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

Samuel Burri[2], Theo Kluter[1], Philip Brisk[1], 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
[2] Ecole Polytechnique Fédérale de Lausanne (EPFL), School of Engineering, CH-1015 Lausanne, Switzerland
[3] Delft University of Technology, Circuits and Systems Group, NL-2600 AA Delft, The Netherlands
Emails: {samuel.burri,ties.kluter,philip.brisk, edoardo.charbon,paolo.ienne}@epfl.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 accessibly exclusively to the ISEs. Unfortunately, the usage of AVS memories creates a coherence problem with the data cache. This paper introduces Virtual Ways, a lightweight solution to this coherence problem that does not employ a full-blown coherence protocol. Using JPEG compression as a case study, we show that Virtual Ways achieves comparable performance to a similar system with a coherence protocol, while reducing the area overhead and energy consumption. Compared to a 4 kB 4 way set-associative data cache without AVS, Virtual Ways increases the overall cache area by 9%, compared to a 29% overhead for a coherence protocol-based solution. The reference configuration and coherence protocol-based solutions consume comparable energy, while Virtual Ways achieves an energy reduction of around 20%. Lastly, the coherence-based solution is shown to be sensitive to the difference in clock frequency between the processor and main memory, whereas, Virtual Ways is wholly robust. Altogether, Virtual Ways is a better choice for extensible processors used in cost- and energy-constrained embedded systems.

Key words: Application-Specific Processors, Memory Coherence, Instruction Set Extensions, Virtual Ways

1 Introduction

As technology advances, the performance requirements of embedded systems grow continuously; however, these requirements must be met within a stringent

energy budget. At the same time, readily increasing fabrication costs mandate high production volumes in order for products to remain economically competitive. Furthermore, early entry into emerging markets is of the utmost importance for a new product line to gain traction; consequently, the classic engineering mandates of "better, faster, and cheaper," which qualify the product itself, must now be augmented with "earlier" with respect to speed of its design.

As systems increase in complexity, a greater portion of the product life cycle is consumed by verification, both software and hardware, and pre- and post-silicon. As verification is a classic intractable problem, further pressure is placed on product engineers to deliver properly functioning products on time. For this reason, one emerging trend in embedded system design is component reuse; rather than designing and verifying a new product from scratch, hardware platforms can be assembled by composing pre-existing *intellectual property (IP)* cores, coupled with the reuse of existing software libraries, whenever possible.

Processors are among the most flexible forms of intellectual property, as they can perform any computation; their drawback, however, is that they are not a particularly efficient implementation in terms of performance, area utilization, or silicon efficiency, compared to an *application-specific integrated circuit (ASIC)* implementation of the same computation. Nonetheless, it is easier to program a processor than to design an ASIC, and the time and risk involved in software verification is much less than in pre- and post-silicon verification of an ASIC.

For the reasons outlined above, system designers are increasingly turning toward customizable embedded processors to meet their needs. These processors are extensible, meaning that they contain hooks that allow the user to augment the processor's instruction set with application-specific custom *instruction set extensions (ISEs)* [1, 2]. ISEs are effectively ASICs on a small scale, in the sense that they are designed for small computational kernels, rather than complete applications; however, this approach is easily justified due to the fact that most embedded applications spend a large proportion of their execution time in a few deeply nested loops.

Furthermore, ISEs are favorable from the perspective of designer productivity. Compiler techniques to automatically analyze an application to extract good ISE candidates, synthesize them as custom hardware, and interface them with the processor already exist; comparable techniques for automated ASIC design at the application-level are an ideal that has been pursued for more than 30 years, but has not yet come to fruition. Moreover, the use of extensible processors alleviates the burden of pre-silicon verification from system designers; the base processor itself is a pre-verified IP core. Likewise, the problem of pre-silicon verification of the ISEs and their interface to the processor is reduced to the one-time verification problem of the compiler itself. Thus, system designers must only verify the software implementation of the application under development, and the silicon that comes back from the fabrication facility. These two factors significantly reduce time-to-market and the overall cost of embedded system design.

