Virtual Ways: Efficient Coherence for Architecturally Visible Storage in Automatic Instruction Set Extensions

Theo Kluter[1,5], Samuel Burri[2], Philip Brisk[4], Edoardo Charbon[2,3], and Paolo Ienne[1]

[1] Ecole Polytechnique Fédérale de Lausanne (EPFL), School of Computer and Communication Sciences, CH-1015 Lausanne, Switzerland
paolo.ienne@epfl.ch
[2] Ecole Polytechnique Fédérale de Lausanne (EPFL), School of Engineering, CH-1015 Lausanne, Switzerland
samuel.burri@epfl.ch
[3] Delft University of Technology, Circuits and Systems Group, NL-2600 AA Delft, The Netherlands
edoardo.charbon@epfl.ch
[4] University of California, Riverside, Department of Computer Science and Engineering, Riverside, CA 92521, USA
philip@cs.ucr.edu
[5] Bern University of Applied Sciences, EKT, Microlab, Quellgasse 21, CH-2501 Biel/Bienne, Switzerland
theo.kluter@bfh.ch

Abstract. Customizable processors augmented with application-specific *Instruction Set Extensions (ISEs)* have begun to gain traction in recent years. The most effective ISEs include *Architecturally Visible Storage (AVS)*, compiler-controlled memories accessible exclusively to the ISEs. Unfortunately, the usage of AVS memories creates a coherence problem with the data cache. A multiprocessor coherence protocol can solve the problem, however, this is an expensive solution when applied in a uniprocessor context. Instead, we can solve the problem by modifying the cache controller so that the AVS memories function as extra ways of the cache with respect to coherence, but are *not* generally accessible as extra ways for use under normal software execution. This solution, which we call *Virtual Ways* is less costly than a hardware coherence protocol, and eliminate coherence messages from the system bus, which improves energy consumption. Moreover, eliminating these messages makes Virtual Ways significantly more robust to performance degradation when there is a significant disparity in clock frequency between the processor and main memory.

Keywords: Application-Specific Processors, Memory Coherence, Instruction Set Extensions, Virtual Ways.

1 Introduction

Extensible processors are a cost-effective platform that can help embedded system designers meet their targets for performance and energy efficiency. These

Y.N. Patt et al. (Eds.): HiPEAC 2010, LNCS 5952, pp. 126–140, 2010.

processors are augmented with application-specific custom *instruction set extensions (ISEs)* that improve performance and energy efficiency for critical loops in embedded applications. ISEs can be identified automatically [11,4], and a system designer must only verify the ISEs and their interface to the processor, as the processor itself has been pre-verified by the vendor. Although extensible processors cannot compete with *application-specific integrated circuits (ASICs)* in terms of performance and energy efficiency, they offer an economic advantage through a simplified design and verification process and a reduced time-to-market.

To increase performance, ISEs have been augmented with *Architecturally Visible Storage (AVS)*, which can be registers or compiler-controlled memories [3]. AVS memories are distinct from the cache hierarchy, and *Directed Memory Access (DMA)* transfers move data between main memory and the AVS, bypassing the caches, which creates a coherence problem. Kluter et al. [9] solved the coherence problem using a snoopy hardware coherence protocol, which was designed for use in multiprocessor systems. This solution has two drawbacks: area overhead, and performance degradation due to coherence messages on the system bus competing with off-chip memory accesses.

Virtual Ways, presented here, is a scheme by which the cache controller is modified to ensure coherence between the data cache and the AVS memory. Under this scheme, the data cache and AVS memory share a common interface, and prefetch instructions are used in lieu of DMA transfers. A relaxed form of inclusion between the data cache and AVS memory provides coherence: the data in the AVS memory is always a subset of the data in the cache, but writes in the AVS memory are not automatically written through to the cache, but writes to the AVS memory (cache) and not written through to the cache (AVS memory). The cache controller, therefore, evolves into a low-cost hardware coherence protocol for this specific case.

Virtual Ways and Speculative DMA are compared using a standard cell design flow to estimate the area overhead of the memory subsystem. For JPEG compression, in which the AVS memory is a 64-entry register file containing 8-bit registers, and 8 read and 8 write ports, the area overhead of Speculative DMA was 1.29x, due to the cost of the coherence protocol, including the AVS memory, while the area overhead of Virtual Ways was 1.09x, due mostly to the AVS memory and a slightly larger data cache state machine.

Virtual Ways and Speculative DMA are compared using an FPGA-based soft processor emulation system to measure task latency and memory system energy consumption. The experiments include a detailed case study of JPEG compression, and an evaluation of four EEMBC consumer V2 benchmarks: CJPEGV2 (compression), MPEG encoding and decoding, and AES. The most significant result is that the speedups achieved by Speculative DMA degrade significantly as the frequency of the processor increases while the frequency of off-chip memory remains constant, whereas, Virtual Ways does not suffer from any noticeable performance degradation. For CJPEGV2 and MPEG encoding and decoding, Virtual Ways achieved a higher speedup and reduced energy consumption compared

to Speculative DMA, while the results for both metrics were equal for AES, due to the fact that all data structures in the AVS memories are read-only.

