Way Stealing: A Unified Data Cache and Architecturally Visible Storage for Instruction Set Extensions

Theo Kluter, Philip Brisk, *Member, IEEE*, Edoardo Charbon, *Senior Member, IEEE*, and Paolo Ienne, *Member, IEEE*

Abstract— **Way Stealing is a simple architectural modification to a cache-based processor that increases the data bandwidth to and from application-specific instruction set extensions (ISEs), which increase performance and reduce energy consumption. Way Stealing offers higher bandwidth than interfacing the ISEs the processor's register file, and eliminates the need to allocate separate memories called architecturally visible storage (AVS) that are dedicated to the ISEs, and to ensure coherence between the AVS memories and the processor's data cache. Our results show that Way Stealing is competitive in terms of performance and energy consumption with other techniques that use AVS memories in conjunction with a data cache.**

Index Terms— **Architecturally visible storage (AVS), customizable processor, data cache, instruction set extension (ISE), Way Stealing.**

I. Introduction

CUSTOMIZABLE processors are specialized by the user to a particular application or application domain through instruction set extensions (ISEs) [1]–[5]. ISEs have been shown to increase performance and energy consumption in single- and multicore processor systems [6], [7], and algorithms that can find the best set of ISEs for an application have matured over the past decade [8]–[17]. Most customizable processors communicate with ISE logic through the register file, which imposes a significant I/O bottleneck and limits the parallelism and speedups that can be achieved through ISEs. To address this concern, ISEs can be augmented with architecturally visible storage (AVS), software-controlled local memories that communicate directly with main memory via direct memory access (DMA) transfers [18]; ISEs can read and write to the AVS memories directly, bypassing the processor's register file and its associated bottlenecks. AVS is placed under software control, much like scratchpad memories. The compiler disambiguates AVS-resident data structures and inserts DMA transfers to load them from main memory into the AVS and store them back when the ISEs are finished. Unfortunately, this creates a coherence problem between the AVS and the processor's data cache, which can be solved using a hardware coherence protocol [19], [20] or a specialized cache controller [19], [21].

This paper introduces a simpler solution, which provides a mechanism by which the ISE logic can read and write data directly to the ways of the processor's cache, bypassing the tag lookup process. We refer to this architectural solution as Way Stealing [19], [22], and this paper explores the architectural possibilities and evaluates its performance in detail.

Way Stealing does not add extra memory elements to the system and, therefore, is memory coherent and consistent in presence of caches; unlike past solutions to the coherence problem for AVS-enhanced ISEs [19]–[21], Way Stealing requires more stringent boundary conditions. These conditions do not endanger its generality, but do require extensive, yet noncritical, changes to the processor pipeline and data cache.

II. Way Stealing

A. Using the Data Memories

Fig. 1(a) shows a RISC, processor pipeline augmented with AVS-enhanced ISEs; the hardware realization of an ISE is called an application-specific custom unit (ASCU), and it executes in parallel with the arithmetic logic unit (ALU) stage of the pipeline. Without loss of generality, if Virtual Ways [19], [21] is applied to ensure coherence between the processor's data cache and the AVS memory, then the tag portion of the AVS memory must be visible to the data cache (to provide coherence), while the data portion is only available to the ASCU.

The basic idea of Way Stealing is to merge the AVS into the data cache. This eliminates the area overhead of instantiating external AVS memories and the need to explicitly maintain coherence between the AVS memories and the processor's data cache [19]–[21]. The drawback is that AVS memories may be multiported if the application benefits from reading and/or writing several data values in parallel; in contrast, each way of the data cache has a single read and write port, which limits concurrency. Additionally, extra instructions must be added to the processor to initialize components of the cache and to load and lock data into specific ways of the cache.

Manuscript received January 8, 2012; revised August 17, 2012; accepted November 27, 2012. Date of publication January 29, 2013; date of current version December 20, 2013.

T. Kluter is with the Bern University of Applied Sciences, EKT, Microlab, 2501 Biel/Bienne, Switzerland (email: theo.kluter@bfh.ch).

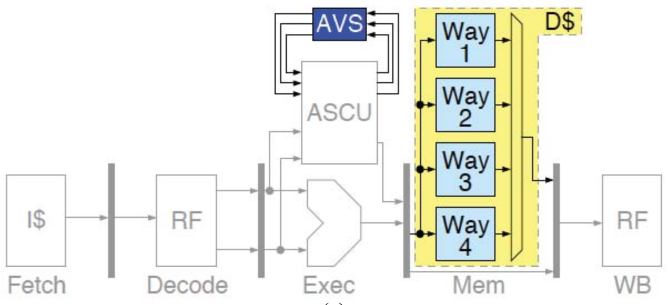
P. Brisk is with the Department of Computer Science and Engineering, University of California, Riverside, CA 92521 USA.

E. Charbon is with the Circuits and Systems Group, Delft University of Technology, 2600 AA Delft, The Netherlands.

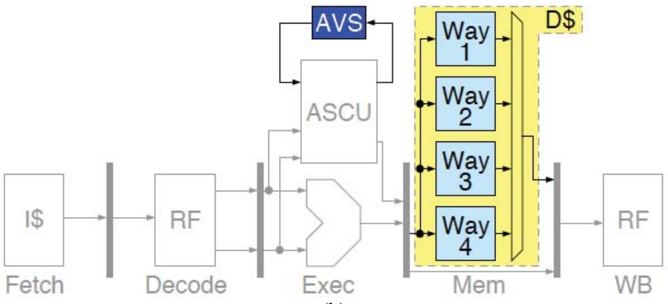
P. Ienne is with the Processor Architecture Laboratory, School of Computer and Communications Sciences, École Polytechnique Fédérale de Lausanne, CH-1015 Lausanne, Switzerland.

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

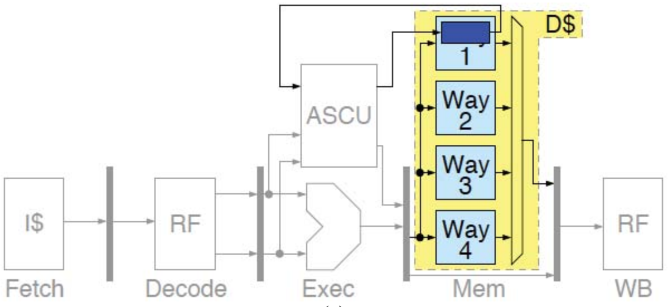
Digital Object Identifier 10.1109/TVLSI.2012.2236689



(a)



(b)



(c)

Fig. 1. (a) Block diagram of an ISE-enhanced processor pipeline with AVS memories [18]–[21]; coherence mechanisms between the AVS and processor's data cache are not shown. (b) AVS memories are single-ported, then they resemble the cache's data memories. (c) AVS memories are single-ported and no larger than the data cache's ways, then they can be "merged."

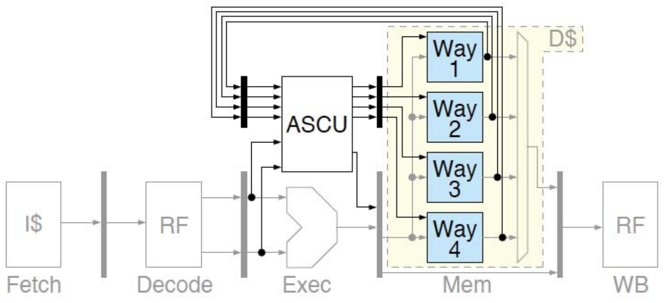


Fig. 2. Way Stealing creates data lanes between the cache's data memories and the ASCU. For an n-way set-associative cache Way Stealing adds n read and n write ports to the ASCU.

This, in turn, has several drawbacks, which include: 1) the compiler must generate a new data cache for each architecture which increases time to market, as cache testing and verification is a complex and time consuming task and 2) creating caches whose internal SRAM memories contain a varying number of read and write ports, potentially breaking symmetry (the ways in a standard data cache are otherwise equivalent). Moreover, the data cache is a critical component in a processor pipeline. Adding extra ports to the cache's data memories could severely impact the critical path, which, in turn, impacts the processor frequency. Both of these reasons render the idea to merge a multiported AVS memory with the data cache impractical.

For systems containing single-ported AVS memories, as illustrated in Fig. 1(b), the preceding arguments against merging the AVS with the data cache become moot, as the ways of the cache are single-ported by construction.

In Fig. 1(c), merging the single-ported AVS memories with the ways of the cache is straightforward, but requires read and write paths between the ASCU and each way. Creating these paths is a one-time effort, and does not impede time to market. The impact on the cache's critical path is negligible.

Way Stealing builds on these observations by: 1) ensuring that all AVS memories detected by an AVS-enhanced ISE identification algorithm [18] are single-ported and 2) ensuring that the data structures contained within them are no larger than the cache's data-memory size. By enforcing these restrictions, it becomes possible to move the AVS into the ways of the data cache. An n-way set associative data cache can provide the illusion of an AVS memory with n banks that the ASCU can read and write concurrently. Fig. 2 shows an example for a four-way set associative data cache.

Although the principle of Way Stealing is simple, its implementation poses some challenges due to the dynamic runtime behavior of the data cache and the location of the data cache in the processor pipeline with respect to the location of the ASCU. We describe these issues in detail as follows.

1) Dynamic Run-Time Behavior: The data cache's state machine controls the contents of the cache's data memories at run-time. The run-time dynamics—i.e., whether a given data element will be present in the cache, which way it will be placed in, etc.—are intractable to determine at compile time; therefore, they cannot be used for compile-time decisions, such as ISE selection. Upon entering a code segment that contains an AVS-enhanced ISE, either: 1) the required data segment does not reside in the data cache; 2) the required data segment resides completely in one way of the data cache (but not necessarily the "right" way from the ASCU's perspective; and/or 3) the required data segment is scattered over the different ways of the data cache. Each of these three effects must be handled properly to guarantee correct execution of the ISE-enhanced application, as discussed in Section II.

2) Location of the Data Cache in the Processor Pipeline: For the ASCU to be able to write the data cache, both components should be in the natural pipeline order. The ASCU computes a value, and one cycle later the results can be written to the cache's data memories; however, for the ASCU to be able to compute in the execute stage (Exec), it must read data from the cache, which are not yet available.

The pipeline in Fig. 2 provides the data two cycles after the custom instruction completes the Exec stage. This problem can be solved by allowing the ASCU to request the data in the decode stage (Decode); however, this impedes proper pipeline behavior, because both the decode and memory (Mem) stages access the data cache at the same time, creating a resource conflict. Section III describes how to modify the processor pipeline to guarantee correct read path execution.