One of the challenges in extensible processor design is to provide high data bandwidth between the processor and the ISE. One approach that has been proposed in the past is to augment the ISEs with *Architecturally Visible Storage (AVS)*, which can be either registers or compiler-controlled memories [3]. AVS-aware ISE identification methods exist, so their inclusion does not complicate or otherwise slow down the process of system design.

Correctly dealing with AVS memories has proven to be a challenging task with significant area overheads. AVS memories, historically, have been distinct from the cache hierarchy. *Direct Memory Access (DMA)* has been used to transfer data between main memory and AVS memory, bypassing the caches. This approach has two drawbacks. Firstly, a DMA engine, which requires a considerable quantity of silicon, is required. Secondly, the use of a separate memory structure outside of the cache hierarchy creates coherence problems. To address the second drawback, Kluter et al. [4] proposed to use a multiprocessor cache coherence protocol. This approach is reasonable in multiprocessor systems where a coherence protocol can safely be assumed to exist; however, the cost of adding a coherence protocol is likely to be prohibitive.

To address this concern, this work introduces *Virtual Ways*, which ensure coherence between the AVS and an L1-cache, without using a full coherence protocol. The key idea is to let the AVS memories function as extra ways of the cache with respect to coherence; the cache controller is modified to be aware of the AVS in order to guarantee coherent behavior. The AVS itself is not available to the cache as an actual way for use during normal processor execution. Our analysis shows that Virtual Ways are much less costly than a full-blown coherence protocol in terms of both energy consumption and area overhead, and are also significantly more robust to the difference in clock frequency between the processor and main memory.

The rest 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. Section 6 concludes the paper.

2 Related Work

In the earliest works on automated ISEs, all communication between the ISE and the extensible processor goes through the processor register file [1, 2]. If the processor is a RISC, this means that each ISE was limited to two inputs and one output. This significantly limited the size of the ISEs that could be found. To overcome the I/O constraint, several techniques have been proposed to identify and synthesize multi-cycle ISEs that transfer data between the register file and ISE logic every cycle [5–9].

Although these techniques were effective, the I/O bandwidth between the register file and ISE logic quickly became a performance bottleneck. Several ar-

chitectural techniques have since been proposed to increase the I/O bandwidth. One proposal, for example, used the pipeline forwarding logic in a RISC processor to increase the number of inputs to the ISE logic by two [10]. Another proposal permits the ALU of the extensible processor to transmit data directly to *shadow registers* in the ISE logic in parallel with writes to the register file; one advantage of this method is, that it does not require any additional information to be encoded in the base processor instruction set, however, it creates some complications for the compiler. Although both of these methods are effective, they only increase the input bandwidth to the ISE logic; Verma et al. [7] identified a large 22-input, 22-output ISE in the *Advanced Encryption Standard (AES)* benchmark; consequently, even with these modifications, 22 cycles would be required to write the data back to a processor register file with only a single write port.

Another proposal, which increases both input and output bandwidth, is to replace the processor's register file with a multi-bank clustered VLIW-style register file, where each cluster can be read and written independently [11]. The drawback of this approach is that parallel data must reside in different banks, creating a complex data placement problem and copy coalescing problem that the compiler must solve; in principle, the overhead of the copies required to achieve the favorable data placement works against the speedup achieved by the ISEs.

None of the aforementioned techniques considered the use of AVS, or provided any type of path between the memory subsystem and the ISE logic. Historically, memory operations (e.g., loads and stores) were forbidden from ISE logic, because their latencies vary, depending on whether the memory access is a hit or a miss in the L1 cache. On the other hand, memories placed under compiler control, so-called *scratchpad memories* [12], have deterministic latency because the compiler ensures that all accesses hit. Thus, the use of ISEs in conjunction with scratchpad memories has two benefits: load and store operations are no longer forbidden from inclusion in ISEs, which permits the compiler to find larger and more effective ones; and, the scratchpad can be read and written concurrently with reads and writes to the processor register file, thereby increasing data bandwidth.

