is a write, then the copy of the cache line in AVS is invalidated.

serialize the parallel writes; however, serialization restricts the exploitation of data parallelism in the ASCU.

The hybrid inclusion/exclusion policy of Virtual Ways solves this problem. Since exclusive cache lines can remain in the AVS after an ISE executes, parallel writes to nonexclusive cache lines in the AVS must be serialized, as shown in Figure 7(b)–7(d). Subsequent writes to exclusive cache lines can then execute in parallel, as shown in Figure 7(e), as the cache lines in the lower levels of the memory hierarchy have already been invalidated. Figure 7(f) illustrates the write-back process, as enabled by the AVS exclusive copy bits.

In Figure 8, the AVS and L1 data cache both contain valid copies of the same cache line. Processor reads from the L1 cache are handled normally; however, processor writes must guarantee inclusion by either updating or invalidating the copy of the cache line in the AVS. Invalidation is preferable, because processors may use write buffers to compensate for write delays. Write buffers provide no real-time guarantees of the latency between the processor write and the actual update; as such, the following consistence problem can occur: (1) the processor write is queued in the write buffer; (2) an ISE is issued and reads data from the AVS; (3) the read occurs before the write

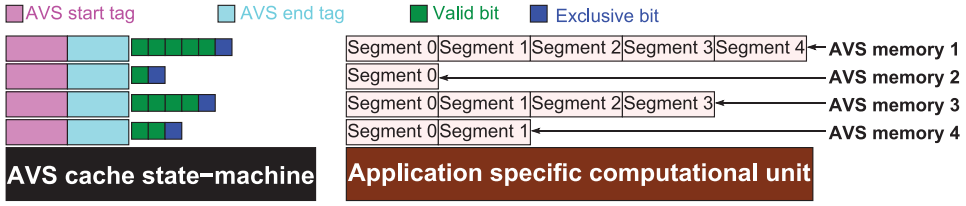


Fig. 10. A block diagram of four AVS memories. Each is essentially an irregular cache, where each cache line is a sub-blocked version of the segmented AVS memories of different size. From the software point of view, the data portion of the cache is direct mapped, as the DMA always transfers a data structure to the same sub-blocked cache line; however, from the perspective of the cache state machine, it is fully associative.

update completes; (4) the ISE accesses an invalid cache line in the AVS; and (5) the write update completes after the ISE reads invalid data. Preemptive invalidation, as shown in Figure 8(c), prevents this problem from occurring.

In Figure 9, the processor tries to write to a cache line that is held exclusively in AVS. In Figure 9(a), the write is preempted; in Figure 9(b) and 9(c), the exclusive copy bit of the AVS is reset and the copy of the cache line in the AVS is invalidated. In the case of a read, the copy of the cache line would propagate from the AVS to lower levels of the cache hierarchy in accordance with the write policy, and the exclusive bit would be reset; however, the copy in the AVS would not be invalidated because it has not been modified.

3.2. AVS Architecture

Figure 10 illustrates the key architectural components of AVS; this example shows an AVS with four distinct SRAM memories. Each SRAM memory is sized to match the storage requirement of the data structure(s) that it will hold, as is essentially a single sub-blocked cache line, using a two-bit, three-state Invalid, Valid, Exclusive (IVE) scheme. A dirty bit is not required, as inclusion implies that the coherence state is only required at the lowest level of the memory hierarchy. Each AVS segment is equal in size to a data cache line. Unlike a cache, the AVS hit/miss detection circuitry must know the start and end addresses of the data structure, stored in AVS start and end tags, respectively.

The compiler determines when the AVS has exclusive access to a data structure and sets the exclusive bits when necessary (as part of the AVS-in operation). This allows the cache/AVS controller to manage multiple SRAM memories at once within the AVS. The AVS can be viewed as being direct mapped, as each data structure is always loaded into a specific SRAM memory; however, the AVS state is hardware managed, as eviction and invalidation is under control of the state machine. In this respect, the AVS cache is fully associative, and the controller determines if each line is valid, modified, or exclusive.

Figure 11 shows the state machine associated with the IVE scheme; the CPU write arc, from the exclusive to the invalid state, enforces the restoring phase shown in Figure 9(b), which is essentially a lazy write-back scheme that does not require compiler-initiated data transfers. The state machine does include an explicit AVS-out operation, which can be useful in situations where the compiler or programmer can identify opportunities to overlap computation and communication.

3.3. AVS Hit and Miss Detection Circuitry

Figure 12(a) shows the physical memory address provided by the processor. In a cache, the address space is divided into three portions: *tag*, *index*, and *select*. Equations (1) through (3) compute the number of bits for each portion as a function of the physical

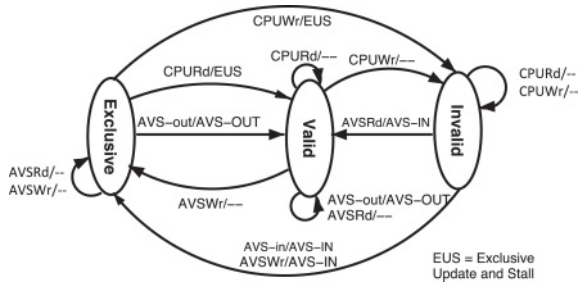


Fig. 11. Diagram of the IVE states of each AVS segment. To avoid confusion, AVS-in/out refers to signals sent from the processor to the controller (as per AVS-in/out ISEs), whereas AVS-IN/OUT refer to the action taken by the memory subsystem. AVSRd/Wr actions should never occur in the "Invalid state"; they are always initialized by the AVS-in action.

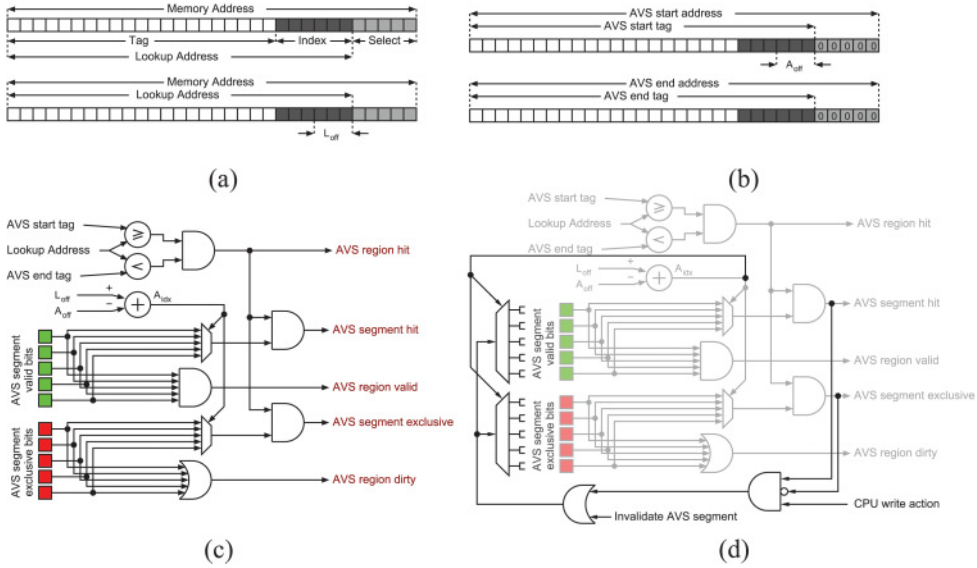


Fig. 12. (a) In a cache, the memory address consists of three fields: tag, index, and select. (b) The AVS memory requires start and end tags, which specify the location of the data structure. The AVS address offset (A_{off}) specifies the segment in the AVS where the data structure starts, as it may not be aligned with the beginning of a cache line. When a memory address is generated and sent to the AVS, the lookup address offset (L_{off}) represents the location of the segment to access relative to the AVS address offset. The index of the segment in the AVS memory is $A_{idx} = L_{off} - A_{off}$. (c) The AVS hit detection circuitry, which compares the lookup address to the AVS start and end tags to determine a hit, uses A_{idx} to select the valid and exclusive bits for the corresponding segment. (d) Extensions to the hit detection circuitry to support fast segment invalidation.

parameters of the cache:

$$B_{sel} = log_2(S_{cl}), \tag{1}$$

$$B_{index} = log_2\left(\frac{S_{ca}}{S_{cl}N_{ca}}\right), \text{and} \tag{2}$$

$$B_{tag} = B_{addr} - B_{index} - B_{sel}. \tag{3}$$

The number of select bits (B_{sel}) depends on the cache line size (S_{cl}); the number of bits required for the index (B_{index}) and tag (B_{tag}) depend on cache size (S_{ca}) and associativity (N_{ca}) as well as S_{cl}. The lookup address (tag and index) are required for hit/miss detection; the selection portion will not be discussed further.