B. Prefetching and Locking

Due to the dynamic run-time behavior of the data cache, no ISE identification algorithm can predict at compile time where a disambiguated data structure will reside in the memory hierarchy. To support Way Stealing, an ISE-enhanced code segment requires that the data structures consumed or produced by the ASCU reside in a specific way of the data cache to guarantee correct execution.

Since the cache can load and evict data, the first requirement can be guaranteed; however, RISC architectures request one data element at a time. Assuming an n-way set associative data cache, the ASCU, in contrast, might request as many as n data elements per cycle, which could induce n concurrent cache misses. Although the data cache can be modified to account for this situation, doing so does not help the compiler to choose which ways of the cache will store the data structures.

The compiler knows which data structures are used in each code segment. Cache locking techniques [23], [24] exploit this knowledge for better predictability in real-time systems. Way Stealing requires a different form of prefetching that selects a specific way for each data structure; fortunately, the required modifications to standard prefetching techniques are fairly simple to understand and implement.

A conventional prefetch instruction contains a pointer to the data structure to be prefetched, along with its size, and relies on the cache's replacement policy, under dynamic hardware control, to place the data structure in the cache. Way Stealing requires a prefetch instruction that specifies the desired way, thereby overruling the replacement policy. This raises a problem if portions of the data structure happen to reside in the cache in ways other than the one specified by the prefetch instruction at the time the prefetch instruction is issued. This dynamic placement is not guaranteed to match the way chosen by the compiler and specified in the prefetch instruction.

Fig. 3 shows the prefetching of a data structure of the size of three cache lines into Way 1. In Fig. 3(a) the data does not reside in the cache and is loaded using the normal load/evict procedure. Fig. 3(b) depicts the situation where the data is not only present in the cache but also in the correct way. Finally, Fig. 3(c) shows the most complicated situation, in which the data is present in the cache but in the wrong way.

In Fig. 3(c), the cache state machine can apply two policies to guarantee the location of the data by either 1) copying the data from Way 2 to Way 1 and invalidating the copy in Way 2 or 2) swapping the contents of the corresponding lines in the two ways. The former method is simpler, but may negatively impact performance, as 1) the cache line in Way 1 may contain data in the dirty state requiring a write back to main memory and/or 2) the data in the cache line residing in Way 1 might be required shortly after the prefetch instruction. Swapping the corresponding cache lines avoids these effects.

Now that the location of the data structure in the cache has been established, the run-time behavior of the cache causes another problem. Due to cache misses, the prefetched data structure may be evicted from the cache after it has been loaded into the data cache, but before the ISE executes. To prevent this situation from occurring, the compiler can lock the data structure into a specific way of the data cache, as long as the data cache supports locking. Using prefetching in conjunction with locking ensures that the data is present in the appropriate way and that it stays there until it is unlocked. The prefetch-and-lock instruction guarantees correct execution of the ISE-enhanced code segment; and an unlock instruction resumes normal cache operation.

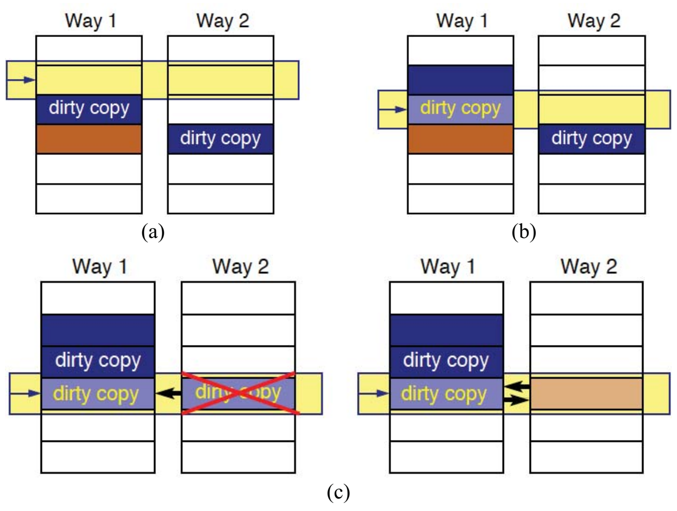


Fig. 3. Way Stealing extends a prefetch instruction with predefined way enforcement. The data structure should be loaded into Way 1. Three situations may occur (a) requested line is not present in the cache, (b) requested line is present in the cache and in the correct way, and (c) requested line is present in the cache, but in the wrong way: the cache state machine can either invalidate the line currently present in Way 1 (left) or swap the lines in the two ways (right).

A subtle problem can occur when all ways of the cache are locked for a given number of cache lines. If a cache miss occurs due to aliasing on a locked cache line, then eviction becomes impossible, and the new line cannot be loaded into the cache. One possibility is for the compiler to ensure that at least one way of the cache is unlocked for each line; the alternative is for the cache state machine to treat this request as uncacheable, and forward the result directly to the processor without storing the data in the cache. The former method requires the compiler to know the cache size and associativity to determine if the deadlock situation may occur; the latter method removes this requirement, but does so at a high cost, all requests that alias cache lines where all ways are locked must now go directly to main memory.

C. Data Cache Lookups

Due to the location of the ASCU in the pipeline with respect to the data cache (Fig. 2), the ASCU requires a request for the data already in the decode stage. Fig. 4(a) shows that this early request can lead to conflicts if the memory stage is executing a load or store instruction. The data cache contains single-ported ways that can only serve one request per cycle.

This conflict can be resolved in hardware by serializing the accesses to the data cache or by the compiler through instruction rescheduling or the insert of no operation (NOP) instructions. Serialization requires hardware collision detection

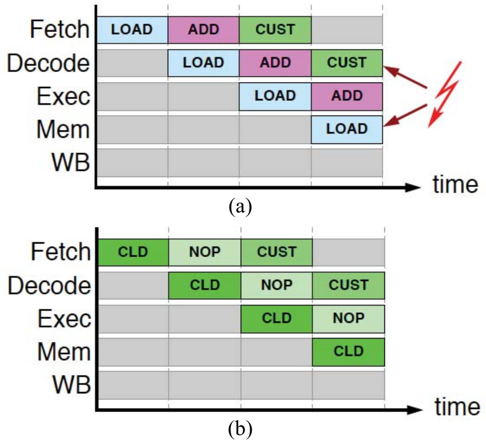


Fig. 4. (a) ISE (CUST) must fetch data from the cache in the decode stage (Decode) to be able to operate on it in the execute stage (Exec). The request of the ISE collides with the load instruction (LOAD) that is in the memory stage (Mem). A single-ported data cache cannot service both requests. (b) ISE (CUST) is extended to three instructions. The CLD instruction instructs the data cache to load the data when it reaches the memory stage (Mem). The NOP ensures pipeline correctness such that the data is available to the custom instruction when it enters the execute stage (Exec).

circuitry and serializer similar to features that are found in out-of-order superscalar processors. The alternative is to introduce new constraints into the compiler's scheduling algorithm.

Although serialization may seem preferable, one important aspect is overlooked. The early request requirement of the ISE requires a lookup request path from the pipeline's decode stage to the data cache. This request path may require long wires due to the geometric arrangement of the processor's components on the chip. These long wires could severely impact the critical path, impeding the processor frequency; therefore, our preference is to rely on the compiler, rather than to provide an early request service in hardware.

To avoid the early request, the compiler replaces each ISE with a three-ISE sequence, as shown in Fig. 4(b), which issues a custom load (CLD) instruction followed by an NOP and then the ISE (CUST). This extension has no impact on the compiler and its scheduler, as the three instructions execute atomically. No hardware changes are required, as data cache requests are scheduled in pipeline order, enforcing memory consistence.

The prefetch-and-lock instruction, described in the previous section, which must complete before an ISE can access the data, includes the following fields: 1) the physical memory address from which the data will be loaded (in principle, variations of the CLD instruction could support different addressing modes); 2) an identifier to specify the way of the cache that holds the data; 3) the size of the data structure in bytes; and 4) the type of the data structure (e.g., char, short, long, etc.). The CLD instruction specifies the way from which a datum will be read, and implicitly increments the AGU's read pointer.

The prefetch-and-lock instruction loads data structures into specific ways of the data cache; however, the data may not be at the same physical index, as shown in Fig. 5. A cache addresses all ways using the same index. To support different addressing schemes, as required by the ISEs, each way of the

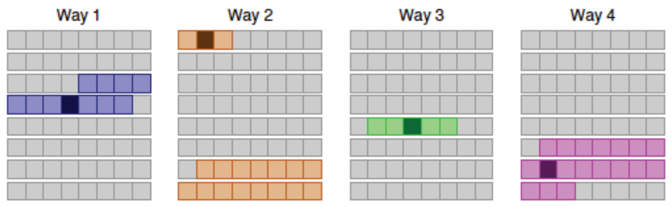


Fig. 5. Block diagram of the data memories of a four-way set-associative cache containing four prefetched and locked data structures. The elements of these data structures required by a custom instruction have been highlighted. The physical index of each of these highlighted elements might be different.

cache requires an individual addressing mode. The number of address bits required to index each way individually is unlikely to fit into the CLD instruction. To compensate, we introduce an address generation unit (AGU) for each way of the cache (see Section III-B for details). The AGU provides an efficient mechanism to compute offsets required for array traversals. The AGU's complexity depends on the addressing schemes that it supports.

Fig. 6 shows an instruction sequence that illustrates the Way Stealing process for a Discrete Cosine Transform (DCT) operation.

D. Impact on ISE Identification Algorithms

ISE identification algorithms search for an ISE that maximizes a merit function, $M(C)$, where C is the set of assembly-level operations that are included in the ISE [8]–[17]. C has a software cost, $S(C)$, which is the estimated latency of executing the operations comprising C in software, and a hardware cost, $H(C)$, which is the estimated latency of executing the operations comprising C as an ISE. The merit function estimates the latency savings, $M(C) = S(C) - H(C)$.

For AVS-enhanced, $H(C)$ should include an estimate of the overhead due to data transfer requirements, such as DMA transfers [18], [19]. Here, we discuss the overhead incurred by the prefetch-and-lock instructions that are required to support Way Stealing.

1) ASCU Store Operations: Fig. 2 shows that store operations involve writing to a pipeline register; the hardware cost is therefore the setup time of the pipeline register. Alternative schemes provide local memories for the ASCU write to these memories directly [18]–[21].

2) ASCU Load Operations: Fig. 2 shows that the read operation does not occur in pipeline order. The proposed solution is to build a meta-ISE consisting of the CLD, NOP, and CUST instructions shown in Fig. 4(b). ISE identification algorithms must model this adjustment. Moreover, looking at Fig. 2, the ASCU reads data directly from a register, not the ways of the data cache; consequently, the cost should also include the clock-to-valid output delay of the register.