We performed a detailed analysis and case study of an internally-modified version of JPEG compression that only compresses one color component; we call this version "JPEG" for simplicity. The analysis of JPEG includes a kernel-by-kernel breakdown of the task latency and energy consumption of the two techniques, and include a design space exploration in which the size and associativity of the instruction and data caches are varied. In the former study, Virtual Ways achieves a significant energy reduction in the Quantisation kernel, while achieving comparable task latencies across all kernels. In the latter study, Virtual Ways reduces task latency and energy consumption, compared to Speculative DMA, for each configuration. Looking across configurations, Virtual Ways generally achieves the best results; however, a handful of the best performing configurations of Speculative DMA do achieve better task latency and/or energy consumption than the worst performing configurations of Speculative DMA; the overall trend, however, favors Virtual Ways.

The remainder of the paper is organized as follows: Section 2 details related work in the domain. Section 3 introduces Virtual Ways and describes their implementation in an extensible processor featuring AVS-enhanced ISEs. Section 4 describes an FPGA-based soft processor emulation system that we use for our performance evaluation, and Section 5 presents an in-depth case study using JPEG compression, followed by a more general study using EEMBC consumer V2 benchmarks. Section 6 concludes the paper.

2 Related Work

In early work on ISEs, the processor's register file was the I/O interface [4,11]. A typical register file has two read ports and one write port, which limit the size of each ISE and the attainable speedup. Multi-cycle ISEs [1,10,12,14,15] overlap computation with I/O operations; however, the I/O interface remained a bottleneck. Several microarchitectural modifications have successfully improved input bandwidth, including *shadow registers* [5], register file clustering [7], and utilizing the pipeline forwarding logic [6]. Although generally effective, these techniques do not improve output bandwidth, and data is transferred to the ISE logic on the granularity of scalar variables; they do not support bulk transfers of arrays.

Biswas et al. [2] introduced *architecturally visible storage (AVS)*, which was limited to small ROMs that hold constant values, and state registers. In a subsequent work, they augmented their ISEs with small compiler-controlled memories that hold arrays. DMA transfers move data into and out of the AVS memories bypassing the cache [3]. This solution is similar to *scratchpad memories* [13], which are also placed under compiler control. Scratchpad memories have been proposed as an *alternative* to caches for embedded systems, because eliminating the tag array reduces per-access energy consumption, and deterministic hit/miss behavior improves predictability of worst-case execution time.

AVS memories, in contrast, co-exist with caches, rather than replacing them. As observed by Kluter et al. [9], this leads to a coherence problem, as the DMA transfers between main memory and AVS memories bypass the cache hierarchy, and no mechanism exists to ensure coherence between the AVS memory and the data cache. They corrected the problem using a hardware coherence protocol; however, the area overhead of the DMA controller and the coherence protocol were significant. Additionally, coherence messages transmitted on the bus consume energy, and may increase the latency of accesses to off-chip memory.

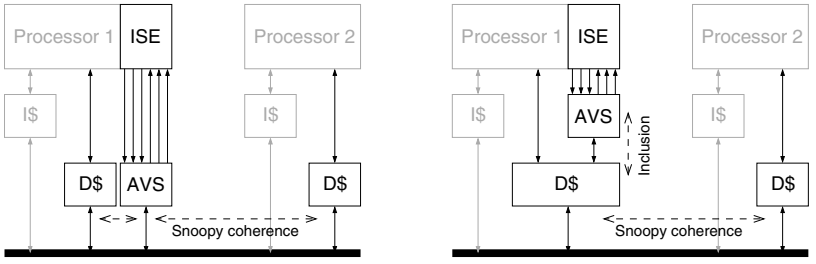
Under this scheme, the compiler inserts *Speculative DMA* transfer instructions to move data from main memory into the AVS memory before an ISE that accesses the latter may execute. Each line of the AVS memory is augmented with *valid* and *dirty* bits, similar in principle to a cache, but without the tag arrays. These bits are required to integrate the AVS memory into the hardware coherence protocol; additionally, the valid bits facilitate the *speculative* aspect of DMA, by suppressing transfers when data in the AVS memory is up-to-date.

The data cache and AVS memory snoop bus transactions. A write to data in the AVS (cache) invalidates a copy of the data that may exist in the cache (AVS). If the processor reads invalid data from the cache, the coherence protocol retrieves the valid copy from the AVS memory. Speculative DMA transfers from main memory into the AVS memory request the write-back o a dirty copy of the data that may exist in the cache. When a DMA transfer overwrites valid and dirty data in the AVS memory, the coherence protocol ensures that the valid data is written back to main memory. When a DMA transfer overwrites valid and dirty data in the AVS memory, the coherence protocol ensures that the valid data is written back to main memory, eliminating the need for explicit DMA transfer instructions to remove data from the AVS memory.