In a cache, B_{sel}, B_{index}, and B_{tag} are integers, which ensures that the number of cache lines (N_{cl}) is an integer power of two—for example, $N_{cl} = 2^{B_{index}}$. In contrast, AVS SRAM memories are application specific and are sized to match the storage requirements of the data structures they will hold, which may not be an integer power of two; consequently, AVS SRAM memories require nontraditional circuits for hit/miss detection. This requires AVS start and AVS end tags, which are derived from the start and end addresses of the data structure in memory, as shown in Figure 12(b). These tags are based on the assumption that the smallest granularity in the system is the cache line size (S_{cl}) and that the AVS contains a continuous memory segment.

Let d_k be the data structure in question. Let $A_{ds}(d_k)$ be the start address of d_k and $N_{se}(d_k)$ be the number of segments in d_k contained in the AVS memory. Then, the AVS start tag, $T_{avss}(d_k)$, and the end tag, $T_{avse}(d_k)$, are computed as follows:

$$T_{avss}(d_k) = \left\lfloor \frac{A_{ds}(d_k)}{S_{cl}} \right\rfloor S_{cl}, \text{ and} \tag{4}$$

$$T_{avse}(d_k) = \left\lceil \frac{T_{avss}(d_k) + N_{se}(d_k) S_{cl}}{S_{cl}} \right\rceil S_{cl}. \tag{5}$$

An AVS-in operation must store the AVS start and end tags.

Figure 12(c) illustrates the hit detection circuitry for the AVS, which differs from that of a cache. The *AVS region hit* signal determines if the generated address matches the range in memory occupied by the data structure, as specified by the AVS start and end tags. It is still necessary to determine whether the data contained in a given segment is IVE. Since the start address of the data structure may not be an even power of two, we compute an AVS index (A_{idx}), which specifies the physical address in an AVS SRAM memory of the actual segment that is being accessed. The number of AVS index bits, $N(A_{idx})$, is the largest power of two that can contain the number of segments in the data structure—that is, $N(A_{idx}) = \lceil log_2(N_{se}(d_k)) \rceil$.

The AVS index is calculated by subtracting the AVS lookup address index offset (L_{off}) from the AVS index offset (A_{off}), as shown in Figure 12(b) and 12(c). A_{idx} provides access to the corresponding valid and dirty bits of the corresponding segment in the AVS SRAM, which determine whether the segment is valid/invalid (*AVS segment hit*) or exclusive (*AVS segment exclusive*).

The *AVS region valid* signal indicates the case when *all* segments in the AVS are valid, and an *AVS region dirty* signal indicates the presence of at least one exclusive segment in the AVS. These signals extend the cache state machine to accommodate Virtual Ways. The state machine can quickly query the IVE state of an array stored in AVS whenever a processor issues a memory address to the L1 cache that hits in the AVS. The state machine can take appropriate action, such as the operations shown in Figures 7 through 9.

3.4. Fast Segment Invalidation

Figure 12(d) illustrates several extensions to the hit detection circuitry that are required to handle AVS invalidations that may occur when the processor updates data in the cache. In this case, three general situations can occur, depending on the state of the AVS segments. If the segment is invalid, then the AVS contains no data and coherence is satisfied implicitly. If the segment is exclusive, then all copies residing outside

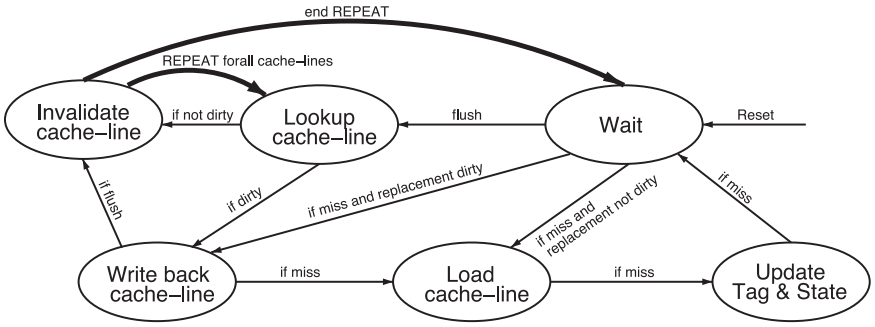


Fig. 13. Flow diagram of the write-back policy of a basic cache controller.

of the AVS are invalid, and the AVS must restore all dead copies that exist elsewhere in the memory hierarchy before the processor can write to the cache (e.g., Figure 9), stalling the processor. If the segment is valid but nonexclusive, then another valid copy exists elsewhere in the memory hierarchy, and the AVS segment can be invalidated safely.

A logical function of the AVS segment hit, AVS segment exclusive, and CPU write action signals, shown in Figure 12(d), determines if a fast segment invalidation is possible. Fast segment invalidation can be performed in parallel with the memory lookup, as it is transparent from the perspective of the processor and does not hinder performance.

3.5. State Machine Controller

Figure 13 introduces a flow diagram of the write-back actions of a standard data cache. Hits in the cache and delivery of data to the processor are omitted for brevity.

Figure 14 extends the controller to support the AVS-out action, using a hybrid-inclusion/exclusion policy in which the data cache may or may not hold a copy of the data in the AVS memory (in contrast, a strictly exclusive policy would mandate that the data cache could not hold a copy of the data). When a cache miss occurs, the controller first checks to determine if the data is present in the AVS before accessing main memory. To support the hybrid scheme, the controller loads the data from the AVS memory ("Load cache-line") if the AVS segment hit signal is active, which indicates that the AVS contains a valid or exclusive copy of the data. On an AVS-out action, the controller checks to see if the cache contains a copy ("Lookup segment related cache-line") and allocates a cache line if it does not contain a copy of the segment. This process follows the normal cache procedure. An AVS-out action can be suppressed if no AVS segments are exclusive, meaning that the ASCU has not modified any of the segments.

AVS region hit = 1 and AVS region valid = 1. The AVS already contains the latest copy of the complete data structure; the AVS-in action can be suppressed.

AVS region hit = 0 and AVS region dirty = 1. The AVS contains at least one exclusive segment, but the AVS-in action is requesting a new data structure. The controller must initiate an AVS-out action (Figure 14) to evict the segment and write it back to the L1 data cache before the AVS-in operation can replace it with the requested data.

AVS region hit = 1 and AVS segment hit = 1. The AVS contains at least one valid or exclusive segment. This segment already contains the latest copy of the data structure and therefore can be skipped. This is the special case of the situation where *AVS region valid = 1* and all segments are valid or exclusive.