3) Communication Cost: Unlike other techniques that provide the ASCU with local memories [18]–[21], Way Stealing does not require explicit data transfer instructions to move data to and from the AVS; nonetheless, the communication cost is nonzero, as shown in Fig. 3. The first situation shown in Fig. 3(a) is a standard cache miss that would occur during normal software execution. In Fig. 3(b), the data structure is in the correct way of the cache, so the cost is zero for either

```
/*  Initialize the AGUs.
    Set all Read/Write offset pointers to 0.
    Set all Stride registers to 1, Mask registers to 0xFFFFFF.
*/
asm volatile ( "l.mtspr  r0,r0,0xF010");
asm volatile ( "l.mtspr  r0,r0,0xF011");
asm volatile ( "l.mtspr  r0,%[in1],0xF012"::[in1]"r"(1));
asm volatile ( "l.mtspr  r0,%[in1],0xF013"::[in1]"r"(-1));

/*  Prepare the AGU to load the DCT buffer (Non-blocking)
    Set Mask register of Way 1 to the size of the DCT buffer
    Allocate and lock the workspace:
        Load the 128-byte workspace buffer into Way 1
        Lock Way 1
        Set the base-address of the DCT buffer Way 1's AGU
*/
asm volatile ( "l.mtspr  r0,%[in1],0xF00B"::[in1]"r"(DCTSIZE-1));
asm volatile ( "l.allocl %[in1], %[in2],
                    w0,1,1"::[in1]"r"(workspace),[in2]"r"(128));

for (elemr = 0; elemr < DCTSIZE; elemr++) {
    elemptr = sample_data[elemr] + start_col;

    /*  Load the line of the picture data (indexed by "elemr") into
        Way 3 and lock the data. The base address of the data is
        set automatically in the base-address of the GPU. This
        call is non-blocking
    */
    asm volatile ( "l.allocl %[in1], %[in2],
                        w2,0,0"::[in1]"r"(elemptr),[in2]"r"(DCTSIZE));

    /*  CLD instruction: specify the way from which the data is
        read;  this implicitly increments the AGU's read pointer.
    */
    asm volatile ( "l.cld w2\nl.nop\nl.nop" );

    /* Perform the DCT and unlock */
    asm volatile ( "l.cdsp %[in1],0,1,0xFF,0"::[in1]"r"(DCTSIZE));
    …
}
```

Fig. 6. Code Sequence Illustrating the Way Stealing Process for an IDCT ISE. All Way Accesses use the AGU's Addressing Mode: Base_Addr + (Read/Write_Counter+ Stride) & Mask. Instructions Added to the Processor are Specified Using GCC's Extensions for Inline Assembly Instructions Using the C Syntax ("Asm Volatile").

software execution or an ISE. In Fig. 3(c), the data resides in the wrong way of the cache and needs to be moved into the correct way. The likelihood that this situation occurs is runtime dependent and cannot be determined accurately at compile time. At best, profile-guided analyses can be used; however, the application may exhibit high variance from one dataset to another.

Our cost model makes the conservative assumption that the data structure resides entirely in the wrong way of the cache.

III. Architectural Impact of Way Stealing

The discussion in this section assumes that the data cache is four-way set associative, and has been enhanced with prefetching and locking mechanisms. Fig. 7 shows a simplified block diagram of the data cache's read and write paths. Way Stealing requires two changes to the conventional cache architecture, which are discussed in detail in this section.

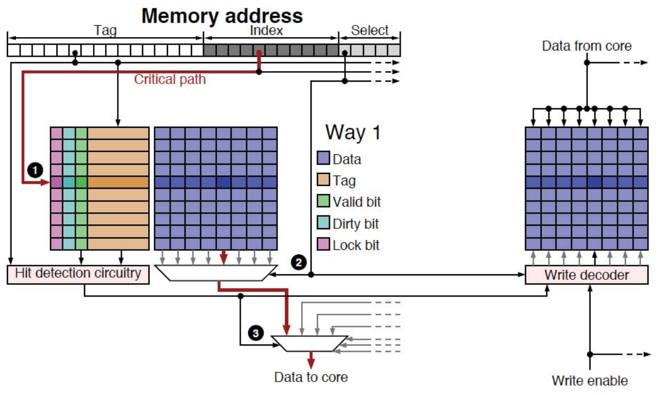


Fig. 7. Simplified block diagram of a prefetch and locking-enhanced four-way set-associative data cache. One of the four ways is shown. The cache's read path is shown to the left, and the write path to the right. For clarity, the data memory is shown twice. The index signal (1) is used to select a cache line. The select signal (2) selects one datum of the data memory and the tag is compared with the selected entry in the tag memory to determine a hit or miss. The hit signal (3) selects which datum is forwarded to the processor. The timing critical path is shown with red arrows.

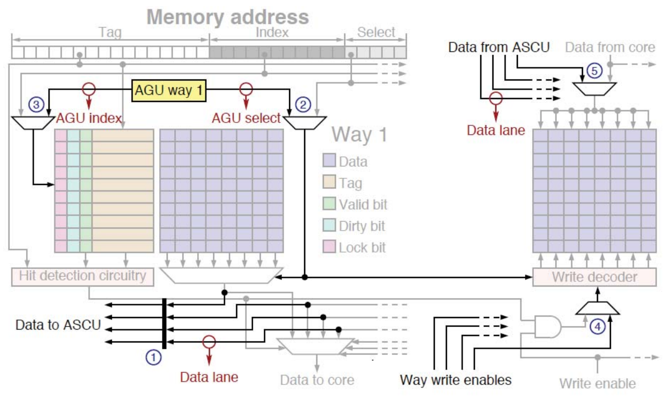


Fig. 8. Simplified block diagram of the modifications to the cache's read and write path as required for Way Stealing. The registers (1) protect the cache's critical path against the load due to long lines between the cache and the ASCU. The multiplexers (2) and (3), in conjunction with the AGU, provide flexibility in the selection of any datum within the cache's ways. Although multiplexer (3) is in the cache's timing critical path, its influence is negligible. Multiplexer (4) enables the simultaneous writing of m out of n ways. Finally, multiplexer (5) selects between the data on the ASCU's data lanes and the data coming from the processor.

A. Creating Data Lanes

Fig. 2 shows that Way Stealing creates data lanes between the cache's data memories and the ASCU. Fig. 8 shows the required changes to the cache's read and write paths to establish these data lanes.

1) Read Data Lanes: The ACSU connects to each output multiplexer of the cache's data memories, creating four data lanes. This extends the cache's critical path (Fig. 7) due to its capacitive load. Longer data lanes have longer wires, and thus have a greater influence on the critical path. Adding pipeline registers after the data memories' output multiplexer limits this influence, as shown in Fig. 8. The input load of a register is small, which limits its influence on the cache's critical path; however, this adds an extra pipeline stage to the read path from

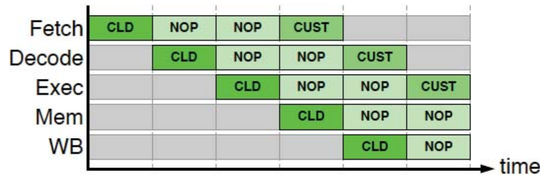


Fig. 9. Way Stealing introduces an extra register into the cache's read path. The three-cycle ISE, shown in Fig. 4(b), requires an extra NOP.

the cache to the ACSU, which adds an extra cycle (NOP) to each ISE invocation, as shown in Fig. 9.

2) Write Data Lanes: Pipeline registers are not introduced to the data cache's write path; direct wires connect the registers and ASCU, as shown in Fig. 2. The delay of the data lanes plus the multiplexer is a small fraction of the processor's critical path. Some modifications to the write path are still required. A conventional data cache (Fig. 7) writes the data into one the ways, as specified by the hit signal. A cache that is modified for Way Stealing, in contrast, can support $m \leq 4$ simultaneous ASCU writes; a multiplexer is added before the write decoder (Fig. 8). By activating m way-write-enable signals, m data lanes are simultaneously written to the data memories.

B. Flexible Addressing

In Fig. 5, the elements of the data structures required by the ASCU may not reside at the same location in the different ways of the data cache; however, a conventional data cache (Fig. 7) addresses the ways by using the index of the memory address to enable the same cache line in all the ways and uses the select line to choose one datum of the enabled cache lines. To avoid this restriction, Way Stealing introduces an AGU for each way of the data cache (Fig. 8). The AGU (Fig. 10) provides an index to enable a cache line in the cache's data memory and a select line to choose one datum to be read from or written to the enabled cache line. Two multiplexers select between normal cache operation and Way Stealing. One of them is in the cache's critical path; however, its delay is far less than the total critical path delay, rendering its influence insignificant.

The AGU needs to know the data structure's read address if the ASCU consumes the data, or its write address if the ACSU produces the data. Under software execution, the processor provides the memory address, which is often determined by adding an index to the start address of the data structure. The prefetch-and-lock instruction contains the start address of the data structure, its size, and the way into which it is prefetched; thus, this instruction can initialize the AGU in preparation to receive the data structure. Additionally, the AGU performs address calculations in lieu of the ISE or software.

The AGU requires fewer bits than a full memory address; the data memory inside one way of the cache only requires the number of bits associated with the memory address' index and select portion shown in Fig. 7. For a 2-kB four-way set-associative cache the AGU requires $\log_2(2048/4) = 9$ bits. As a consequence, the extra area of the AGUs (one per way) is negligible compared to the total cache area.

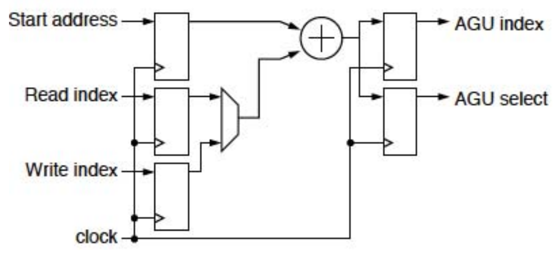


Fig. 10. AGU architecture. Depending on the ASCU's operation (read or write), the multiplexer selects the proper index to calculate the AGU index and AGU select values.