Virtual Ways, presented here, is a lower-cost solution to the coherence problem. Unlike Speculative DMA, Virtual Ways uses a relaxed form of inclusion, in which the AVS memory always contains a subset of the data in the cache; however, ISE writes to the AVS memory employ a write-back policy that only updates the copy in the data cache when the processor, later, tries to read the data from the cache. Speculative DMA, in contrast, does not enforce inclusion.

Under Virtual Ways, data is loaded into the AVS memory using prefetch instructions, which eliminates the DMA engine. There is no hardware coherence protocol, which eliminates both the hardware overhead (i.e., duplicated tags) and the performance and energy overhead due to snooping and coherence traffic on the system bus. This improves both system performance and energy consumption.

Way Stealing is another solution to the coherence problem for AVS-enhanced ISEs [8]. The data cache is modified so that each way can be accessed as a compiler controlled memory, and all of the ways can be read or written in parallel. Each way, however, is a single-ported memory, which can limit the attainable speedup. For JPEG compression, for example, our ideal AVS memory is a 64-entry register file with 8 read and 8 write ports; the large number of read and write ports are feasible for such a small structure, but are not generally scalable



(a) State-of-the-art Automatic Instruction Set Extension algorithms provide high bandwidth to the ISE logic by adding Architecturally Visible Storage; however, they require extensive hardware added to a standard processor pipeline to guarantee memory coherence [3,9].

(b) Virtual Ways, the contribution of this work, puts the AVS on top of the data cache and extends the cache controller state machine to enforce coherence. This approach removes the separate bus interface of the AVS and the need for a coherence protocol in single processor systems.

Fig. 1. The difference between providing coherence in Speculative-DMA and Virtual Ways. In Speculative DMA the AVS is placed at the same level as the L1-caches. In Virtual Ways, on the other hand, the AVS is placed above the L1-caches and coherence is provided by inclusion.

for a larger number of entries. Under Way Stealing, the read and write operations to each stolen way must be serialized due to th small number of ports, which limits the maximum attainable speedup.

3 Virtual Ways

Historically, a single cache based processor system allows for a maximum of two copies of a given data structure in the system. One copy is always in main memory and one can be in the cache. In an n-way set associative cache, the location of a datum within the cache is indicated by the tag arrays and the associated status bits. The cache state-machine keeps track of the datum by updating the tag and state arrays accordingly. Any memory element in the system that is not covered by the tag and state arrays of the cache may exhibit coherence problems. This is precisely what occurs when AVS is introduced to an extensible processor without some form of coherence. The most recent copy of a particular datum may reside in the AVS, rather than the cache. Main memory, therefore, is liable to load an invalid copy of the same datum into the cache, unless it first updates the value from the AVS.

This is the classic problem of cache coherence; the fact that the AVS is not actually a cache does not, in principle, alter the problem; however, it does offer the possibility of a novel lightweight solution that is considerably less costly than a full-blown coherence protocol, which in the past has been used for multiprocessor systems. Our solution, which we call *Virtual Ways*, is to treat the AVS as

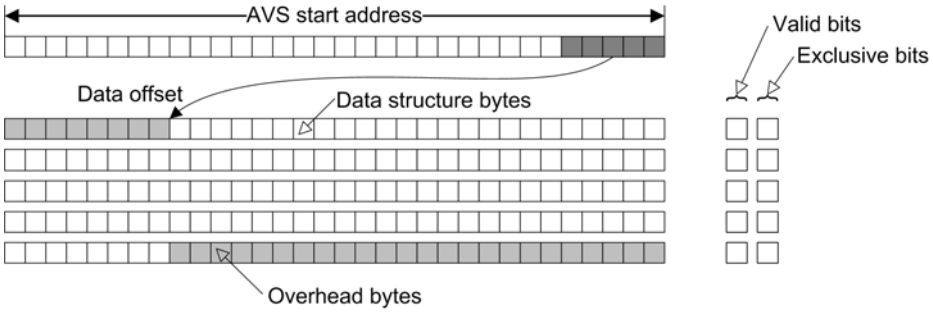


Fig. 2. The AVS is segmented in chunks the size of a cache line and the state is maintained for each segment separately. The tag consists of the start address and end address (length) of the AVS. For optimal performance care must be taken to avoid false sharing between neighboring data structures.

an additional way of the cache with respect to coherence. ISEs still access the AVS memory like a scratchpad under control of the compiler. The tag associated with the AVS memory, which is only used to ensure coherence, is implemented inside the cache. This way, ISE accesses to the AVS memory bypass the tag, which saves energy on each lookup. The cache controller is aware of the status of the data residing in the AVS due to its tag, and takes appropriate actions to ensure coherence. Virtual Ways can ensure coherence between an L1 cache and an AVS memory in a uniprocessor system.

For easier integration into the cache some adaptations are needed in comparison to scratchpad memories. The memory for the data structure held in the AVS memory is padded to a multiple of the size of a cache line. As the data structure to be loaded in the AVS is not necessarily aligned on a cache line boundary, the AVS must hold one additional cache line in order to accommodate all possible alignments. For optimal performance, an AVS-aware compiler could align data structures to avoid false sharing. For example, suppose that one data structure ends near the beginning of a cache line, and another data structure starts somewhere later on the same line. A write to a location in either data structure that resides on the cache line will invalidate the entire line, including a portion of the other data structure. This could, in principle, create unnecessary data transfers between the cache memories and the AVS.