The principle difference between a scratchpad memory and the AVS discussed in this paper is subtle, but important. Scratchpads have been proposed as an *alternative* to caches for embedded systems because the elimination of the tag array reduces the energy consumed per-access, and deterministic hit/miss behavior makes the worst-case execution time predictable. The AVS memories discussed here, like scratchpads, do not have tag arrays and all accesses hit; however, they co-exist with L1 caches, rather than replacing them. This is why the coherence problem manifests itself.

The concept of AVS was introduced in a paper by Biswas et al. [13]; this work limited the AVS to small ROMs that hold constant values and state registers. In a subsequent work, Biswas et al. [3] augmented their ISEs with small memories to hold arrays that are accessed by the ISEs. DMA transfers move data

between these AVS memories and the main memory system, bypassing the cache hierarchy. Kluter et al. [4] observed that this approach could lead to incorrect results because coherence between the AVS and L1-cache was not maintained; to correct the situation, the AVS and L1-cache were integrated into a coherence protocol. Coherence protocols are typically meant for multiprocessor systems: their area overhead and impact on performance and power consumption due to increased bus traffic are significant.

Virtual Ways, the solution proposed in this paper, achieves coherence, but without the overhead of a full cache coherence protocol. Instead, the AVS is integrated into the cache and shares its bus interface, eliminating the need for DMA transfers; the cache controller is extended to guarantee coherence. Virtual Ways retains the performance and bandwidth benefits of DMA-enhanced AVS memories, while ensuring coherence, but provides a much lower cost solution.

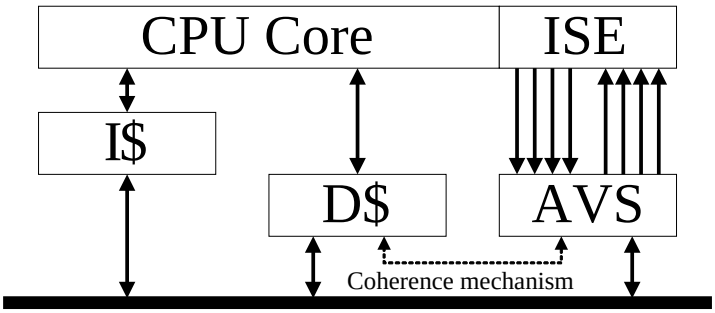

Fig. 1. State-of-the-art Automatic Instruction Set Extension algorithms provide high bandwidth to the ISE logic by adding Architecturally Visible Storage, however, they require extensive hardware added to a standard processor pipeline to guarantee memory coherence [3, 4].

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. By the introduction of the n-way set-associative caches, more than two copies could reside in the system, namely one in main memory and a copy in one or multiple of the n-ways. To prevent an inter cache coherence problem, a n-way set-associative cache holds by construction only a single copy of any datum. The location of this copy is indicated by the tag arrays and associated status bits. The cache state-machine keeps track of the copies by

updating accordingly the tag and state arrays. 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 happens when AVS is introduced to an extensible processor without some form of coherence. The most recent copy of a particular data item may reside in the AVS, rather than the cache. If the data resides in the cache, then some coherence mechanism is required to invalidate it; similarly, if the data is loaded from main memory into the cache before the value in the AVS has been written back, then the data loaded into the cache will be invalid.

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 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.

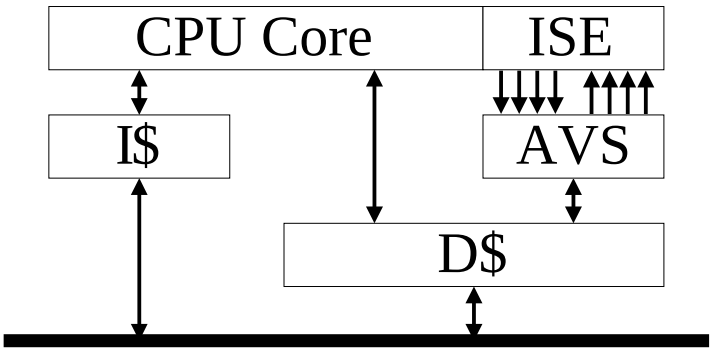

Fig. 2. Virtual Ways, the contribution of this work, puts the AVS on top of the data cache and extends the cache controller state machine to enforce coherence. This approach removes the separate bus interface of the AVS and the need for a coherence protocol in single processor systems.