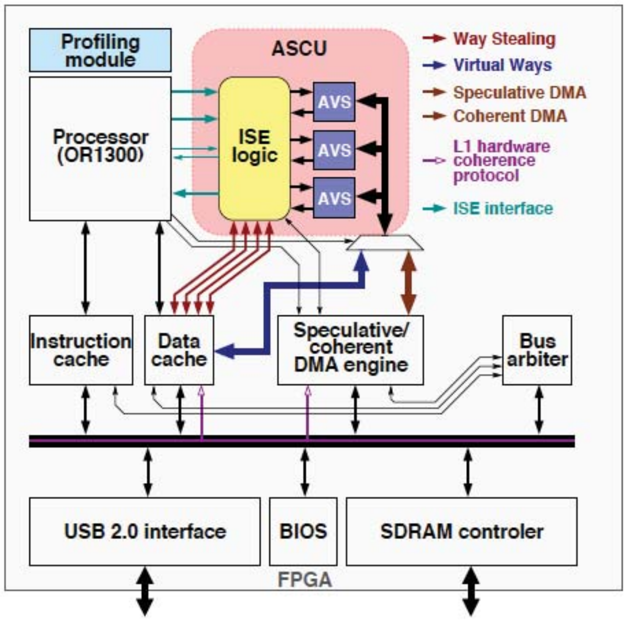


Fig. 11. Block diagram of the system architecture inside the FPGA. The ASCU is automatically generated for each benchmark. The other components are inserted based on a system template.

The AGU in Fig. 10 is simple; enhanced AGUs that support more complicated addressing modes are also possible, but have not been investigated in the context of this work.

IV. Experimental Setup

A. FPGA-Based Emulation Platform

We developed a field-programmable gate array (FPGA) soft processor system that was used for experiments. The platform runs on an internally built printed circuit board that contains a Xilinx Virtex II FPGA (XC2V8000-5FF1517C), 32 MB of SDRAM, and a Cypress FX2-based USB 2.0 connector. An internally developed configurable soft processor runs on the FPGA; Fig. 11 shows the system architecture. The processor, the OR1300, is a 32-bit six-stage RISC pipeline that implements the OpenRISC instruction set; it includes a 1-cycle multiplier, 64-bit pipelined MAC, and 32-cycle hardware divider. The ISE interface is not compatible with the OpenRISC instruction set; instead, we adopt the ISE interface used in Altera's Nios II soft processor.

The soft processor system also includes application-specific ISEs for code acceleration, DMA transfer instructions to support Coherent and Speculative DMA [19], [20], AVS-cache data transfers to support Virtual Ways [19], [21], prefetch and

TABLE I
INSTRUCTION CACHE CONFIGURATION PARAMETERS

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

TABLE II
DATA CACHE CONFIGURATION PARAMETERS

Parameter	Values
Cache size	2 kbyte, 4 kbyte, 8 kbyte, or 16 kbyte
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

lock/unlock instructions to support Way Stealing [19], [22], and software cache flush instructions.

The OR1300's instruction and data caches are configurable by software using special purpose registers (SPRs). They both have a 32-byte line size and support software flushing. The data cache supports up to three outstanding requests. Tables I and II list the configuration options for the caches.

The platform supports automatic design space enumeration for instruction and data cache parameters. A script running on the host PC repeatedly reconfigures the caches in the system to support every combination of instruction and data cache configuration. Each benchmark and dataset is executed on each configuration, and performance and energy estimates are obtained. The caches are flushed between runs to ensure a cold start each time. This allows us to automatically evaluate the impact of ISEs across a variety of processor configurations.

The system bus is atomic and supports one transaction at a time, using a first-come first-serve scheme with priority resolution (the data cache has the highest priority, and the DMA engine has the lowest). Transactions that cease activity over a period of time are aborted to prevent deadlocks. The bus supports the MSI and MESI hardware coherence protocols, which are required for Coherent and Speculative DMA [19], [20]. A single-processor system that implements Way Stealing does not require hardware coherence.

The SDRAM controller supports emulation of the processor-to-memory distance, $D_{\text{pm}} = f_{\text{processor}}/f_{\text{memory}}$, where $f_{\text{processor}}$ and f_{memory} are, respectively, the processor and memory frequencies. The emulator assumes that all bus transactions are equal to the cache line size (32 bytes) and does not model SDRAM refresh cycles. Eight cycles are required to transfer the cache line. Given the SDRAM access overheads reported in Table III, a write burst requires $N_{\text{write}} = 12$ memory cycles, and a read burst requires $N_{\text{read}} = 15$ memory cycles. To emulate the processor-to-memory distance, the emulator loads a counter with the value of $D_{\text{pm}}N_{\text{write}}$ or $D_{\text{pm}}N_{\text{read}}$, depending on the type of access; all future memory transactions are stalled until the counter reaches zero.

B. Energy Model

The energy model used in our experiments includes the instruction and data caches and external SRAM; the bus is not modeled. We observed that the AVS memories tend to be small and their contribution to overall energy consumption is negligible; we omit them from our model. We use CACTI 5.3 rev. 174 [25] for a 90-nm technology node to estimate the energy of cache read and write accesses, E_{read} and E_{write} respectively; these values depend on the parameters of the cache (e.g., size, associativity, etc.). CACTI 5.3 also provided E_{read} and E_{write} estimates for SDRAM reads and writes.

TABLE III
SDRAM OVERHEAD FOR EACH BURST

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

The profiling module employs 64-bit counters to count the following events: the number of instruction caches fetches ($N_{\text{I\$-fetch}}$) and misses ($N_{\text{I\$-miss}}$), the number of data caches reads ($N_{\text{D\$-read}}$), writes ($N_{\text{D\$-write}}$), and misses ($N_{\text{D\$-miss}}$), and the number of SDRAM accesses (N_{SDRAM}).

1) Instruction Cache Dynamic Energy: The processor fetches instructions from the instruction cache; one cache line (8 bytes) is transferred at a time on the bus, and is only written to the instruction cache on a miss. The energy is estimated as

$$E_{\text{I\$}} = N_{\text{I\$-read}} \cdot E_{\text{read}} + 8 \cdot N_{\text{I\$-miss}} \cdot E_{\text{write}}. \tag{1}$$

2) Data Cache Dynamic Energy: Load and store instructions read and write the data cache, respectively. On a miss, a new line (32 bytes) is written; the bus transfers one cache line at a time. Coherent and Speculative DMA [19], [20] include a hardware coherence protocol. Let $\alpha = 1$, if the protocol is enabled, and $\alpha = 0$ otherwise, and E_{tag} denotes the energy of a snoop lookup. The data cache dynamic energy is estimated as

$$\begin{aligned} E_{\text{I}} &= N_{\text{D }-\text{read}} \cdot E_{\text{read}} + (N_{\text{D }-\text{write}} + 8 \cdot N_{\text{I }-\text{miss}}) \\ &\quad \cdot E_{\text{write}} \quad + \alpha \cdot N_{\text{SDRAM}} \cdot E_{\text{tag}}. \end{aligned} \tag{2}$$

3) SDRAM Dynamic Energy: SDRAM dynamic energy is estimated as follows:

$$\begin{aligned} E_{\text{SDRAM}} &= 8 \cdot N_{\text{SDRAM}} \cdot \frac{E_{\text{read}} + E_{\text{write}}}{2} \\ &= 4 \cdot N_{\text{SDRAM}} \cdot (E_{\text{read}} + E_{\text{write}}). \end{aligned} \tag{3}$$

4) Leakage Energy: Let $P_{\text{leak,I\$}}$, $P_{\text{leak,D\$}}$, and $P_{\text{leak,SDRAM}}$ be power estimates provided by CACTI 5.3 for the caches and SDRAM, respectively, and let $T_{\text{benchmark}}$ be the execution time of the benchmark. Total leakage power and energy estimates are

$$P_{\text{leak}} = P_{\text{leak,I\$}} + P_{\text{leak,D\$}} + P_{\text{leak,SDRAM}} \tag{4}$$

and

$$E_{\text{leak}} = T_{\text{benchmark}} \cdot P_{\text{leak}}. \tag{5}$$

C. Compiler

We use an internally developed compiler, Clarity, to perform ISE identification and ASCU generation. Clarity implements the ISE identification algorithm proposed by Pozzi *et al.* [10] and uses extensions introduced by Biswas *et al.* [18] to identify

AVS-enhanced ISEs. Clarity generates a VHDL specification of each ASCU (the hardware realization of each ISE) and produces an annotated C language version of the program with macros to represent ISEs.

We manually insert instructions for DMA transfers (Speculative and Coherent DMA [19], [20]), cache-to-AVS data transfer instructions (Virtual Ways [19], [21]), and prefetech and lock/unlock instructions (Way Stealing [19], [22]). For Way Stealing, we manually replace each ISE invocation with the four-instruction sequence shown in Fig. 9.

The annotated C program is cross-compiled using a gcc 3.4.4 toolchain based on binutils 2.16.1 targeting the OR1300. The cross compiler was extended to support all instructions added to the OR1300. The host PC loads the cross-compiled program onto the FPGA board and executes the application.

D. Benchmarks and Experimental Flow

We use the EEMBC DENBench (Digital Entertainment) suite for experimental evaluation [26]. DENBench has complete applications, not kernels, which makes it a good choice for our implementation. The OR1300 does not run an operating system, and uses the test harness provided by EEMBC instead. The experimental procedure is as follows.

1) Correctness Check: All implementations (i.e., an OR1300 instance with benchmark-specific ISEs) were validated using the CRC check provided by the EEMBC test harness; all results reported here pass the CRC check.

2) Reference Configuration: We executed each benchmark and dataset on a standard OR1300 instance without ISEs, and enumerate all instruction and data cache configurations. We select the configuration that achieves the minimum energy consumption as a reference configuration. Our first set of experiments (Figs. 12 and 13) use the reference configurations for two benchmarks (CJPEG V2 and MP3 player) and all of the datasets provided by EEMBC.

3) Baseline: The baseline is pure software execution without ISEs; for all benchmarks and datasets, the performance (runtime) and energy consumption of software execution are normalized to 1. The performance and energy consumption of the other system configurations are normalized to the baseline.

4) non-AVS-Enhanced ISEs: We generate ISEs that do not include memory access instructions using the algorithm described by Pozzi *et al.* [10]. These ISEs communicate with the processor's register file, rather than AVS memories.

5) Noncoherent AVS Architectures: We reimplemented the noncoherent AVS-enhanced ISE scheme proposed by Biswas *et al.* [18]. All implementations that we tested failed the CRC check or experienced coherence errors that affected control flow, causing early termination with incorrect results. We do not report experimental results for this scheme.

6) Coherent DMA: [19], [20] Coherent DMA integrates AVS memories into the OR1300's hardware coherence protocol. Data structures are divided into segments the size of the cache line, and a DMA controller transfers each segment from main memory into the AVS. The coherence protocol ensures that: 1) modified data in the cache is written back to main memory before it is transferred into the AVS memory; 2) a write to the AVS memory invalidates any copy of the data in the cache; and 3) DMA transfers from the AVS memory back to main memory must finish before the processor can read the data.