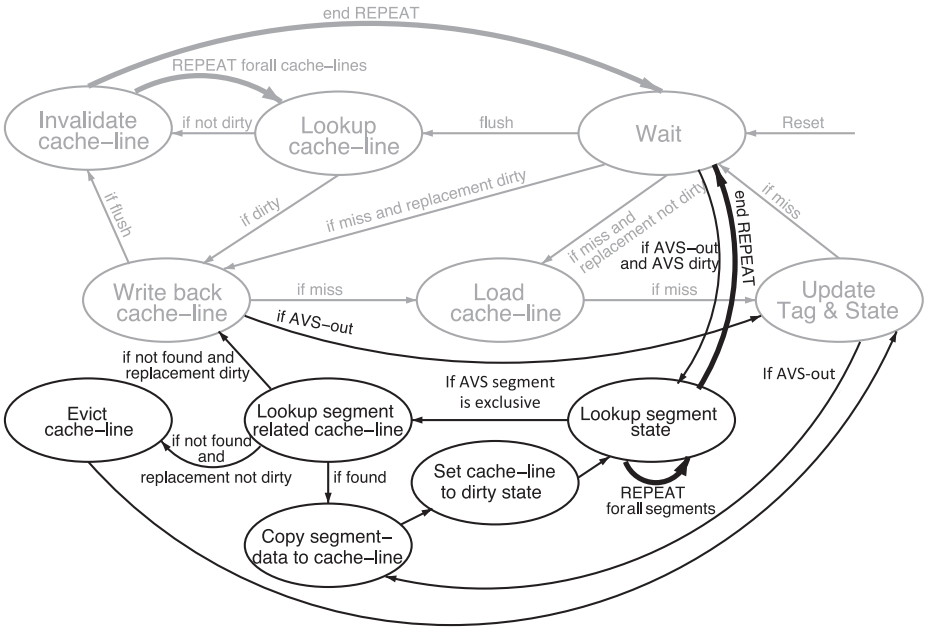


Fig. 14. Extended state diagram with support for AVS-out operations.

Other cases. In all other cases, the AVS does not contain an up-to-date copy of the data structure. When loading a segment, two possible situations can occur: (1) the cache contains the data to be loaded into the segment and can simply copy it into the AVS ("Copy cache-line data to segment"); (2) otherwise, the data must be loaded from a lower level of the memory hierarchy. A standard cache line replacement process is initiated, the data is loaded into the cache and AVS simultaneously, and the segment state is set to valid. After all AVS segments are loaded, the AVS start and end tags are updated.

Figure 16 extends the cache controller to handle the situation shown in Figure 9, where the processor tries to access invalid data in the cache while the AVS contains the exclusive copy. The controller must stall the processor while copying the segment from the AVS back into the cache ("Exclusive Update and Stall"). If the cache access is a read, then the segment in the AVS becomes valid; if it is a write, then the segment becomes invalid, and the controller sets the dirty bit of the corresponding data cache line.

Virtual Ways must support flushing. Under hybrid-inclusion/exclusion, the AVS may contain a copy of the data structure after the cache evicts its copy, so no changes are required to the controller. Under a reduced inclusion policy, in contrast, any data structure that resides in the AVS must also reside in the cache. A flush request must first flush the AVS prior to flushing the cache. A software flush would issue AVS-out and invalidation operations, followed by a cache flush. In contrast, a hardware flush could perform these operations in sequence as a single atomic action.

4. MODELING THE AVS-IN AND AVS-OUT COMMUNICATION COSTS

Compiler algorithms that identify ISEs use merit functions to estimate the speedup that can be attained by converting a software routine into an ISE [Pozzi et al. 2006; Verma et al. 2010]. If S is a subgraph in the compiler's intermediate representation representing an ISE candidate, the merit function is $M(S) = SW(S) - HW(S)$, where $SW(S)$ and $HW(S)$ are the respective latencies of executing the operations in software and

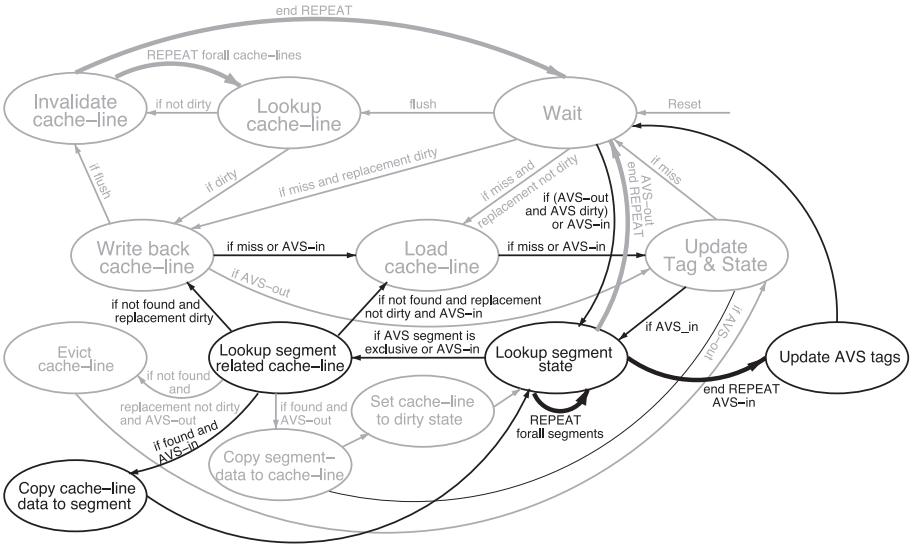


Fig. 15. Extension to the cache controller with support for AVS-in operations.

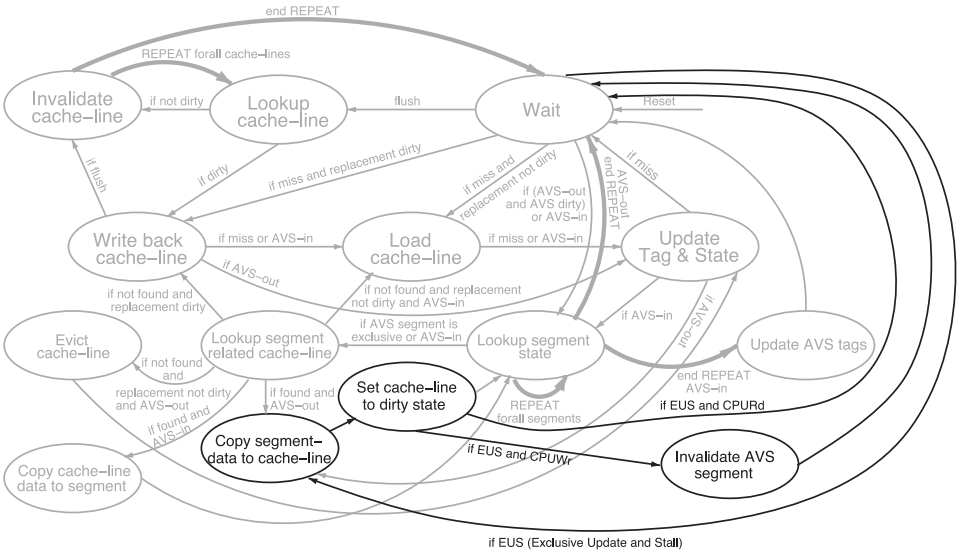


Fig. 16. Extension of the cache controller with the "Exclusive Update and Stall" (EUS) action.

using an ISE. For ISEs that do not access AVS, the merit function is straightforward to compute; however, AVS-aware ISE identification must account for the communication cost to accurately estimate the attainable speedup from the ISE [Biswas et al. 2007].

4.1. The AVS-in Communication Cost

AVS-in instructions copy a data structure from locations in the memory hierarchy to the AVS; three cases must be considered.

If the AVS contains the data structure in the valid or exclusive state, then the communication cost is zero, as the data is already in its proper location.

If the data cache contains a valid or dirty copy of the data, then it can be copied directly into the AVS at a cost of one cycle per cache line.

If neither the AVS nor data cache contains the data structure, then the processor must request it from main memory. The communication cost, $\lambda_{cl}(t)$, in this case, is

$$\lambda_{cl}(t) = D_{pb}\left(A_{bus}(t) + D_{bm}\left(O_m + \left\lceil \frac{S_{cl}}{N_m} \right\rceil\right)\right), \tag{6}$$

where $D_{pb} = f_{cpu}/f_{bus}$ is the ratio of the frequency of the bus with respect to the operating frequency of the processor, $A_{bus}(t)$ is the bus access latency at time t (which depends on the bus utilization and cannot be predicted statically), $D_{bm} = f_{bus}/f_{mem}$ is the ratio of the frequency of the bus (f_{bus}) to the frequency of the memory (f_{mem}), O_m is the memory burst overhead, S_{cl} is the cache line size, and N_m is the number of bits provided per memory cycle. To maintain inclusion, the data is written concurrently to the data cache and AVS.