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 3 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 indicate 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.

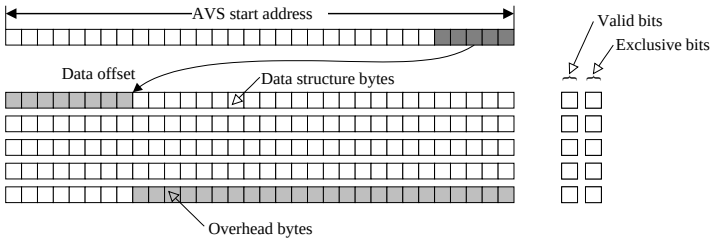

Fig. 3. The AVS is segmented in chunks the size of a cache line and the state is maintained for each segment separately. The tag consists of the start address and end address (length) of the AVS. For optimal performance care must be taken to avoid false sharing between neighboring data structures.

An 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, so 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 4 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.

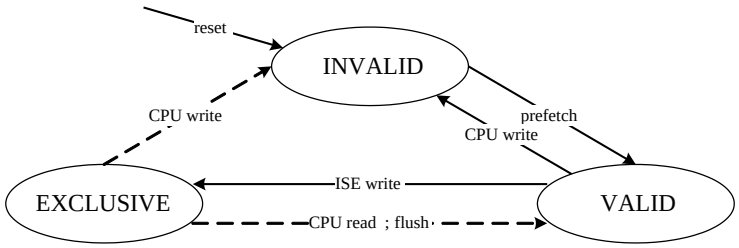

Fig. 4. Each segment of the AVS can be in one of three states. Associated ISEs execute while all segments are either in valid or exclusive state. In exclusive state the AVS contains the most recent copy of the memory produced by the ISE and the segment has to be written back to the cache when it transitions into another state.

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 5 (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 5 (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 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. By delaying the write through pipelining this situation can potentially happen. Applying a write-through without pipelining it would drastically impact the processors 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 5 (d) depicts the case of a CPU write access when the corresponding AVS segment is in exclusive state.

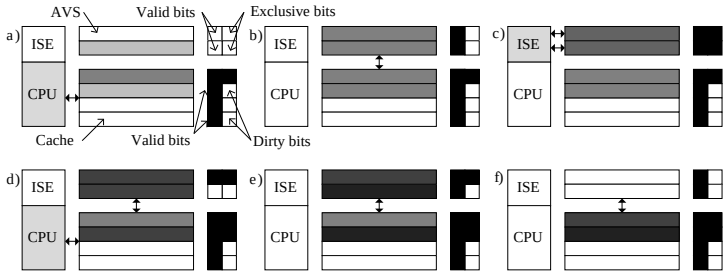

Fig. 5. This figure shows some lines of the cache and the corresponding segments in the AVS together with associated state bits during a typical AVS scenario. The AVS starts up in invalid state (a) and is then preloaded with a data structure (b) and transitions to valid state. Execution of the ISE will 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).

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 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 undecideable. 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

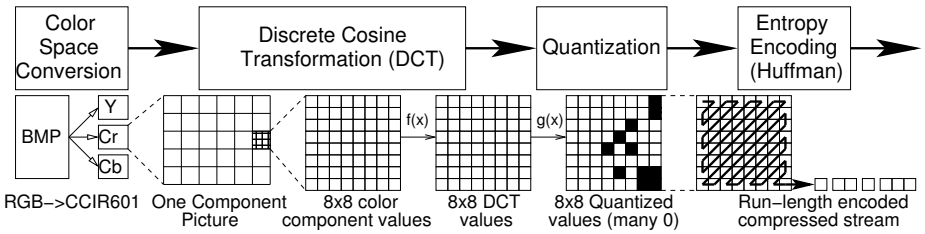

Fig. 6. Block diagram of the JPEG compression algorithm used as motivational example. The *Discrete Cosine Transformation (DCT)* kernel contains a custom instruction utilizing an AVS. The quantization kernel contains a custom instruction not utilizing an AVS.