7) Speculative DMA: [19], [20] It is an extension to Coherent DMA that suppresses unnecessary DMA transfers. For example, if the AVS memory already contains a valid copy of a data structure, then there is no need to overwrite it with a second DMA transfer. Similarly, DMA write-backs to main memory are suppressed until the processor explicitly accesses the data.

8) Virtual Ways: [19], [21] The cache controller is modified to ensure coherence with the AVS memory using a reduced form of inclusion: 1) data is transferred from the L1 data cache into the AVS memory under software control; 2) writes to the AVS memory do not update copies of the data in the cache hierarchy; 3) processor read requests issued to the cache must check to see if the AVS memory contains a modified copy of the data; 4) processor writes to the cache invalidate copies of the data in the AVS memory; and 5) if the AVS memory contains a modified cache line, write-back requests must update the cache's copy of the line. This approach sidesteps the overheads that Coherent and Speculative DMA incur due to their usage of a hardware snoopy coherence protocol.

9) Architectural Enumeration: Figs. 14 and 15 enumerate the instruction and data cache configuration space for two benchmarks and datasets; we report the relative performance and energy of each system configuration (software, non-AVS-enhanced ISEs, AVS-enhanced ISEs with Coherent and Speculative DMA, Virtual Ways, and Way Stealing) for each cache configuration that we consider. We also vary the processor-to-memory distance, observing its impact on different ISE/AVS/coherence architectures.

V. Experimental Results

A. CJPEG V2

Fig. 12 presents experimental results for the CJPEG V2 benchmark across seven datasets. Both the processor and main memory run at 100 MHz. The overhead of the DMA transfers limits the speedups that Coherent and Speculative DMA can achieve, while increasing overall system energy consumption; Virtual Ways achieves greater speedups and consumes less energy than Coherent and Speculative DMA.

Coherent and Speculative DMA outperform Way Stealing by 3%, while Virtual Ways outperforms Way Stealing by 8%. The main reason for this performance loss is due to the DCT kernel. The best parallel implementation of the DCT kernel performs eight simultaneous read and write accesses to the same data structure. Speculative and Coherent DMA and Virtual Ways allocate an application-specific 64-element AVS memory (effectively a register file) with eight read and eight write ports. In contrast, Way Stealing stores the data structure in one way of the cache, which limits the concurrency of the memory accesses. If we were to limit the AVS memory to have one read one write port, then the performance would be comparable across the different architectures.

Compared to the baseline, Coherent and Speculative DMA incur an additional energy penalty, due to the overhead of

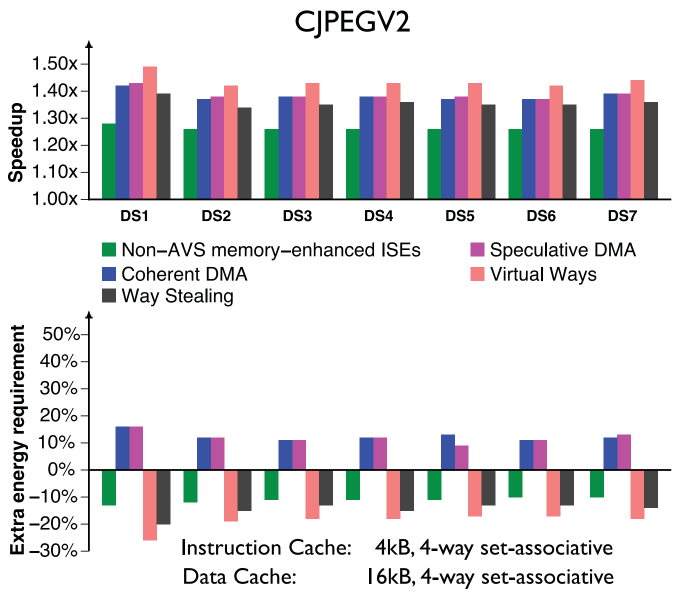


Fig. 12. Performance and energy consumption for the ISE-enhanced architectures, normalized to software execution, of CJPEG V2 across seven datasets, using the reference cache configuration.

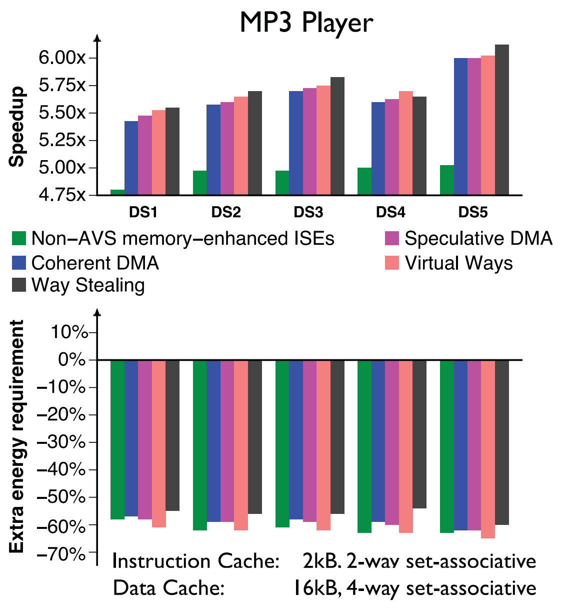


Fig. 13. Performance and energy consumption for the ISE-enhanced architectures, normalized to software execution, of MP3 player across five datasets, using the reference cache configuration.

the hardware coherence protocol. Virtual Ways and Way Stealing both reduce the energy consumption compared to the baseline. Virtual Ways achieves a greater energy reduction than Way Stealing because concurrent ISE accesses to the multiported AVS memory reduce the number of instruction fetches, which significantly impacts energy consumption [7], [27]. If we were to limit the AVS memory to have one read and one write port, then the energy disparity between Virtual Ways and Way Stealing would most likely be eliminated; Coherent and Speculative DMA would still consume more energy due to the overhead of the snoopy coherence protocol.

Using ISEs without AVS achieves lower speedups than the AVS-enhanced architectures, and more modest reductions in energy consumption than Virtual Ways and Way Stealing.

B. MP3 Player

Fig. 13 presents experimental results for the MP3 player benchmark across five datasets. Way Stealing achieves the highest speedup for four of the five datasets. For Dataset 4, Virtual Ways achieves a marginally larger speedup than Way Stealing.

Way Stealing achieves smaller reductions in energy compared to the other ISE-enhanced architectures, including ISEs without memory access. This extra energy is due to the cost of swapping data between ways of the cache, as shown in Fig. 3(c), which is integral to prefetching and locking.

For MP3 player, Coherent and Speculative DMA perform much better in terms of both speedup and energy consumption compared to CJPEG V2. The reason for this is spatial locality. In CJPEG V2, data was transferred from the data cache to the AVS, and then back to the AVS through main memory. In contrast, the MP3 player benchmark has less spatial locality, as the data resides in main memory. Coherent and Speculative DMA, Virtual Ways, and Way Stealing must load data from main memory (either into the AVS or a preselected way of the cache), and thus incur comparable communication costs.

Using ISEs without AVS achieves significantly lower speedups than the AVS-enhanced architectures, although the improvements in energy consumption are comparable.

C. Design Space Enumeration

Figs. 12 and 13 present experimental results for a system where the processor and memory frequency are identical using only the reference cache configurations. In Figs. 14 and 15, we enumerate the instruction and data cache design space for each ISE-related architectural configuration, and vary the processor frequency while holding the memory frequency constant at 100 MHz; results are reported for processor frequencies of 100, 500, and 900 MHz. This experiment observes how other architectural parameters affect the performance and energy consumption of Coherent and Speculative DMA, Virtual Ways, and Way Stealing.

Fig. 14 presents results for CJPEG V2 on Dataset 1; Fig. 15 presents results for MP3 player on Dataset 5. Similar plots for the other datasets have been obtained and the results are comparable [19]; they have been omitted from this paper due to space limitations.

For each CPU and memory frequency configuration of Figs. 14 and 15, each data point is normalized to the reference configuration under software execution (no ISEs), whose normalized (energy, performance) value is (1.0×, 1.0×). The absolute performance and energy consumption of all configurations do vary as the processor frequency changes.

For CJPEG V2 Dataset 1 (Fig. 15), the normalized energy consumption of each ISE and AVS coherence mechanism varies significantly across the set of cache configurations; this variance is less pronounced for Coherent and Speculative DMA than others, but lies toward the higher end of the normalized energy spectrum compared to other approaches.

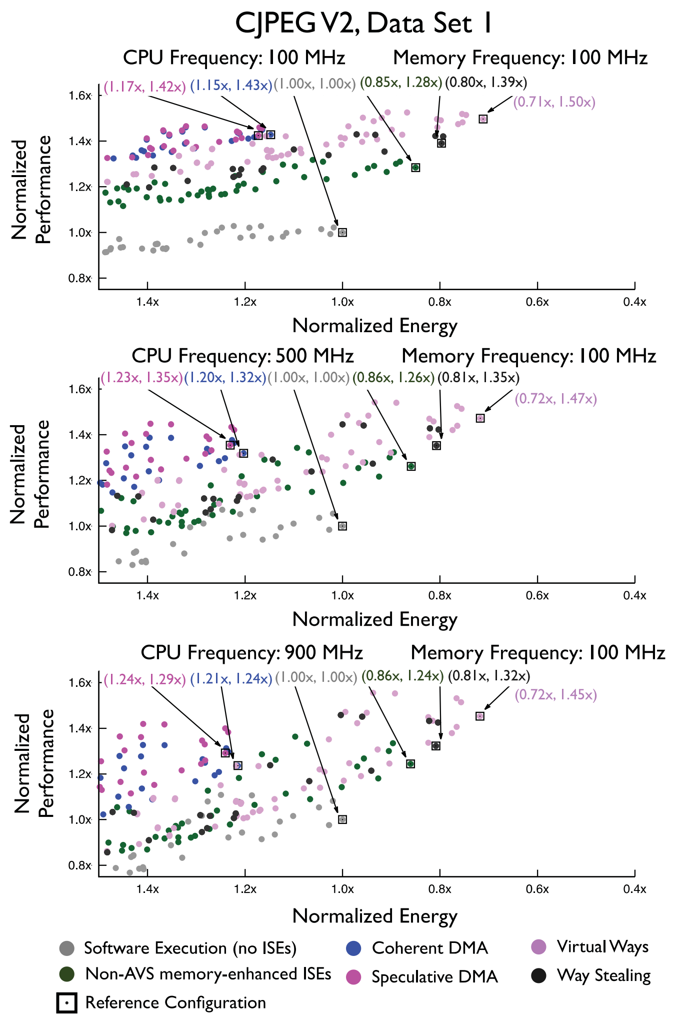


Fig. 14. Design space exploration of the CJPEG V2 compression algorithm for Dataset 1. The processor is run with frequencies of (a) 100 MHz, (b) 500 MHz, and (c) 900 MHz; the memory runs at 100 MHz. The design space enumeration varies the instruction and data cache parameters.