Figure 2 illustrates the memory structure used to implement an AVS as a Virtual Way. Two bits per segment are required: one bit determines whether the segment is valid, and the second bit determines whether the copy in the AVS is exclusive. One set of tags for the AVS indicates the starting and ending addresses of the data structure stored in the AVS. This set of tags is used to determine if a CPU access issued to the cache is within the region contained within the AVS. The set of tags and the state bits permit the cache controller to determine where the most recent copy of the requested datum resides.

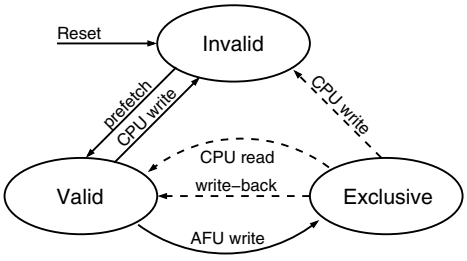


Fig. 3. Transition digram for each segment of the AVS; each segment can be in one of three states Invalid, Valid, or Exclusive. Associated ISEs execute while all segments are either valid or exclusive. ISE writes to an AVS segment cause it to become exclusive. An exclusive segment must be written back to the cache when the segment transitions into another state.

An ISE enhanced with an AVS can only execute when all segments are valid, as all accesses to the AVS must hit. We do not impose any restrictions on the ISE's access patterns within the AVS, beyond the requirement that the data reside in the AVS before the ISE begins to execute. Specialized prefetch instructions are used to load data into the AVS and update the tag before the ISE can execute. Similar to caches, data eviction from the AVS is achieved via lazy write back; however, an AVS-flush instruction is also available. If the data is accessed through a normal software instruction, the cache controller, which maintains coherence, will copy the data into the cache, and invalidate the data in the AVS if it is overwritten. In our experiments, we did not use the AVS-flush operation. Our expectation is that the AVS flush operation would only be used to facilitate context switching; our evaluation platform is application-specific, so we do not employ multiple processes and context switching does not occur.

3.1 AVS Segment States

Each segment of the AVS can be in one of three states. These are:

1. *Invalid State*: the initial state of the AVS, in which no segment contains valid data. This occurs when the processor is first powered up, or if the AVS contains a copy of a data structure that is not the most recent, i.e., a separate copy, either in the cache or main memory, has been modified, while the copy residing in the AVS memory has not been updated.
2. *Valid State*: a segment of the AVS contains the most recent copy of a data structure. Valid copies of the same line also exist in the cache.
3. *Exclusive State*: a segment of the AVS contains the most recent copy of a data structure. The copy in the cache, if any, is dead.

Figure 3 depicts the state machine for one segment of an AVS. Dashed arrows indicate the transitions where the data must be written back to the cache.

3.2 Prefetching Operation

Here, we describe the basic actions of the prefetch instruction, which must complete before an ISE can access the AVS. Here, we define an AVS region to be a set of m segments, each of which is equal to the size of a cache line. There are two general cases to consider:

1. *AVS Region Match*: This occurs if the address of the requested data matches a segment contained within the AVS. If the state of the segment is valid or exclusive, then the most recent copy of the data already exists in the AVS; the data must be loaded into the AVS only if the state is invalid. If a valid copy of the data exists in another way of the cache, then it can be loaded directly into the AVS, bypassing the bus; otherwise, the data is loaded from main memory and is written to the cache and AVS concurrently. See Figure 4 (e) for a prefetch operation that reloads only one segment.
2. *AVS Region Mismatch*: This occurs if the address of the requested data does not match a segment contained within the AVS. If one of the segments contained within the AVS is currently exclusive, then it must be written back to the cache/main memory so that the most recent copy of the data is not lost. Afterwards, all segments are marked invalid and the start and stop tags are updated for the new data structure. The load operation then proceeds as described above, with a region match and the AVS segments in an invalid state. See Figure 4 (f) for the case where the AVS is written back before it is loaded with a new data structure.

The region matching behavior enforces an inclusive, write-through policy. *Inclusion* is maintained, because the lines in the AVS are a subset of the lines in the cache. This is a relaxed form of inclusion, however, because ISE writes that modify an AVS segment do not modify the corresponding line in the cache. The policy is *write through*, in the sense that prefetch instructions write "through" the cache directly to the AVS.

3.3 Maintaining Coherence After the ISE Executes

We assume that the data has been prefetched into the AVS, as described in the preceding section. When an ISE executes, it may modify the data structure in the AVS. If the data is modified, then at least one line is left in the exclusive state. After the ISE executes, control returns to the CPU. The data in the AVS will either be written back upon request, or as dictated by coherence requirements. The correct action to take by a software load or store instruction depends on the state of the segment.