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 [4] and Virtual Ways. Our multi-processor platform allows us to emulate 1-7 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 goal is to demonstrate that Virtual Ways offers a comparable speedup to Speculative DMA, but at a significantly reduced hardware and energy cost. We used the same JPEG compression algorithm as Kluter et al. [4], who introduced Speculative DMA, for comparison. The JPEG compression algorithm consists of four computational kernels as shown in Figure 6. Automated ISE identification algorithms [3, 4] detected two ISEs. The first ISE, found in the *Discrete Cosine Transformation (DCT)* kernel, utilizes an AVS memory. The data structure is an 8x8 matrix of 16-bit integers. We implemented this AVS as a small memory (a register file, for all intents and purposes) with 8 read and 8 write ports. Such a number of read and write ports would be exorbitant for larger memories, but is feasible in this particular case. The second ISE was a hardware divider found in the *Quantization* kernel, which does not employ an AVS memory. Our OpenRISC platform does not provide a native division instruction.

An internally developed compiler identified the ISEs and generated their VHDL implementations of the ISE logic; the necessary cache modifications to

support Virtual Ways were done by hand. We modified the AVS memory such that it supports Speculative DMA and Virtual Ways by software control.

All the C-code has been cross-compiled using a gcc 3.4.4 toolchain based on "newlib" for the OpenRISC. For all the experiments we used the same 24-bit RGB encoded picture of 1024x768 pixels, similar to the resolution of current high-end web-cams and standard portable phones.

5 Experimental Results

To perform a comparison between the different methods, we performed a design space exploration of the JPEG algorithm on a non-ISE enhanced processor. We varied the size and associativity of both the instruction and data caches. The configuration with a 2 kB 2 way set-associative instruction cache and a 4 kB 4 way set-associative data cache has the best energy-performance product for this application. We treat this cache configuration as the 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.

The result of the design space exploration is plotted in Figure 7. 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 comparable performance with a small, but noticeable, reduction in energy compared to Speculative DMA. Next, we compare the data points corresponding to the reference cache configuration in greater detail.

Figure 8 shows the performance and energy breakdown for the four different kernels of the JPEG algorithm for the reference cache configuration. As stated before, 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 8 shows no distinct differences between Speculative DMA and Virtual Ways in terms of performance or energy consumption. 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

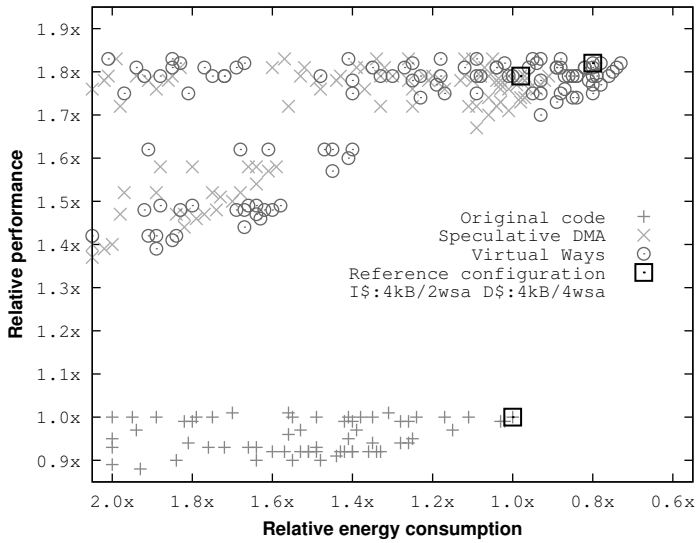

Fig. 7. Design space exploration of the JPEG compression algorithm for the different architectural versions.