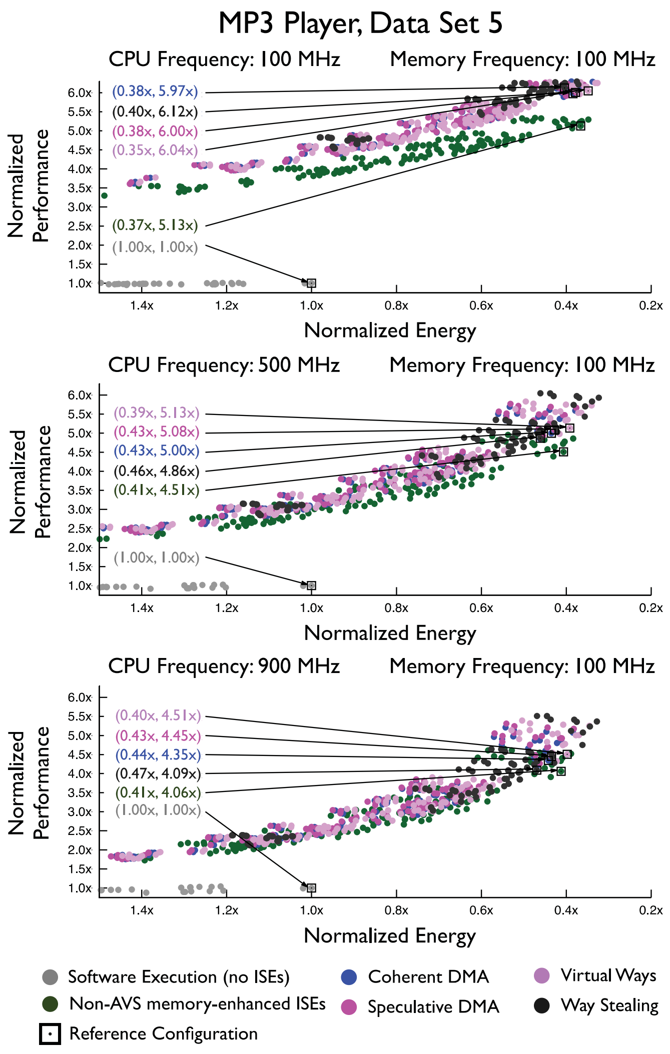


Fig. 15. Design space exploration of the MP3 player benchmark algorithm for Dataset 5. The processor is run with frequencies of (a) 100 MHz, (b) 500 MHz, and (c) 900 MHz; the memory runs at 100 MHz. The design space enumeration varies the instruction and data cache parameters.

When both the processor and memory frequency are 100 MHz (Fig. 15, top), the normalized performance of each ISE and AVS coherence mechanism is relatively stable. For the most profitable cache configurations, Virtual Ways and Way Stealing achieve the highest normalized performance and the lowest energy consumption, with normalized energy ranging from approximately 0.6× to 1.0×. Coherent and Speculative DMA achieve good normalized performance, but their normalized energy consumption lies in the 1.2–1.5× range.

As the processor frequency increases, the variance in normalized performance starts to increase. For example, at 100 MHz, the normalized performance for Way Stealing ranges from approximately 1.2× to 1.4×, and the normalized performance for Virtual Ways ranged from approximately 1.2× to 1.5×. At 500 MHz, the normalized performance for Way Stealing ranges from approximately 1.0× to 1.4×, and from 1.0× to 1.5× for Virtual Ways.

On the other hand, if we look at the reference configurations, the normalized performance for Virtual Ways is 1.50× at 100 MHz, 1.47× at 500 MHz, and 1.45× at 900 MHz; similarly, for Way Stealing, the normalized performance is 1.39× at 100 MHz, 1.35× at 500 MHz, and 1.32× at 900 MHz. Thus, for the best cache configurations, the normalized performance (and energy) results are actually quite robust as the processor-to-memory distance varies.

The worst-performing cache configurations show noticeable degradation in normalized performance as the processor-to-memory distance increases; this essentially demonstrates a greater sensitivity to memory access latency for these configurations. The better-performing caches tend to be larger, and can hold the entire working set of the algorithm, which limits the negative impact of long-latency memory accesses. In contrast, the smaller caches may only hold part of the working set, so portions of the working set are consistently transferred between the cache and main memory. Thus, these configurations exhibit far greater sensitivity to the processor-to-memory distance, regardless of whether Virtual Ways or Way Stealing is used to provide coherence between the data cache and AVS memory.

The best 10–15 data points in Fig. 15 are all either Virtual Ways or Way Stealing, although Virtual Ways does do better. This is because the AVS memories that store different data

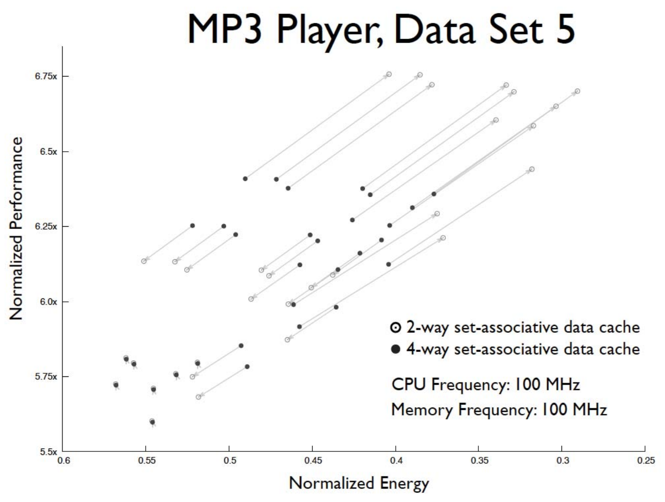


Fig. 16. Performance and energy consumption when the number of stolen ways is limited to two, even for higher associativity caches. At high, fewer ways is preferable, because it limits the number of times that prefetch-and-lock instructions swap data between ways.

structures under Virtual Ways are application-specific and multiported, and therefore, Virtual Ways achieves slightly higher performance than Way Stealing.

The results for MP3 player, Dataset 5 (Fig. 16), show a different trend. Looking at the reference configurations, the normalized performance of Virtual Ways starts at 6.04× at 100 MHz, then drops to 5.13× at 500 MHz, and 4.51× at 900 MHz; similarly, the normalized performance of Way Stealing starts at 6.12× at 100 MHz, then drops to 4.86× at 500 MHz, and 4.09× at 900 MHz. This result indicates that memory access latency has a much greater impact for this benchmark and dataset, compared to CJPEG V2 Dataset 1. This suggests that the application's working set exceeds the capacity of the largest cache configurations considered; therefore, all configurations are sensitive to the processor–memory distance, and exhibit comparable degradation in normalized performance as the processor-to-memory distance increases.

In the case of Way Stealing, the placement of prefetch instructions significantly affects performance as the processor-to-memory distance increases. If a prefetch instruction is placed too close to an ISE that requires the data, then the processor pipeline stalls as it waits for the data to be delivered.

Moreover, locking a data structure into a way of the cache reduces the associativity available to software execution in the general code region of the ISE. Limiting the associativity can lead to more cache misses, each of which requires a cache line load or write-back action.

The speedups obtained by Coherent and Speculative DMA clearly depend on the processor-to-memory distance because they use DMA transfers to stream data into AVS memories [19], [20]; in contrast, the speedups obtained by Virtual Ways do not depend on the processor-to-memory distance, as the data cache communicates directly with the AVS [19], [21].

D. Way Stealing Under Limited Associativity

Way Stealing requires high associativity caches to be effective, especially for ISEs that read and write several data

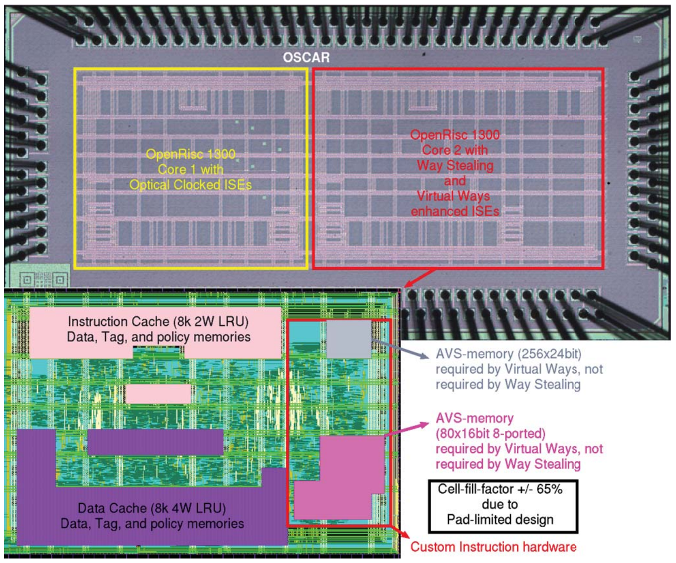

OSCAR
OpenRisc 1300 Core 1 with Optical Clocked ISEs
OpenRisc 1300 Core 2 with Way Stealing and Virtual Ways enhanced ISEs
Instruction Cache (8k 2W LRU) Data, Tag, and policy memories
Data Cache (8k 4W LRU) Data, Tag, and policy memories
AVS-memory (256x24bit) required by Virtual Ways, not required by Way Stealing
AVS-memory (80x16bit 8-ported) required by Virtual Ways, not required by Way Stealing
Cell-fill-factor +/- 65% due to Pad-limited design
Custom Instruction hardware

Fig. 17. Dual-core OR1300 chip featuring AVS-enhanced ISEs fabricated in 90-nm CMOS technology. The layout shows the area dedicated to AVS memories in relation to the rest of the processor.

structures. This experiment considers two-way and four-way set associative data caches, but limits the number of stolen ways to two. Fig. 17 reports the result of this experiment for MP3 player, Dataset 5, where both the processor and memory run at 100 MHz. A two-way set associative data cache means that two ways are available. For the smallest data cache sizes (2 and 4 kB, the lower-left hand corner of Fig. 17), the performance and energy consumption are equal for both two- and four-way set associative caches: in both cases, the data is not present in the caches and must be fetched from main memory; therefore, the performance overhead and energy consumption are equal.