We assume that the data path between the data cache and AVS is 32 bits wide and that each cache lines contains $S_{cl} = 32$ bytes (256 bits). The communication cost for one cache line is $S_{cl}/4$ cycles. Let d_k be the data structure to be loaded and $N_{se}(d_k)$ be the number of segments of d_k. At time t, let $a(d_k, t)$ be the number of segments of d_k in a valid or exclusive state in the AVS, $\beta(d_k, t)$ be the number of segments of d_k that do not reside in either the data cache or the AVS and must therefore be fetched from main memory, and $d(d_k, t)$ be the number of segments of d_k that are in the cache in a valid or dirty state; note that $N_{se}(d_k) = a(d_k, t) + \beta(d_k, t) + d(d_k, t)$. The cost of an AVS-in operation is

$$\lambda_{AVS-in}(d_k, t) = \beta(d_k, t)\,\lambda_{cl}(t) + \delta(d_k, t)\,\frac{S_{cl}}{4}. \tag{7}$$

Determining $a(d_k, t)$, $\beta(d_k, t)$, and $d(d_k, t)$ at compile time is infeasible in the general case. The most conservative approach is to assume that all data segments rely on memory, although static analysis or profiling techniques could be used to refine this assumption.

4.2. The AVS-Out Communication Cost

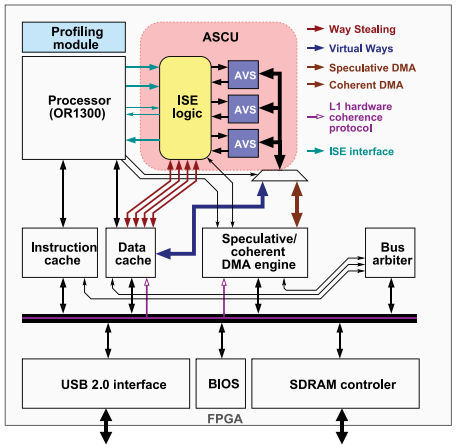
Virtual Ways uses a lazy write-back scheme, which copies data from the AVS upon request. In a single-processor system, only the processor can issue such a request. Hence, it suffices to copy the data from the AVS memory to the data cache, with an overhead of $S_{cl}/4$ per line. Modeling the communication cost for a multiprocessor system is more complicated and beyond the scope of this work.

5. EXPERIMENTAL SETUP

5.1. FPGA-Based Emulation Platform

We developed an in-house FPGA-based soft processor emulation platform, which includes a Xilinx Virtex II FPGA (model no. XC2V8000–5FF1517C), 32Mb of SDRAM, and a Cypress FX2 USB 2.0 connection mounted on a PCB of our design. An internally developed customizable soft processor runs on the platform. The processor and its memory subsystem completely fit onto the FPGA, so experimental results are obtained through *direct execution*; a small emulation layer compensates for variations between the processor and memory operating frequencies, which estimates system performance when the processor runs at higher frequencies than possible on the FPGA.

Figure 17 shows the system architecture. The processor (OR1300) is a 32-bit six-stage in-order RISC pipeline based on the OpenRISC instruction set [Lampret 2006]. The



(a)

Instruction Cache Configuration Parameters

Parameter:	Values:
Cache size	2 kB, 4 kB, 8 kB, or 16 kB
Cache mapping	Direct, 2-way, or 4-way set-associative
Replacement policy	FIFO, PLRU, or LRU

Data Cache Configuration Parameters

Parameter:	Values:
Cache size	2 kB, 4 kB, 8 kB, or 16 kB
Cache mapping	Direct, 2-way, or 4-way set-associative
Replacement policy	FIFO, PLRU, or LRU
Write policy	Write-through or write-back
Hardware coherence	Off, MSI, or MESI

SDRAM Overhead for Read/Write Bursts

SDRAM operation	RAS latency	CAS latency	Page close latency	Total burst overhead
Read	2 cycles	3 cycles	2 cycles	7 cycles
Write	2 cycles	0 cycles	2 cycles	4 cycles

(b)

Fig. 17. (a) Block diagram of the system architecture inside the FPGA. The ASCU is automatically generated for each benchmark. The other components are inserted based on a system template. (b) Instruction and data cache configuration parameters, and SDRAM latencies.

arithmetic units include a single-cycle multiplier, 64-bit pipelined MAC, and 32-cycle divider. The OR1300 includes an ISE interface identical to the Altera Nios II soft processor. We added dedicated ISEs to the OpenRISC ISA to support coherent data transfers to and from the AVS memory; our experimental evaluation compares Virtual Ways (this article) with Coherent and Speculative DMA [Kluter et al. 2008].

The OR1300's instruction and data caches are configurable through special purpose registers (SPRs). They both have a 32-byte line size and support software flushing. The data cache has no write buffers and supports up to three outstanding requests. The data cache stalls either due to data dependencies or more than three outstanding requests. The configurable caches support software-controlled design space enumeration; we can enumerate every combination of instruction and data cache parameters shown in Figure 17(b), and a benchmark application can then execute on each configuration. The caches are flushed between executions to ensure a cold start for each configuration.

The system includes an atomic bus that supports one transaction at a time. The bus arbiter implements a first-come, first-served policy and handles concurrent requests with a priority resolution scheme: the data cache and DMA engine have the highest and lowest priorities, respectively. The bus preemptively aborts transactions that remain inactive for a specified period of time to prevent deadlocks. The bus supports snoopy MSI and MESI hardware coherence protocols, which are used by Coherent and Speculative DMA.

The SDRAM controller allows the user to configure the platform to emulate the processor to memory distance, $D_{pm} = f_{cpu}/f_{mem}$, where f_{cpu} and f_{mem} are the processor and memory frequencies, respectively. The emulation layer assumes bursty bus transactions equal to the cache line size (32 bytes) and ignores SDRAM refresh overhead. A write/read burst requires $N_{write}/N_{read} = 12/15$ memory cycles. The emulation layer loads $D_{pm}N_{write}$ or $D_{pm}N_{read}$ into a counter, depending on the access. The counter is decremented each cycle; all future memory transactions stall until the counter reaches zero.

5.2. Energy Model

All energy results reported in this article are based on dynamic and leakage energy estimation using the CACTI 5.3 revision 174 [Shyamkumar et al. 2008] energy model for a 90nm technology node. We model instruction cache, data cache, and external memory energy consumption, but not bus energy. We do not model AVS memories because they are small and their energy consumption has a negligible impact on total system energy.

The profiling module counts events of interest using 64-bit counters, as shown in Figure 17(a). The energy model requires the following information: instruction cache fetches ($N_{I\$-fetch}$) and misses ($N_{I\$-miss}$); data cache reads ($N_{D\$-read}$), writes ($N_{D\$-write}$), and misses ($N_{D\$-miss}$); and the number of bus idle cycles (N_{bus}) and SDRAM accesses (N_{SDRAM}). The cache line size is 32 bytes. Each bus access reads or writes 4 bytes (one word); consequently eight reads/writes are required to load or store a complete cache line.

In the energy model, E_{read} and E_{write} denote the energy required for a cache or SDRAM read or write operation, respectively; these values are obtained from CACTI and depend on the technology and cache configuration parameters.