1. *Invalid State*: An invalid segment can be ignored; the data at the requested address resides in the cache or main memory.
2. *Valid State*: Here, the AVS contains valid data that was not modified by the ISE. A valid copy of the data may also exist in the cache. For a read access, either valid copy of the data can be returned. Writes are somewhat more

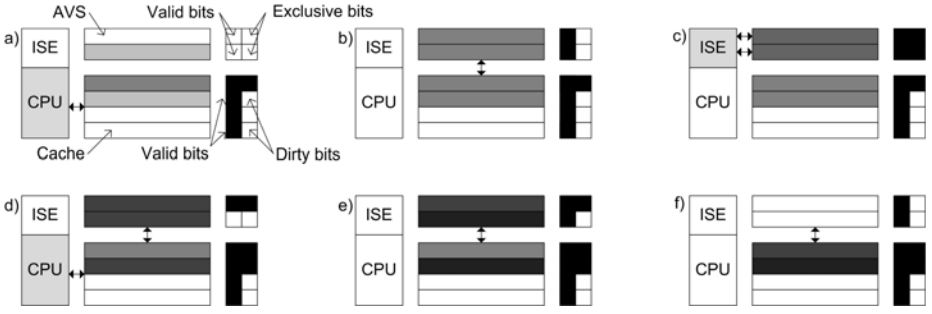


Fig. 4. This figure shows some lines of the cache and the corresponding segments in the AVS together with associated state bits during a typical AVS scenario. The AVS starts up in invalid state (a) and is then preloaded with a data structure (b) and transitions to valid state. Execution of the ISE will modify the data structure turning on (some of) its exclusive bits (c). On a CPU access the data is copied back to the cache and, on a write access, invalidated in the AVS (d). A prefetch instruction for the same structure will restore it to the AVS (e). A prefetch instruction for another structure will write back all exclusive lines and load the requested structure (f).

complex, as coherence must be maintained between the cache and the AVS. One possibility is to employ a write-through policy that updates the data in both the AVS and the cache; a second alternative is to update the data in the cache and invalidate the data in the AVS. We have opted for the latter option, because a pipelined write-through could potentially cause a memory consistence problem between the data cache and the AVS. A memory consistence problem occurs when a read of data does not return the latest value written to it. This situation can occur with a pipelined write-through. Applying a write-through without pipelining would drastically impact the processor's critical path.

3. *Exclusive State*: In this case, only the AVS contains the most recent copy of the data, and this copy must be written back to the cache before the access can complete; the corresponding line in the cache is marked as dirty, and the AVS segment reverts to the valid state, as the data in the AVS is no longer exclusive. Figure 4 (d) depicts the case of a CPU write access when the corresponding AVS segment is in exclusive state.

3.4 Multiple AVS Memories

The preceding discussion assumes that there is one AVS memory. In principle, an ISE may access multiple data structures, and writes to both may benefit from parallel execution. In this case, we would want to instantiate multiple AVS memories: one per data structure. To facilitate this change, we require an additional tag and state bits for each AVS that must be checked to maintain coherence.

The compiler can avoid inter-AVS transfers by guaranteeing that memory regions loaded in distinct AVS memories will never overlap. In the most general case, pointer analysis is undecidable. As described by Biswas et al. [3], only data structures that have been disambiguated can be moved into an AVS memory. Although this approach is conservative, it is necessary to ensure correctness when compiling languages such as *C/C++* that permit arbitrary pointer arithmetic.

4 Experimental Setup

Our experimental platform is an internally-developed FPGA-based soft processor that implements the OpenRISC instruction set. We modified the data cache implementation to account for Speculative DMA [9] and Virtual Ways. Our multi-processor platform allows us to emulate from one to seven OpenRISC processors. The platform has software-configurable 16 kB instruction caches and software-configurable 16 kB data caches with a choice of MSI-states, MESI-states, or disabled hardware coherence protocol. Our implementation of Speculative DMA uses the MESI-states protocol in our experiments. Our implementation of Virtual Ways eliminates the DMA controller, as data is brought into the AVS memory through the data cache interface. The only other hardware modification was to augment the cache state machine as described in the preceding section.

Mimosys Clarity, a compiler that uses the algorithm proposed by Biswas [3], identified the ISEs and generated the VHDL implementations of the ISE logic. We modified the AVS memory to support Speculative DMA through a DMA interface and Virtual Ways through the data cache interface; the appropriate interface is selected via software control. A system deployed in the real world would support one option or the other, but not both.

Our goal is to demonstrate that Virtual Ways offers a comparable speedup to Speculative DMA, but at a significantly reduced hardware and energy cost. We took the EEMBC consumer V2 testbench suite and performed an ISE identification on the unmodified source code by taking the first dataset of each algorithm as test case. It has to be noted at this point that Mimosys Clarity does not implement the opportunistic Speculative DMA as proposed by Kluter *et al.* [9]. All the C-code has been cross-compiled using a gcc 3.4.4 toolchain based on "newlib" for the OpenRISC.