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.

On the other hand, Figure 8 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 quantization kernel, as shown in Figure 9; Figure 9 also shows that the performance

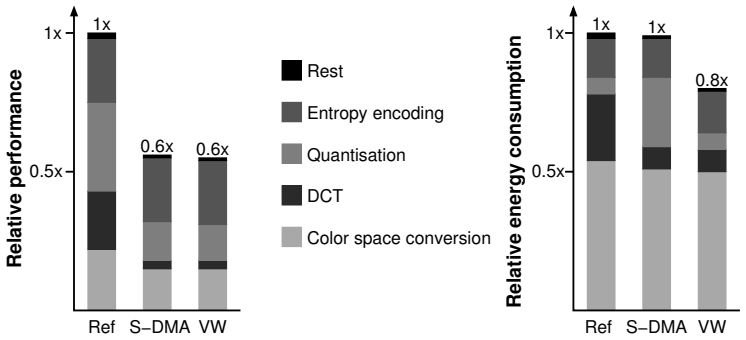

Fig. 8. Kernel breakdown of the JPEG compression algorithm for the reference configuration.

of Virtual Ways is independent of the difference between processor and memory frequencies.

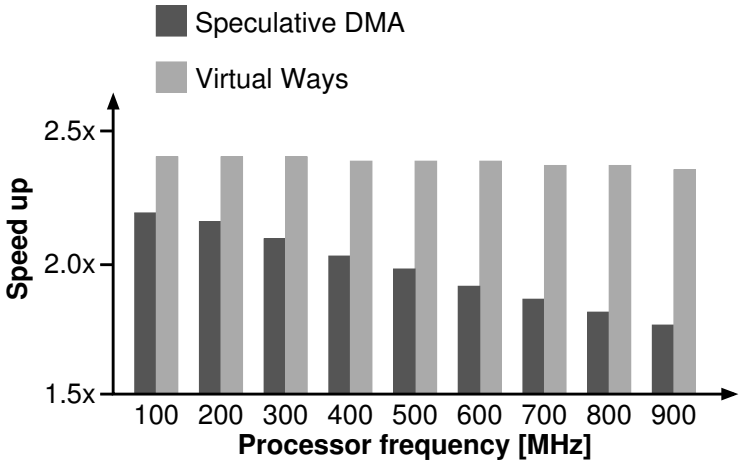

Fig. 9. Influence of the processor frequency with respect to the external memory frequency for the execution of the quantization kernel.

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 10, which shows that Virtual Ways increases the area of the baseline cache by 9%, while Speculative DMA increases the area by 29%.

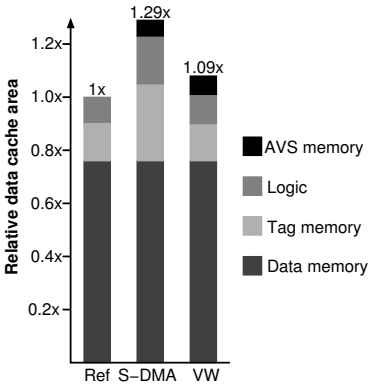

Fig. 10. Area overhead comparison of a standard data cache (Ref), a Speculative DMA enhanced data cache (S-DMA), and a Virtual Ways enhanced data cache (VW).

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, an existing coherence protocol-based solution; additionally, Speculative DMA was shown to be sensitive to differences in clock frequencies between the processor and main memory, while Virtual Ways was wholly robust to the difference. 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. 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) (July 2006) 1209–29
2. 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) 129–40
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) (March 2007) 435–46
4. 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) 243–48

5. 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) 2–10
6. 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)
7. 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) 125–34
8. 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) 334–39
9. Atasu, K., Mencer, O., Luk, W., zturan, C., Dnda, 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) 1–6
10. 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) 43–48
11. 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) 166–71
12. 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)
13. 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) 729–34