Instruction cache dynamic energy. Reads are straightforward; on a miss, a cache line (eight words) is written into the cache. The processor never writes directly to the instruction cache. The instruction cache dynamic energy is estimated as

$$E_{I\$} = N_{I\$-read}E_{read} + 8N_{I\$-miss}E_{write}. \tag{8}$$

Data cache dynamic energy. Coherent and Speculative DMA require a hardware coherence protocol, whereas Virtual Ways does not. Our model includes a parameter, α, which is set to *1* when the coherence protocol is enabled, and *0* otherwise. When coherence is enabled, E_{tag} is the energy consumed by a snoop lookup. The data cache dynamic energy is estimated as

$$E_{D\$} = N_{D\$-read}E_{read} + \left(N_{D\$-write} + 8N_{D\$-miss}\right)E_{write} + \alpha N_{SDRAM}E_{tag}. \tag{9}$$

SDRAM dynamic energy. The SDRAM dynamic energy is estimated as follows; note that E_{read} and E_{write} here are for SDRAM rather than SRAM technology:

$$E_{SDRAM} = 8N_{SDRAM}\frac{E_{read} + E_{write}}{2} = 4N_{SDRAM}\left(E_{read} + E_{write}\right) \tag{10}$$

Leakage energy. Let $P_{leak,I\$}$, $P_{leak,D\$}$, and $P_{leak,SDRAM}$ be the leakage power consumed by the caches and SDRAM, respectively, as provided by CACTI 5.3, and $T_{benchmark}$ be the execution time of each benchmark. The leakage energy is estimated as

$$E_{leak} = T_{benchmark}\left(P_{leak,I\$} + P_{leak,D\$} + P_{leak,SDRAM}\right). \tag{11}$$

5.3. Compiler

We used an internally developed compiler, Clarity, to perform ISE identification and ASCU generation [Pozzi et al. 2006] with extensions to integrate AVS memories [Biswas et al. 2007]. Clarity uses statistical profiling to select "hot" program regions to search for ISEs. We modified the cost function used by Clarity's ISE identification algorithm to account for Coherent and Speculative DMA [Kluter 2010], and Virtual Ways; this ensures that the algorithm identifies appropriate ISEs for each coherence mechanism.

After choosing ISEs, Clarity generates VHDL code for the ASCU for the target platform (the Nios II ISE interface, as supported by the OR1300). Clarity does not support automatic AVS generation for Xilinx FPGAs or the architectural modifications to support Coherent and Speculative DMA [Kluter et al. 2008] or Virtual Ways; we implemented all of these features manually using VHDL.

Table I. Reference Configurations for the Different Benchmarks and Datasets

EEMBC benchmark	Dataset	Instruction cache		Data cache	
		Size	Mapping	Size	Mapping
CJPEGV2	1	8kB	4 WSA	16kB	4 WSA
	2,3,4,5,6,7	4kB	4 WSA	16kB	4 WSA
MPEG2 enc.	1,2,3,4	8kB	4 WSA	8kB	4 WSA
	5	4kB	4 WSA	4kB	4 WSA
MPEG2 dec.	1	16kB	4 WSA	16kB	4 WSA
	2,3	8kB	4 WSA	8kB	4 WSA
	4,5	16kB	2 WSA	16kB	4 WSA
AES	1	4kB	DM	2kB	2 WSA

DM, direct-mapped; k WSA, k-way set-associative.

Clarity outputs updated C-language source code implementation of the benchmark, which uses macros to represent ISE invocations. We manually inserted AVS-related data transfer instructions into the program. We cross-compiled each application using a gcc 3.4.4 toolchain based on binutils 2.16.1 targeting the OR1300. We extended the cross-compiler to support AVS-related data transfers, cache flushing, and so forth, and ISEs. The host PC loads the cross-compiled program onto the FPGA board and then initiates execution.

5.4. Benchmarks and Experimental Flow

We used the EEMBC DENBench applications [Halfhill 2000] for experimental evaluation; DENBench consists of complete applications, not kernels, which is why we chose it. We used the test harness provided by EEMBC for all experiments, as the OR1300 does not presently support an operating system. Each application was evaluated using input datasets provided by EEMBC. We ran the following test procedure to validate each custom processor generated by our system.

Correctness check. All results reported here pass the EEMBC test harness CRC check.

Noncoherent AVS architectures. We implemented a noncoherent AVS-enhanced ISE scheme [Biswas et al. 2007]. All implementations either failed the CRC check or completed incorrectly; in the latter case, coherence errors changed the control flow and caused the program to terminate early. We do not report these results.

Reference configuration. We enumerated the instruction and data cache configurations for each benchmark, running in software, without ISEs. The configuration that yields the best performance/energy trade-off is selected as a reference configuration (Table I) and used for the initial set of experiments. The reference configuration is optimal for a software implementation but may or may not be good for (AVS-enhanced) ISEs.

Architecture exploration. We enumerated the instruction and data cache configurations for all ISE-enhanced architectures (both non–AVS enhanced and AVS enhanced) and executed each benchmark. Performance (speedup) and energy consumption are reported. The results are normalized to the reference configuration running without ISEs.

6. EXPERIMENTAL RESULTS

We compare Virtual Ways with ISEs that do not include AVS generated by a branch-and-bound search [Pozzi et al. 2006], where each ISE has at most four inputs and two outputs. We also compare with Speculative DMA [Kluter et al. 2008], which uses

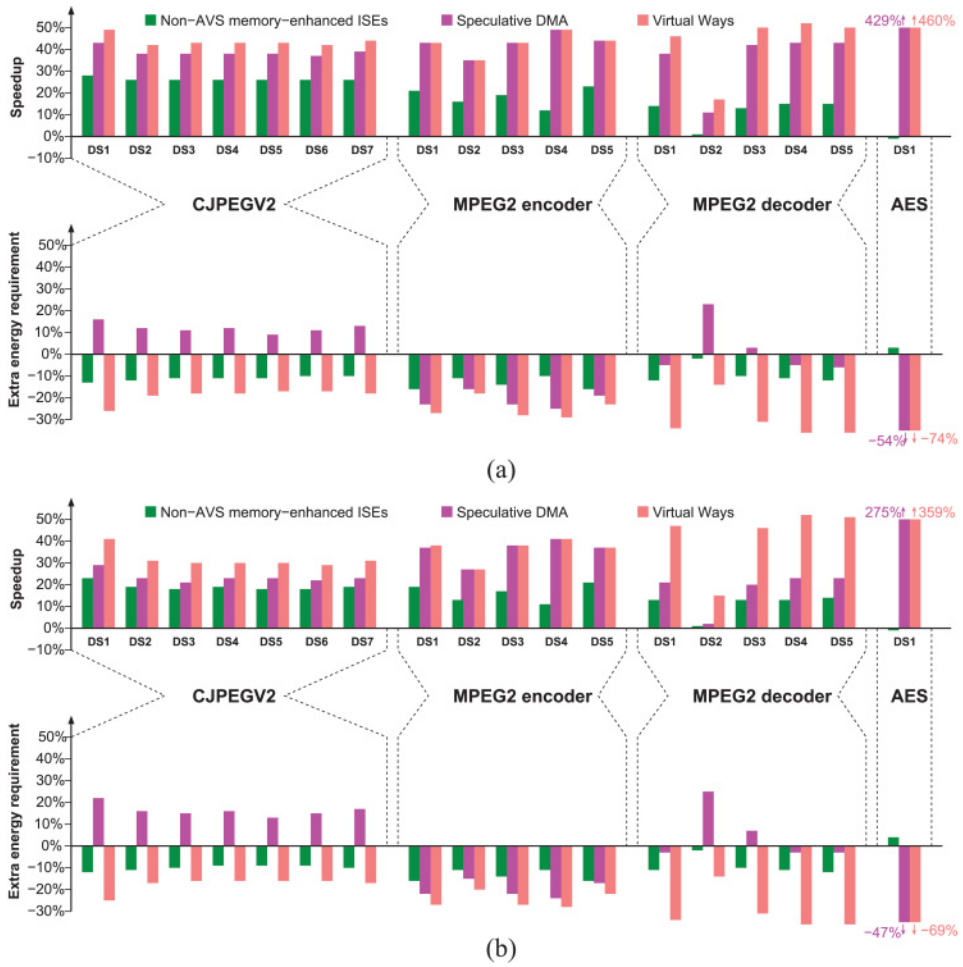


Fig. 18. Experimental results for the three ISE-enhanced architectures relative non–ISE-enhanced execution on the reference configuration and Speculative DMA [Kluter et al. 2008]. DSx, dataset x. (a) Results for a system where both the processor and external memory run at 100MHz. (b) Results for a system where the processor runs at 900MHz and external memory runs at 100MHz.