5 Experimental Results

To perform a comparison between the different methods, we performed a design space exploration of all algorithms on a non-ISE enhanced processor. We varied the size and associativity of both the instruction and data caches. The configuration with the best energy-performance product for a given algorithm and dataset is chosen as reference for comparison. We performed a similar design space exploration for the processor augmented with larger AVS-enhanced ISEs, using both Speculative DMA and Virtual Ways to ensure coherence.

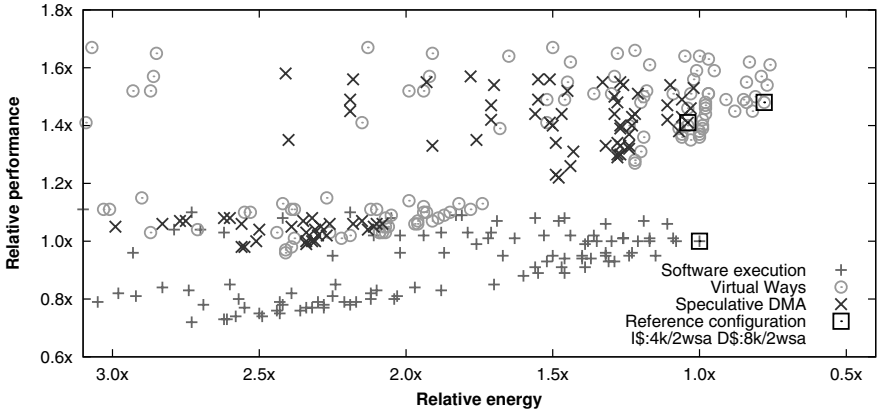


Fig. 5. Design space exploration of the CJPEGV2 dataset 1 compression algorithm for the different architectural versions.

The result of the design space exploration for the CJPEGV2 testbench using the first data set is plotted in Figure 5. Both Speculative DMA and Virtual Ways achieved greater speedups than the original code across all cache configurations. Many, but not all, configurations achieved greater reductions in energy when Speculative DMA or Virtual Ways were used. Except for the reference cache configuration, the figure does not indicate which Speculative DMA and Virtual Way data points correspond to the same configuration; the general trend, however, appears to be that Virtual Ways achieve marginal better performance with a noticeable reduction in energy compared to Speculative DMA.

Figure 6 shows the energy and performance plots of the EEMBC consumer version 2 benchmark suite. For all benchmarks Virtual Ways outperforms the state-of-the-art while consuming significantly less energy. There are two observations to be made: (1) for the AES algorithm both Speculative DMA and Virtual Ways perform equally with an identical energy footprint. The reason lies in the detection of two AVS memories that contain read-only data structures; therefore, both methods do not have to infer coherence traffic, and (2) for the MPEG2_ENC both methods provide better performance at a significant energy cost when compared to the baseline. The increase in energy lies in the access pattern of the detected AVS. In the MPEG2_ENC benchmark a temporary buffer of the size of 8×8 16-bit integers is used to perform a 64-point *Discrete Cosine Transform (DCT)*. The DCT is selected as potential ISE, and the buffer is placed in an AVS. Due to the algorithm the buffer is moved forth and back between the AVS and the data cache consuming significant energy. In case of execution on a non-ISE enhanced processor the buffer is never evicted from the data-cache due to the *Least Recently Used (LRU)* replacement policy. To explain why the other algorithms do not suffer similarly, we compare the data points corresponding to the reference cache configuration of the CJPEGV2 algorithm using dataset 1 in greater detail.

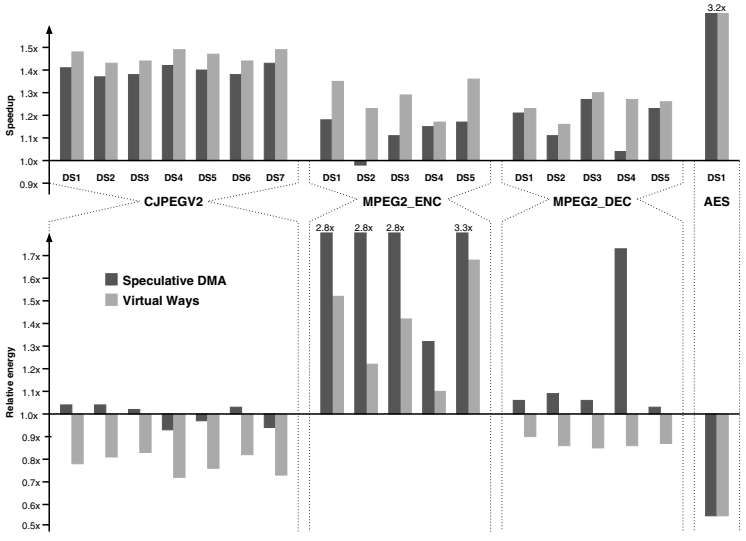


Fig. 6. Performance and energy results for four EEMBC benchmarks. Each of the algorithms, except the AES, contains five to seven different datasets (DSx). The baseline is the cache configuration that provides the best energy-performance product when running on a non-ISE enhanced processor. Overall, Virtual Ways provides similar to more performance with significant reduced energy consumption when compared to Speculative DMA.