For 8-kB data caches, the four-way set associative organization achieves higher performance and lower energy consumption than the two-way set associative organization; the opposite is true for 16-kB data caches. The issue at hand is the cache replacement policy, which is least recently used (LRU) in Fig. 17. Significant performance and energy overheads may be incurred if the data is present in the cache, but in the incorrect way, requiring the prefetch-and-lock instruction to swap the ways that contain the data, as shown in Fig. 3(c). The impact of these swaps is difficult to predict at compile time. The likelihood of data being present in the correct way becomes lower as cache associativity increases. Thus, for the largest cache sizes, lower associativity leads to better overall performance, as there is much greater likelihood that data will be loaded into the correct way. We repeated the experiment shown in Fig. 17 with an first-in, first-out (FIFO) replacement policy and the results were similar.

One possible remedy, which we do not evaluate, is to create way-specific load instructions that override the cache's replacement policy and deliver data to a statically chosen way. If no bits are available in the load instruction's opcode format to specify the way, then extra opcodes could need to be used, i.e., one instruction for each way. This could be reasonable for set associative caches with up to four ways, but would be prohibitive for caches with higher associativity.

Doing so would eliminate the performance advantage of low-associativity caches at higher capacity levels.

E. Area Overhead

Figs. 13–16 show that Virtual Ways and Way Stealing tended to achieve the best performance and lowest energy consumption compared to the other architectures we considered. Virtual Ways benefitted from the ability to customize multiported AVS memories to the needs of the application, whereas Way Stealing was restricted to memories equal to the size of the ways of the cache, and with one read and one write port. As Way Stealing does not require an external SRAM structure (outside of some minor cache modifications), it retains an area advantage. The number of AVS memories, their size, and the number of ports required for Virtual Ways, is application specific.

A test chip for a separate project (Fig. 17) was fabricated in 90-nm CMOS technology comprising two OR1300 processors, one of which included AVS-enhanced ISEs [28]. The test chip, named "OSCAR," validated a globally asynchronous locally synchronous (GALS) optical clocking scheme that runs ISEs at a higher clock frequency than the processor. Although OSCAR's objective is orthogonal to this paper, the test chip's physical layout provides evidence that the area overhead the AVS memories compared to that of the instruction and data caches can be nonnegligible. The test chip contains two AVS memories, whose collective area is slightly smaller than the data array of an 8-kB two-way set associative instruction cache, and approximately half of the area dedicated to ISEs (logic plus AVS). The contribution to total system area would be more pronounced with more ISEs and AVS memories. In contrast, Way Stealing does not use AVS memories, although its data cache would be slightly larger, as shown in Fig. 9.

VI. Related Work

A. Hybrid Cache-Scratchpad Architectures

Like Way Stealing, several hybrid-reconfigurable cache architectures have been proposed within the past decade that allow the processor to access specific regions of the cache as though they were scratchpad memories. Ranganathan *et al.* [29], for example, argued that associativity should be runtime-configurable to meet the needs of different workloads. One of their proposed configurations provides compiler control of the data/tag SRAM, which shares some principle similarities to Way Stealing. Their evaluation focused on instruction reuse for multimedia applications, which is considerably different from our goal to support AVS-enhanced ISEs.

Intel's buffer-integrated cache (BiC) [30], [31] is a last-level cache (LLC) architecture used for communication processors and on-chip accelerators. Portions of the BiC are reserved as buffers, which are used to transfer data to the accelerators, which can also use the buffers as local stores. The ISEs in customizable processors are tightly coupled with the processor pipeline, and their latencies are generally known *a priori*; in contrast, the accelerators in Intel's architecture are loosely coupled, and notify the processor when their execution is complete. There are also differences in the granularity of data transfers, as the LLC buffers are larger than AVS memories.

Cong *et al.* [32] developed a reconfigurable cache that dynamically remaps scratchpad memory blocks to cache sets with low utilization. This requires additional hardware, as the overhead of managing the remapping process in software would be prohibitive. This cache architecture could be made compatible with way stealing; however, the ASCU has direct connections to each of the ways in the cache, and expects a given data structure to be stored in a particular way. To ensure compatibility, the cache must track the movement of the scratchpad memory within the cache, and crossbars at the ASCU input and outputs would be required to correctly route the data; we suspect that the area and energy overhead of implementing this approach in silicon would be prohibitive.

B. Increasing ASCU Data Bandwidth

Traditional ISEs communicate directly with the register file of the extensible processor [8]–[17]. A typical RISC processor's register file has two read ports and one write port. As a consequence, I/O bandwidth between the processor and ASCU limits the speedup that an ISE can achieve, as its execution must be spread across multiple clock cycles [14], [33], even if it offers significant internal parallelism.

AVS and Way Stealing increase I/O bandwidth: the ASCU can access AVS, or a Way Stealing-enhanced cache, and communicate with the register file at the same time. Other architectural mechanisms can increase ASCU input bandwidth with the processor; unfortunately, they do not increase output bandwidth. In principle, these mechanisms can be used in conjunction with AVS memories and/or Way Stealing.

Cong *et al.* [34] introduced shadow registers, which are placed next to the ASCU and outside of the processor's pipeline. The bitwidth of the instruction set of the processor is extended so that ALU instructions can write to the shadow registers and the processor's register file at the same time. The ASCU can read data from the shadow registers while it communicates with the processor's register file. All ASCU outputs must be written back to the processor's register file.

Jayaseelan *et al.* [35] allow the ASCU to read data from multiple pipeline registers in the execute and write-back pipeline stages, along with the register file. This increases input bandwidth, but imposes stringent scheduling constraints, as the desired instructions that produce the data must be scheduled in the slots immediately preceding the ISE. The input bandwidth increase is dependent on the number of pipeline stages; a five-stage pipeline can provide two additional ASCU inputs through the forwarding path.

Karuri *et al.* [36] designed a clustered register file, which allows the ASCU to read data from several clusters in parallel. The compiler is responsible for distributing the data that will be read by the ASCU across the clusters. In the worst case, the compiler may need to introduce cluster-to-cluster register copy instructions, whose execution times may offset the speedup obtained by the ISEs.

C. ISE Identification Algorithms

Many compiler algorithms have been proposed over the past decade to automatically identify and synthesize ISEs from an application specified in a high-level language [8]—[17]. These algorithms can be extended to identify AVS-enhanced ISEs as described by Biswas *et al.* [18]. The speedup function that guides the ISE identification process must be extended with a reasonable model to account for the cost of transferring data into the AVS memories. This paper provided a cost function that is appropriate for Way Stealing [19], [22]; in his Ph.D. thesis, Kluter [19] provides cost models that are appropriate for Coherent DMA, Speculative DMA, and Virtual Ways.

VII. Conclusion

Way Stealing modifies a processor's data cache so that ISEs can read and write its ways directly, under software control. This sidesteps the traditional tag lookup and hit/miss detection circuitry, provides a low-energy access path, and guarantees deterministic memory access latencies once the data has been loaded. The modifications have an insignificant impact on data cache area and critical path delay. Way Stealing achieves similar performance and energy consumption in comparison with other techniques that instantiate AVS memories in parallel with the data cache—Coherent and Speculative DMA and Virtual Ways—but without the area overhead of the extra AVS memories.

Way Stealing has many other potential uses going beyond ISE enhancement, especially through the exploitation of the AGUs associated with each way. For example, a Way Stealing-enhanced cache could connect to the ALU(s) of an RISC (or VLIW) processor to efficiently implement vector and/or accumulation operations. The AGU could perform all address calculations, freeing up the ALU. The data could stream directly from the Way Stealing-enhanced cache into the ALU and then back, bypassing the register file. Other similar architectural optimizations are also possible.

This paper evaluated Way Stealing as a mechanism to provide high-bandwidth I/O to application-specific ISEs used in configurable processors. The performance and energy consumption of Way Stealing are close to Virtual Ways [19], [21] and better than Coherent and Speculative DMA [19], [20]. Although the best choice of coherence mechanism depends on context and engineering tradeoffs, we strongly believe that Way Stealing is among the better choices, especially for single-processor systems where hardware coherence protocols are not already present in the system.

References

[1] R. E. Gonzalez, "Xtensa: A configurable and extensible processor," *IEEE Micro*, vol. 20, no. 2, pp. 60–70, Mar.–Apr. 2000.

[2] R. E. Gonzalez, "A software-configurable processor architecture," *IEEE Micro*, vol. 26, no. 5, pp. 42–51, Sep.–Oct. 2006.

[3] T. R. Halfhill, "ARC cores encourages 'plug-ins,'" *Microprocess. Rep.*, Jun. 2000.

[4] T. R. Halfhill, "MIPS embraces configurable technology," *Microprocess. Rep.*, Mar. 2003.

[5] T. R. Halfhill, "Tensilica's software makes hardware," *Microprocess. Rep.*, Jun. 2003.

[6] J. Clemons, A. Jones, R. Perricone, S. Savarese, and T. M. Austin, "Effex: An embedded processor for computer vision based feature extraction," in *Proc. 48th Design Autom. Conf.*, San Diego, CA, Jun. 2011, pp. 1020–1025.

[7] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz, "Understanding sources of inefficiency in general-purpose chips," *Commun. ACM*, vol. 54, no. 10, pp. 85–93, Oct. 2011.

[8] N. Clark, H. Zhong, and S. A. Mahlke, "Automated custom instruction generation for domain-specific processor acceleration," *IEEE Trans. Comput.*, vol. 54, no. 10, pp. 1258–1270, Oct. 2005.

[9] F. Sun, S. Ravi, A. Raghunathan, and N. K. Jha, "Custom-instruction synthesis for extensible-processor platforms," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 23, no. 2, pp. 216–228, Feb. 2004.

[10] L. Pozzi, K. Atasu, and P. Ienne, "Exact and approximate algorithms for the extension of embedded processor instruction sets," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 25, no. 7, pp. 1209–1229, Jul. 2006.

[11] F. Sun, S. Ravi, A. Raghunathan, and N. K. Jha, "A scalable synthesis methodology for application-specific processors," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 14, no. 11, pp. 1175–1188, Nov. 2006.

[12] P. Biswas, S. Banerjee, N. D. Dutt, L. Pozzi, and P. Ienne, "ISEGEN: An iterative improvement-based ISE generation technique for fast customization of processors," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 14, no. 7, pp. 754–762, Jul. 2006.

[13] P. Yu and T. Mitra, "Disjoint pattern enumeration for custom instructions identification," in *Proc. Int. Conf. Field Program. Logic Appl.*, Amsterdam, The Netherlands, Aug. 2007, pp. 273–278.

[14] A. K. Verma, P. Brisk, and P. Ienne, "Fast, nearly optimal ISE identification with I/O serialization through maximal clique enumeration," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 29, no. 3, pp. 341–354, Mar. 2010.

[15] H. P. Huynh, Y. Liang, and T. Mitra, "Efficient custom instructions generation for system-level design," in *Proc. Int. Conf. Field Program. Technol.*, Beijing, China, Dec. 2010, pp. 445–448.