DMA transfers to move to/from the AVS, and employs a MESI states snoopy hardware coherence protocol.

6.1. Comparison with Non–AVS-Enhanced ISEs and Speculative DMA

Table I lists the reference configuration for each benchmark and dataset. All experiments summarized here are performed using the reference configuration.

Figure 18(a) shows the experimental comparison between the different ISE-enhanced architectures for a system where both the processor and the external memory run at 100MHz. Figure 18(b) repeats the experiments with the processor running at 900MHz.

In Figure 18(a), Virtual Ways achieves higher performance and consumes less energy than Speculative DMA. Speculative DMA suffers from excessive data transfers between the data cache and AVS memories (via main memory), which we call the *ping-pong effect*. These data transfers consume time and energy, and also induce coherence protocol actions. In contrast, Virtual Ways transfers data directly between the

data cache and AVS memory, improving performance and significantly reducing energy consumption.

The ping-pong effect actually occurs because the data cache has good temporal and spatial locality; if locality is poor, data thrashing in an application could prevent the data cache from maintaining locality. In this case, the communication cost incurred by Virtual Ways would be equal to the cost incurred by Speculative DMA. With or without ISEs, data must be loaded from main memory, either into the cache (software or non–AVS-enhanced ISEs) or into the AVS (AVS-enhanced ISEs). In particular, the MPEG2 encoder benchmark experiences thrashing, as its data locality tends to be poor. Figure 18 confirms this observation, as the performance and energy consumption of Speculative DMA and Virtual Ways are similar, indicating that Virtual Ways does not benefit from the more efficient data transfer path between the data cache and AVS.

Figure 18(b) repeats the experiments for a system with a 9 times larger processor-to-memory distance than in Figure 18(a). With a greater processor-to-memory distance, the impact of loads and stores on performance is much greater. Both Speculative DMA and Virtual Ways achieve more modest speedups than in Figure 18(a); however, the degradation in speedup for Speculative DMA is far worse than for Virtual Ways. This indicates that Speculative DMA is more sensitive to variations in the processor-to-memory distance than Virtual Ways; however, there were two exceptions, which we discuss in detail next.

CJPEGV2. Across the different datasets, Figure 18 shows that increasing processor-to-memory distance by a factor of nine reduces the speedup attained by Virtual Ways by around 10%, whereas the reduction for the non–AVS-memory-enhanced system is around 6%. This behavior is due to a restriction caused by an implementation issue.

Virtual Ways extends the cache state machine to handle AVS-in and AVS-out actions. These modifications prevent the data cache from servicing load or store requests issued by the processor during AVS-in and AVS-out actions; similarly, AVS-in and AVS-out actions are stalled during cache misses. Both situations prevent the overlap of communication with computation because the processor stalls.

Speculative DMA has separate state machines for the data cache and DMA engine; thus, it does not suffer from these restrictions. That being said, these restrictions are due to an implementation choice and are not germane to the general concept of Virtual Ways.

AES. AES has relatively small data and instruction footprints, as confirmed by the small instruction and data cache sizes reported as reference configurations in Table I. The ISE identification algorithm detects three read-only arrays that are moved by the compiler to AVS memories, removing them from the runtime data footprint. Moving them into AVS significantly reduces the amount of data stored in the cache and memory subsystem.

AES contains many bitwise logical operations that operate on eight-bit data chunks. These operations benefit significantly from ISEs, because they require one cycle in software, but negligibly the delays of ISEs that include them; AES also has a large amount of data parallelism. This limits the performance of software and non–AVS-enhanced ISEs where it is necessary to load scalar data from arrays into the register file. With AVS-enhanced ISEs, more parallelism is available, and multiported AVS can provide high data bandwidth into the ASCU; this also results in larger ISEs [Verma et al. 2010]. Given that AES begins with a small code size, the replacement of instruction sequences with ISEs reduces the instruction footprint more than for other applications.

As a consequence, the reference configuration is a poor implementation choice for AES with AVS-enhanced ISEs; a secondary consequence is that the application is so

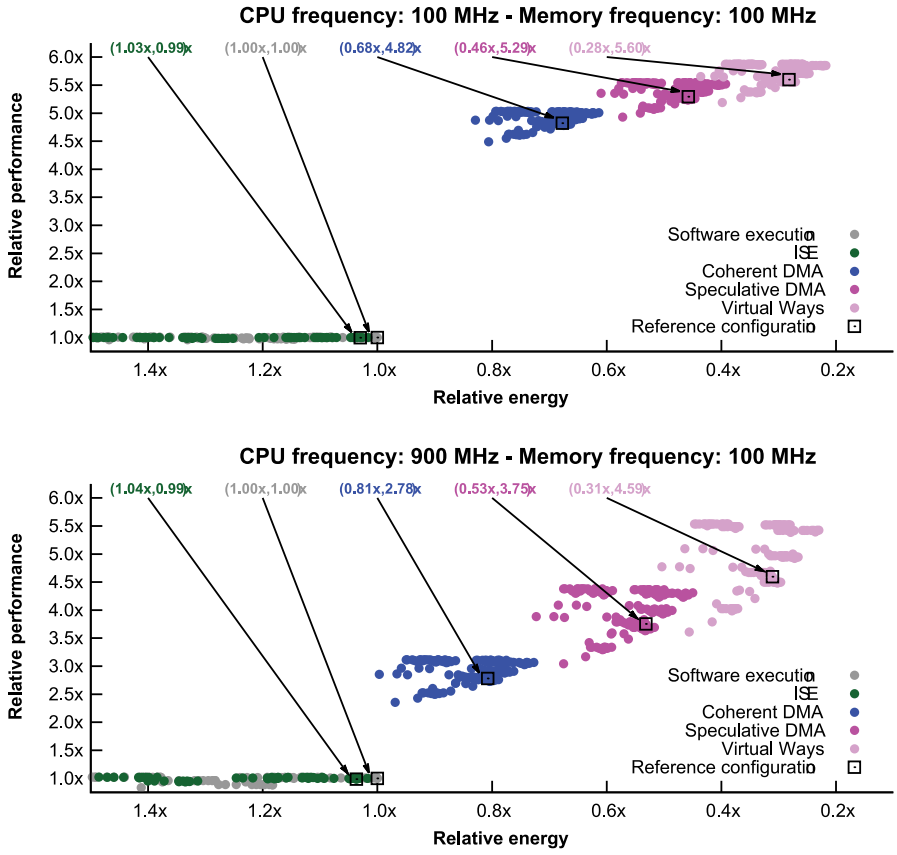


Fig. 19. Design space exploration of the AES benchmark.

small in terms of both code and data size that the processor-to-memory distance has a minimal impact on relative performance and energy consumption for Speculative DMA.

6.2. Design Space Exploration: AES

Figure 19 shows the results of an architectural design space exploration for AES considering software execution, execution with non–AVS-enhanced ISEs, as well as AVS-enhanced ISEs using Coherent DMA, Speculative DMA, and Virtual Ways for coherence. Coherent and Speculative DMA use DMA transfers to move data between main memory and the AVS, and employ a hardware coherence protocol. Speculative DMA includes some additional AVS state information, which enables it to suppress some unnecessary DMA transfers, although at the expense of a much more complicated hardware implementation than Coherent DMA. The design space exploration considers processor speeds of 100MHz and 900MHz, whereas main memory runs at 100MHz. We have performed similar design space explorations for the other benchmarks and datasets [Kluter 2010]; they have been omitted from this article due to limited space.