Figure 7 shows the performance and energy breakdown for the four different kernels of the CJPEGV2 algorithm for the reference cache configuration. Similarly to the MPEG2_ENC benchmark the DCT kernel is the only kernel containing a custom instruction with an AVS. One would expect to observe two different scenarios: (1) upon entering the DCT kernel, the data has to be copied to the AVS, before the custom instruction can start processing the data, and (2) after leaving the DCT kernel the data has to gradually move back to the data cache for the processor to be able to process it in the quantization kernel.

Looking into the copying of the data structure into the AVS, Figure 7 shows no distinct differences between Speculative DMA and Virtual Ways in terms of performance or energy consumption, contrary our observation for the MPEG2_ENC benchmark. The reason for this lies in the calculation pattern of the color space conversion. The color space conversion processes a "band" of 1024 pixels, 8 rows at a time. As this "band" corresponds to a memory size of 24 kB, it cannot fit in the data cache entirely, and therefore will evict parts of the processed data. By the time the DCT kernel starts processing, the data required in the AVS is no longer present in the data cache; therefore, no coherence problem exists and both Speculative DMA and Virtual Ways need to prefetch the data from main memory. As this process affects both methods, both architectures perform equally and consume about the same amount of energy in this particular case.

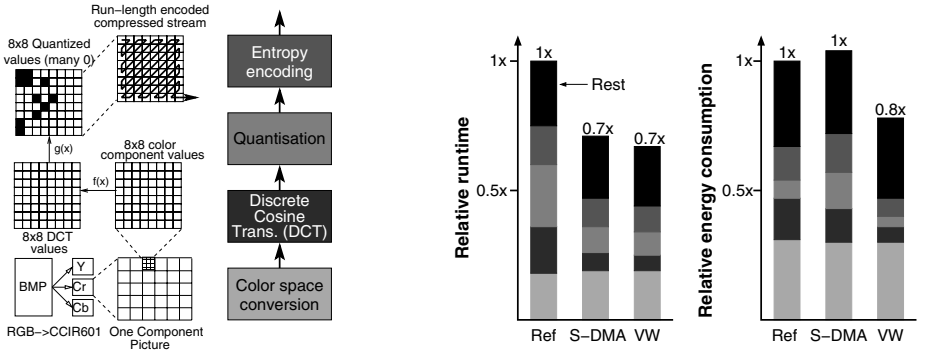
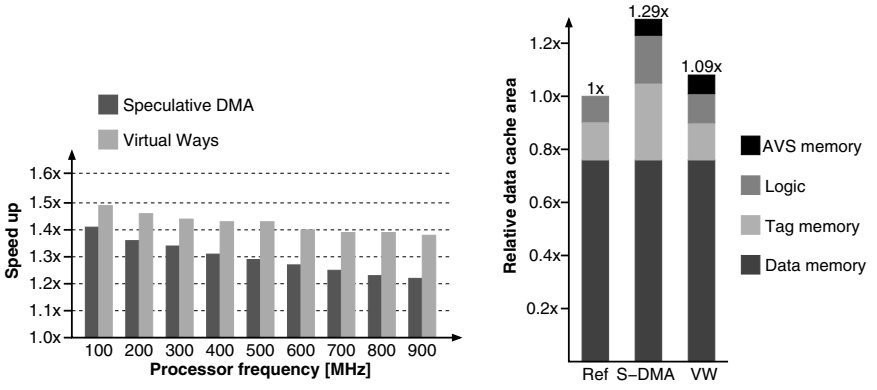


Fig. 7. Left: Schematic diagram of the kernels of the CJPEGV2 compression algorithm. Right: Performance and energy consumption broken down into the different kernels as shown on the left for the baseline (Ref), Speculative DMA (S-DMA), and Virtual Ways (VW).

Figure 7 shows distinct differences for the data eviction process from the AVS. Where for Speculative DMA the energy consumption in the quantization kernel is high (4.4× the energy consumed by the non-ISE enhanced architecture), Virtual Ways expends a comparable amount of energy as the software implementation. The reason for this is that the data structure in the AVS has been modified by an ISE in the DCT kernel and is then directly used in the quantization kernel. In this case, a coherence problem exists between the AVS and the data cache. In Speculative DMA the coherence protocol will move the data structure back from the AVS to both the data cache and main memory, which includes expensive bus transfers; this consumes a significant amount of energy. In contrast, Virtual Ways simply copies the data directly from the AVS segments to the cache. This eliminates the need for bus transfers and writes to main memory.

The bus dependency of the Speculative DMA coherence mechanism is an uncertainty. Due to the well known *memory wall problem* the processor normally runs at higher clock frequencies than the external memory. For all of the preceding experiments, we assumed memory and processor frequencies of 100 MHz, which is a favorable situation for Speculative DMA. Increasing the processor clock frequency can influence the operation of Speculative DMA in the benchmark, as shown in Figure 8(a); Figure 8(a) also shows that the performance of Virtual Ways is less dependent on the difference between processor and memory frequencies.