Referring back to the preceding subsection, moving several arrays into the AVS and converting a sizeable percentage of the application's computation into ISEs significantly affects the code and data size; thus, the reference configuration, which performed the best for software execution, is not the best choice for AVS-enhanced ISEs. Using the most favorable configurations, Virtual Ways achieves a 5.9 times

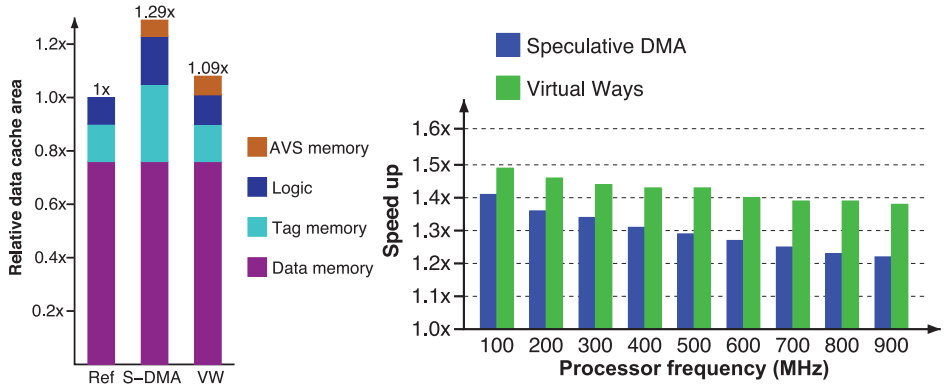


Fig. 20. (a) Area overhead of Speculative DMA and Virtual Ways compared to a standard data cache. (b) The impact of processor frequency on the speedup attainable by Speculative DMA and Virtual Ways. Both experiments are performed using CJPEGV2.

speedup at a processor-to-memory distance of 1 and a 5.7 times speedup at a processor-to-memory distance of 9. In contrast, Speculative DMA achieves a 5.6 times speedup at a processor-to-memory distance of 1 and a 4.5 times speedup at a processor-to-memory distance of 9 for the same configuration.

6.3. Implementation Overhead and the Impact of Processor-to-Memory Distance

Tightly coupling AVS memories to the hardware cache guarantees coherence and consistence for the complete system. This coupling requires nontrivial modifications to the cache's hit detection circuitry and state machine without influencing its critical path. Direct communication between the AVS memories and data cache reduce the impact of the processor-to-memory distance on the speedup attained by AVS-enhanced ISEs.

Figure 20(a) compares the area overhead of Virtual Ways and Speculative DMA specialized for the CJPEGV2 benchmark with AVS memories, along with a traditional data cache; we implemented all three schemes in a 90nm standard-cell technology. Figure 20(a) shows that Virtual Ways increases the area by 9%, whereas Speculative DMA, which includes a hardware coherence protocol and controller, increases the area by 29%.

The AVS memory reported in Figure 20(a) is actually a small register file with 64 eight-bit registers and eight read and eight write ports. In practice, the area overhead of the AVS memory will vary from benchmark to benchmark and may also depend on the compiler algorithm that selects AVS-enhanced ISEs.

Figure 20(b) measures the impact of processor frequency on the speedup attainable by ISEs for the same benchmark. Increasing the processor frequency increases the impact of memory accesses on overall performance; since ISEs speed up computation, but not memory, it is therefore expected that their performance impact will degrade as processor frequency increases. Figure 20(b) shows that the speedup attainable by AVS-enhanced ISEs using Speculative DMA degrades at a faster rate in comparison to Virtual Ways. The reason is that Speculative DMA depends on the processor-to-memory distance, as data must stream directly from main memory into the AVS via DMA; in contrast, Virtual Ways loads data into the AVS via the cache, rendering the performance impact of the AVS-in operation independent from the processor-to-memory distance.

Even without an ISE, the processor must load the data into the L1 cache to read or write it, so the impact of the memory transfer would be incurred irrespective of Virtual Ways. The situation is different with Speculative DMA because there is no direct path

from the L1 cache to the AVS memories. If the data is first loaded into the cache by the processor and then modified, then the hardware coherence mechanism would need to evict it to update the copy in main memory and then transfer the valid copy into the AVS via DMA; both of these operations are sensitive to the processor-to-memory distance.

7. RELATED WORK

7.1. AVS Memory Architecture

Early attempts to integrate AVS memories into ISEs did not account for the potential memory coherence and consistence problems that could result [Biswas et al. 2007]. The example shown in Section 2 of this article demonstrates that these problems are real and could potentially impact correctness of such a system. Coherent and Speculative DMA [Kluter et al. 2008] employ a hardware coherence protocol to fix the memory coherence and consistence problems. Speculative DMA is an enhancement to Coherent DMA that can reduce the number of DMA transfers required at runtime, achieving higher performance and lower energy consumption as a result. The experiments reported in Section 6 compare Virtual Ways to Coherent and Speculative DMA, and they demonstrate better performance, lower energy consumption, lower area overhead, and lower sensitivity to the processor-to-memory distance. The one drawback of Virtual Ways compared to Coherent and Speculative DMA is that it requires a nonstandard cache controller; in contrast, Coherent and Speculative DMA can be implemented using standard memory IP blocks.

One alternative is to store all data structures accessed by AVS-enhanced ISEs in an uncacheable region of memory [Prakash et al. 2012]. This eliminates the memory coherence and consistence problems, as these data structures are never loaded into the cache hierarchy. On the other hand, if the processor ever needs to access these data structures outside of an ISE, then each access must go directly to memory, thereby sacrificing the benefits of caching. In contrast, Virtual Ways provides AVS-enhanced ISEs and caching without the overhead of a hardware coherence protocol.

Way Stealing [Kluter et al. 2009] employs a modified data cache that allows the processor and ASCU to directly access each way as an AVS memory. The compiler inserts prefetch instructions into the program code to move data directly into the desired way prior to an ISE invocation and lock instructions to prevent its dynamic eviction. The ASCU then reads and writes the data in the ways of the processor as needed. Like Virtual Ways, and unlike Coherent and Speculative DMA, Way Stealing does not use DMA transfer instructions to move data between main memory and the AVS, and does not require a hardware coherence protocol; since the data cache and AVS memories are one and the same, the memory coherence and consistence problems are implicitly eliminated.

Virtual Ways differs from Way Stealing in two key respects. First, the number of AVS memories under Way Stealing is limited by the associativity of the data cache. Second, each AVS memory under Way Stealing has exactly one read and one write port, which can actually be quite restrictive. For example, the AVS memory used for CJPEGV2, as mentioned in Section 6.3, has eight read and eight write ports. Compared to Virtual Ways, Way Stealing offers less memory access concurrency, thereby limiting the performance benefits attainable by AVS-enhanced ISEs.

7.2. Identifying ISEs without AVS

Most of the work on customizable processors has focused on compiler algorithms to automatically identify and synthesize ISEs [Pozzi et al. 2006; Verma et al. 2010; Atasu et al. 2012]. Any of these algorithms can be extended to identify AVS-enhanced ISEs [Biswas et al. 2007] and can integrate the cost model described in this article into the

merit function that guides the search. Cost models for Coherent and Speculative DMA and Way Stealing have also been developed [Kluter 2010].

ISEs that do not access AVS communicate directly with the processor's register file, which is a significant I/O bottleneck; inclusion of AVS memories, as summarized in the preceding subsection, overcomes this limitation. Many otherwise ideal ISE candidates have large amounts of internal parallelism, which is lost due to I/O serialization across multiple clock cycles [Pozzi and Ienne 2005]. To improve I/O bandwidth, it is possible to access AVS memories and the register file at the same time. Data moved into AVS memories does not conflict with other data in the processor's register file; in addition, the process of loading data into the cache, then into a register, then into the ISE, then back into a register, and storing it back into the cache is more efficient with AVS memories.

7.3. Increasing Data Bandwidth between the Processor an ASCU without AVS

Several architectural mechanisms have been introduced to increase data bandwidth between the processor pipeline and the ASCU. These mechanisms have only been investigated for use with non–AVS-enhanced ISEs; in principle, they could be used in conjunction with AVS-enhanced ISEs, as discussed in Subsection 7.1. All of these techniques increase bandwidth from the processor to the ASCU, but they do not provide any additional bandwidth to transfer data from the ASCU back to the processor.

Cong et al. [2005] add shadow registers to the ASCU, which are external to the processor's register file. The bitwidth of the processor's ISA is extended to allow values computed by the ALU to be written to the shadow registers. The ASCU can read the shadow registers and register file in parallel—and, in principle, could read/write an AVS memory at the same time as well. The drawback of this mechanism is that the processor must either (1) extend the bitwidth of all instructions, knowing that only a handful will actually write data to shadow registers, or (2) implement a variable-bitwidth instruction encoding scheme so that only instructions that write data to shadow registers use the extra bits; however, this would significantly complicate the pipeline's fetch-and-decode logic. In contrast, Virtual Ways requires an AVS-in processor instruction but does not add any additional bits to the instruction work.

Jayaseelan et al. [2006] allow the ASCU to read data from several pipeline registers at once, along with the register file, exploiting the data-forwarding path in a RISC pipeline. A five-stage pipeline allows two additional inputs to be read; however, exploiting this mechanism imposes new instruction scheduling constraints, as the instructions that compute values that will be read by the ISE must be scheduled in the two time slots immediately preceding it. To use Virtual Ways, a compiler must be able to insert AVS-in instructions at appropriate program locations; since these instructions are blocking, no other complications akin to scheduling constraints are otherwise imposed.

Karuri et al. [2007] proposed a clustered register file, wherein the ASCU reads data concurrently from several clusters. This complicates the compiler's register allocator. If the register file contains k clusters, and the ASCU reads k data values concurrently, then all of the values must be allocated to different clusters, or some values must be replicated across clusters, which increases register pressure. The compiler must insert cluster-to-cluster data transfers to position the data correctly before the ISE can execute. In contrast, a compiler that supports Way Stealing requires no modifications to the register allocator.

7.4. Memory Coherence and Consistence

The problems of memory coherence and consistence have traditionally arisen in the context of multiprocessor systems; numerous papers have been written on this topic,

and we acknowledge the work of McFarling [1992] and Wilton and Jouppi [1994] as being most influential to this work, as they looked into variations of traditional inclusion. Many other papers have also looked at exclusive [Barosso et al. 2000] and noninclusive [Jaleel et al. 2010] cache hierarchies; however, they focus on server-class and general-purpose processors with considerably different workloads than this particular study.

8. CONCLUSION

This article has shown that Virtual Ways preserves correct execution of AVS-enhanced ISEs by ensuring coherence and consistence of the AVS with respect to the data cache. Compared to Coherent and Speculative DMA, Virtual Ways does not require a hardware coherence protocol, it offers higher performance, consumes less energy, requires less area, and is more robust to changes in the processor-to-memory distance. The limitation of Virtual Ways is the changes required to the data cache; however, our experimental setup has shown that a generic interface can be provided. Generation of this interface is a one-time effort and enables automatic algorithms to interface the AVS memories in a well-defined way. The reduced communication cost improves the ability of ISE identification algorithms to select high AVS-enhanced ISEs, unlike Coherent and Speculative DMA where the cost is higher. Altogether, our results have show that Virtual Ways is a better solution to the coherence problem than Coherent and Speculative DMA.

It is an open question as to whether or not the techniques presented in this article can generalize to related problems in coprocessor memory management. One possibility is to apply Virtual Ways to hybrid cache-scratchpad memories in general-purpose, rather than application-specific, processors. Another is to study Virtual Ways in multicore SoCs, with/without hardware coherence protocols. Future work will investigate these issues.

REFERENCES

K. Atasu, W. Luk, O. Mencer, C. C. Özturan, and G. Dündar. 2012. FISH: Fast instruction synthesis for custom processors. *IEEE Transactions on Very Large Scale Integration (TVLSI) Systems* 20, 1, 52–65.

R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel. 2002. Scratchpad memory: A design alternative for cache on-chip memory in embedded systems. In *Proceedings of the International Symposium on Hardware/Software Codesign*. 73–78.

L. A. Barroso, K. Gharachorloo, R. Mcnamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. 2000. Piranha: A scalable architecture based on single-chip multiprocessing. In *Proceedings of the International Symposium on Computer Architecture*. 282–293.

P. Biswas, N. D. Dutt, L. Pozzi, and P. Ienne. 2007. Introduction of architecturally visible storage in instruction set extensions. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 26, 3, 435–446.

J. Cong, Y. Fan, G. Han, A. Jagannathan, G. Reinman, and Z. Zhang. 2005. Instruction set extension with shadow registers for configurable processors. In *Proceedings of the International Sympsosium on Field Programmable Gate Arrays*. 99–106.

R. E. Gonzalez. 2000. Xtensa: A configurable and extensible processor. *IEEE Micro* 20, 2, 60–70.

R. E. Gonzalez. 2006. A software-configurable processor architecture. *IEEE Micro* 26, 5, 42–51.

T. R. Halfhill. 2000. EEMBC releases first benchmarks. *Microprocessor Report*, May 1, 2000.

T. R. Halfhill. 2003. Tensilica's software makes hardware. *Microprocessor Report*, June 23, 2003.

R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz. 2011. Understanding sources of inefficiency in general-purpose chips. *Communication of the ACM* 54, 85–93.

A. Jaleel, E. Borch, M. Bhandaru, S. C. Steely Jr., and J. S. Emer. 2010. Achieving non-inclusive cache performance with inclusive caches: Temporal locality aware (TLA) cache management policies. In *Proceedings of the International Symposium on Microarchitecture*. 151–162.

R. Jayaseelan, H. Liu, and T. Mitra. 2006. Exploiting forwarding to improve data bandwidth of instruction-set extensions. In *Proceedings of the Design Automation Conference*. 43–48.

K. Karuri, A. Chattopadhyay, M. Hohenauer, R. Leupers, G. Ascheid, and H. Meyr. 2007. Increasing data-bandwidth to instruction-set extensions through register clusters. In *Proceedings of the International Conference on Computer-Aided Design*. 166–171.

T. Kluter. 2010. Architectural support for coherent architecturally visible storage in instruction set extensions. Ph.D. Dissertation. EPFL.

T. Kluter, P. Brisk, P. Ienne, and E. Charbon. 2008. Speculative DMA for architecturally visible storage in instruction set extensions. In *Proceedings of the International Conference on Hardware-Software Codesign and System Synthesis*. 243–248.

T. Kluter, P. Brisk, P. Ienne, and E. Charbon. 2009. Way stealing: cache-assisted automatic instruction set extensions. In *Proceedings of the Design Automation Conference*. 31–36.

T. Kluter, S. Burri, P. Brisk, E. Charbon, and P. Ienne. 2010. Virtual ways: efficient coherence for architecturally visible storage in automatic instruction set extensions. In *Proceedings of the International Conference on High-Performance Embedded Architectures and Compilers*. 126–140.

D. Lampret. 2006. OpenRISC 1000 Architecture Manual. Available at http://www.opencores.org/.

S. Mcfarling. 1992. Cache replacement with dynamic exclusion. In *Proceedings of the International Symposium on Computer Architecture*. 191–200.

L. Pozzi, K. Atasu, and P. Ienne. 2006. Exact and approximate algorithms for the extension of embedded processor instruction sets. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 25, 7, 1209–1229.

L. Pozzi and P. Ienne. 2005. Exploiting pipelining to relax register-file port constraints of instruction-set extensions. In *Proceedings of the International Conference on Compilers Architecture, and Synthesis for Embedded Systems*. 2–10.

A. Prakash, C. T. Clarke, and T. Srikanthan. 2012. Custom instructions with local memory elements without expensive DMA transfers. In *Proceedings of the International Conference on Field Programmable Logic and Applications*. 647–650.

T. Shyamkumar, M. Naveen, J. H. Ahn, and N. P. Jouppi. 2008. CACTI 5.1. Technical Report HPL-2008-20, HP Labs. Available online: http://www.hpl.hp.com/techreports/2008/HPL-2008-20.html.

A. K. Verma, P. Brisk, and P. Ienne. 2010. Fast, nearly optimal ISE identification with I/O serialization through maximal clique enumeration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 29, 3, 341–354.

S. J. E. Wilton, and N. P. Jouppi. 1994. Tradeoffs in two-level on-chip caching. In *Proceedings of the International Symposium on Computer Architecture*. 34–45.

Received January 2012; revised November 2013; accepted January 2014