To compare the area of Virtual Ways and Speculative DMA, we implemented both data caches, including AVS memories, in a 90 nm standard-cell technology, along with a baseline cache without an AVS; we did not synthesize instruction caches, the processor, or the ISE computational logic. The results are depicted in Figure 8(b), which shows that Virtual Ways increases the area of the baseline cache by 9%, while Speculative DMA increases the area by 29%.



(a) Influence of the processor frequency with respect to the external memory frequency for the execution of the CJPEGV2 benchmark.

(b) Area overhead comparison of a standard data cache (Ref), a Speculative DMA enhanced data cache (S-DMA), and a Virtual Ways enhanced data cache (VW).

Fig. 8. Frequency robustness and Area of Virtual Ways compared to Speculative DMA

6 Conclusion

Prior work has established that AVS-enhanced ISEs provide a performance improvement over ISEs that do not employ AVS; however, the inclusion of AVS in a processor with caches creates a memory coherence problem. This paper has introduced Virtual Ways as a low-cost alternative to using a coherence protocol to maintain this coherence in a single-processor system. Our results show that a cache enhanced with Virtual Ways consumes less area and energy than Speculative DMA; additionally, Virtual Ways was shown to be less sensitive than Speculative DMA to differences in clock frequencies between the processor and main memory. For these reasons, we believe that Virtual Ways is a much more attractive solution than Speculative DMA for customizable processors used in cost and energy-constrained embedded systems.

References

1. Atasu, K., Mencer, O., Luk, W., Özturan, C., Dünda, G.: Fast custom instruction identification by convex subgraph enumeration. In: Proceedings of the 19th International Conference on Application-specific Systems, Architectures and Processors, Leuven, Belgium, July 2008, pp. 1–6 (2008)
2. Biswas, P., Choudhary, V., Atasu, K., Pozzi, L., Ienne, P., Dutt, N.: Introduction of local memory elements in instruction set extensions. In: Proceedings of the 41st Design Automation Conference, San Diego, Calif., June 2004, pp. 729–734 (2004)
3. Biswas, P., Dutt, N., Pozzi, L., Ienne, P.: Introduction of architecturally visible storage in instruction set extensions. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems CAD-26(3), 435–446 (March 2007)

4. Clark, N., Zhong, H., Mahlke, S.: Processor acceleration through automated instruction set customisation. In: Proceedings of the 36th Annual International Symposium on Microarchitecture, San Diego, Calif., December 2003, pp. 129–140 (2003)
5. Cong, J., Han, G., Zhang, Z.: Architecture and compiler optimizations for data bandwidth improvement in configurable embedded processors. IEEE Transactions on Very Large Scale Integration (VLSI) Systems 14(9), 986–997 (2006)
6. Jayaseelan, R., Liu, H., Mitra, T.: Exploiting forwarding to improve data bandwidth of instruction-set extensions. In: Proceedings of the 43rd Design Automation Conference, San Francisco, Calif., July 2006, pp. 43–48 (2006)
7. Karuri, K., Chattopadhyay, A., Hohenauer, M., Leupers, R., Ascheid, G., Meyr, H.: Increasing data-bandwidth to instruction-set extensions through register clustering. In: Proceedings of the International Conference on Computer Aided Design, San Jose, Calif., November 2007, pp. 166–171 (2007)
8. Kluter, T., Brisk, P., Charbon, E., Ienne, P.: Way stealing: Cache-assisted automatic instruction set extensions. In: Proceedings of the 46th Design Automation Conference, San Francisco, Calif., July 2009, pp. 31–36 (2009)
9. Kluter, T., Brisk, P., Ienne, P., Charbon, E.: Speculative DMA for Architecturally Visible Storage in Instruction Set Extensions. In: Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis, Atlanta, Ga., October 2008, pp. 243–248 (2008)
10. Pothineni, N., Kumar, A., Paul, K.: Application specific datapath extension with distributed I/O functional units. In: Proceedings of the 20th International Conference on VLSI Design, Bangalore, India (January 2007)
11. Pozzi, L., Atasu, K., Ienne, P.: Exact and approximate algorithms for the extension of embedded processor instruction sets. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems CAD-25(7), 1209–1229 (2006)
12. Pozzi, L., Ienne, P.: Exploiting pipelining to relax register-file port constraints of instruction-set extensions. In: Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems, San Francisco, Calif., September 2005, pp. 2–10 (2005)
13. Steinke, S., Wehmeyer, L., Lee, B.-S., Marwedel, P.: Assigning program and data objects to scratchpad for energy reduction. In: Proceedings of the Design, Automation and Test in Europe Conference and Exhibition, Paris (March 2002)
14. Verma, A.K., Brisk, P., Ienne, P.: Rethinking custom ISE identification: A new processor-agnostic method. In: Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems, Salzburg, September 2007, pp. 125–134 (2007)
15. Verma, A.K., Brisk, P., Ienne, P.: Fast, quasi-optimal, and pipelined instruction-set extensions. In: Proceedings of the Asia and South Pacific Design Automation Conference, Seoul, Korea, January 2008, pp. 334–339 (2008)