[16] C. Xiao and E. Casseau, "Efficient custom instruction enumeration for extensible processors," in *Proc. 22nd IEEE Int. Conf. Appl. Specific Syst., Arch., Process.*, Santa Monica, CA, Sep. 2011, pp. 211–214.

[17] K. Atasu, W. Luk, O. Mencer, C. C. Özturan, and G. Dündar, "FISH: Fast instruction synthesis for custom processors," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 20, no. 1, pp. 52–65, Jan. 2012.

[18] P. Biswas, N. D. Dutt, L. Pozzi, and P. Ienne, "Introduction of architecturally visible storage in instruction set extensions," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 26, no. 3, pp. 435–446, Mar. 2007.

[19] T. Kluter, "Architectural support for coherent architecturally visible storage in instructions set extensions," Ph.D. dissertation, Sch. Comput. Commun. Sci., École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland, Mar. 2010.

[20] T. Kluter, P. Brisk, P. Ienne, and E. Charbon, "Speculative DMA for architecturally visible storage in instruction set extensions," in *Proc. 6th Int. Conf. Hardw./Softw. Co-Design Syst. Synth.*, Atlanta, GA, Oct. 2008, pp. 243–248.

[21] T. Kluter, S. Burri, P. Brisk, E. Charbon, and P. Ienne, "Virtual ways: Efficient coherence for architecturally visible storage in automatic instruction set extensions," in *Proc. 5th Int. Conf. High-Perform. Embedded Arch. Compilers*, Pisa, Italy, Jan. 2010, pp. 126–140.

[22] T. Kluter, P. Brisk, P. Ienne, and E. Charbon, "Way Stealing: Cache-assisted automatic instruction set extensions," in *Proc. 46th Design Autom. Conf.*, San Francisco, CA, Jul. 2009, pp. 31–36.

[23] *PowerPC 604e RISC Microprocessor Technical Summary*, Motorola, Inc., Schaumburg, IL, 1996.

[24] X. Vera, B. Lisper, and J. Xue, "Data cache locking for tight timing calculations," *ACM Trans. Embedded Comput. Syst.*, vol. 7, no. 1, p. 4, Dec. 2007.

[25] T. Shyamkumar, M. Naveen, J. H. Ahn, and N. P. Jouppi, "CACTI 5.1," HP Labs, Palo Alto, CA, Tech. Rep. HPL-2008-20, Apr. 2008.

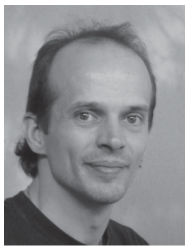
[26] T. R. Halfhill, "EEMBC releases first benchmarks," *Microprocess. Rep.*, May 2000.

[27] W. J. Dally, J. D. Balfour, D. Black-Schaffer, J. Chen, R. C. Harting, V. Parikh, J. Park, and D. Sheffield, "Efficient embedded computing," *Computer*, vol. 41, no. 7, pp. 27–32, Jul. 2008.

[28] C. Favi, T. Kluter, C. Mester, and E. Charbon, "Optically-clocked instruction set extensions for high efficiency embedded processors," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 59, no. 3, pp. 604–615, Mar. 2012.

[29] P. Ranganathan, S. V. Adve, and N. P. Jouppi, "Reconfigurable caches and their application to media processing," in *Proc. 27th Int. Symp. Comput. Arch.*, Vancouver, BC, Canada, Jun. 2000, pp. 214–224.

[30] C. F. Fajardo, Z. Fang, R. Iyer, G. F. Garcia, S. E. Lee, and L. Zhao, "Buffer-integergrated-cache: A cost-effective SRAM architecture for handheld and embedded platforms," in *Proc. 48th Design Autom. Conf.*, San Diego, CA, Jun. 2011, pp. 966–971.
[31] Z. Fang, L. Zhao, R. R. Iyer, C. F. Fajardo, G. F. Garcia, S. E. Lee, B. Li, S. R. King, X. Jiang, and S. Makineni, "Cost-effectively offering private buffers in SoCs and CMPs," in *Proc. 25th Int. Conf. Supercomput.*, Tucson, AZ, May–Jun. 2011, pp. 275–284.
[32] J. Cong, K. Gururaj, H. Huang, C. Liu, G. Reinman, and Y. Zou, "An energy-efficient adaptive hybrid cache," in *Proc. Int. Symp. Low Power Electron. Design*, Austin, TX, Aug. 2011, pp. 67–72.
[33] L. Pozzi and P. Ienne, "Exploiting pipelining to relax register file port constraints of instruction-set extensions," in *Proc. Int. Conf. Compilers, Arch., Synth. Embedded Syst.*, San Francisco, CA, Aug. 2005, pp. 2–10.
[34] J. Cong, Y. Fan, G. Han, A. Jagannathan, G. Reinman, and Z. Zhang, "Instruction set extension with shadow registers for configurable processors," in *Proc. 13th Int. Symp. FPGAs*, Monterey, CA, Feb. 2005, pp. 99–106.
[35] R. Jayaseelan, H. Liu, and T. Mitra, "Exploiting forwarding to improve data bandwidth of instruction set extensions," in *Proc. 43rd Design Autom. Conf.*, San Francisco, CA, Jul. 2008, pp. 43–48.
[36] K. Karuri, A. Chattopadhyay, M. Hohenauer, R. Leupers, G. Ascheid, and H. Meyr, "Increasing data-bandwidth to instruction-set extensions through register clustering," in *Proc. Int. Conf. Comput.-Aided Design*, San Jose, CA, Nov. 2007, pp. 166–171.



Theo Kluter received the Master's degree in electrical engineering from the Technische Universiteit Twente, Enschede, The Netherlands, and the Ph.D. degree from the École Polytechnique Fédérale de Lausanne (EPFL), Lausanne, Switzerland, in 1996 and 2010, respectively.

He is currently a Teacher with the Bern University of Applied Sciences, Burgdorf, Switzerland, and EPFL. He was an R&D Assistant with the Faculty of Computer Controlled Systems and Computer Techniques, the Technische Universiteit Twente from 1996 to 1997. From 1997 to 2002, he was a Design Engineer with the Design Center of Dedris Embedded Systems/Frontier Design/Adelante Technologies, Tiel, The Netherlands. In 2002, he joined Agere Systems as the Acting Interim Product Development Team Leader of the Infotainment Group, Nieuwegein, The Netherlands. In June 2003, he joined EPFL. His current research interests include various aspects of embedded computers and processor architectures, embedded multiprocessor systems-on-chip, design automation, and application-specific embedded system design.

Philip Brisk received the B.S., M.S., and Ph.D. degrees from University of California, Los Angeles, in 2002, 2003, and 2006, respectively all in computer science.

He is currently an Assistant Professor with the Department of Computer Science and Engineering, University of California, Riverside. He was a Post-Doctoral Scholar with the Processor Architecture Laboratory, School of Computer and Communication Sciences, École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland, from 2006 to 2009. His current research interests include application-specific processors, field-programmable gate arrays, and programmable microfluidics.

Dr. Brisk was a recipient of the Best Paper Award at the International Conference on Compilers, Architecture, and Synthesis in 2007 and at the International Conference on Field Programmable Logic and Applications in 2009. He is a member of the program committees of several international conferences and workshops, including Design Automation and Test in Europe, the Asia and South Pacific Design Automation Conference, and the IEEE Symposium on Application-Specific Processors. He was the General Chair or the Co-Chair of the Fourth IEEE Symposium on Industrial Embedded Systems in 2009, the Ninth IEEE Symposium on Application Specific Processors (SASP), in 2010, and the 20th International Workshop on Logic and Synthesis (IWLS), in 2011. He was the Program Chair or the Co-Chair of SASP 2011, IWLS 2012, and the International Symposium on Applied Reconfigurable Computing 2013.

Edoardo Charbon (SM'00) received the Diploma degree from ETH Zurich, Zurich, Switzerland, the M.S. degree from the University of California at San Diego, San Diego, and the Ph.D. degree from the University of California at Berkeley, Berkeley, in 1988, 1991, and 1995, respectively, all in electrical engineering and electrical engineering and computer science.

He is a consultant for numerous organizations, including Bosch, Texas Instruments, Agilent, and the Carlyle Group. He was with Cadence Design Systems, from 1995 to 2000, where he was an Architect of the company's initiative on information hiding for intellectual property protection. In 2000, he joined Canesta Inc., as the Chief Architect, where he lead the development of wireless 3-D CMOS image sensors. From 2002 to 2008, he was a Faculty Member with the École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland, where he was involved in research on CMOS sensors, biophotonics, and ultra low-power wireless embedded systems. In 2008, he joined Delft University of Technology, Delft, The Netherlands, as a Full Professor of VLSI design, succeeding Patrick Dewilde. He has authored or co-authored over 200 papers in journals, conference proceedings, magazines, and two books, and he holds 13 patents. His current research interests include 3-D imaging, advanced bio- and medical imaging, quantum integrated circuits, and space-based detection.

Dr. Charbon was a Guest Editor of the IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS and the IEEE JOURNAL OF SOLID STATE CIRCUITS and a member or the Chair of Technical Committee of ESSCIRC, ICECS, ISLPED, VLSI-SOC, and IEDM. He is a Distinguished Visiting Scholar with the W. M. Keck Institute for Space Studies, California Institute of Technology, Pasadena, CA.

Paolo Ienne (S'94-M'96) received the Dottore degree in electronic engineering from the Politecnico di Milano, Milan, Italy, and the Ph.D. degree from the École Polytechnique Fédérale de Lausanne (EPFL), Lausanne, Switzerland, in 1991 and 1996, respectively.

He joined the Semiconductors Group of Siemens AG, Munich, Germany (which later became Infineon Technologies AG), in 1996, where he was involved in research on datapath generation tools and became the Head of the Embedded Memory Unit, Design Libraries Division. In 2000, he joined EPFL, where he is currently a Professor and the Head of the Processor Architecture Laboratory. His current research interests include various aspects of computer and processor architectures, computer arithmetic, reconfigurable computing, and multiprocessor systems-on-chip.

Dr. Ienne was a recipient of the Best Paper Awards at DAC 2003, at CASES 2007, and at FPL 2009. He is or was a member of the program committees of several international conferences and workshops, including Design Automation and Test in Europe, the International Conference on Computer Aided Design, the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, the International Symposium on Low Power Electronics and Design, the International Symposium on High-Performance Computer Architecture, the International Conference on Field Programmable Logic and Applications, and the IEEE International Symposium on Asynchronous Circuits and Systems. He was the General Co-Chair of the Sixth IEEE Symposium on Application-Specific Processors (SASP'08) and a Guest Editor of a special section of the IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS on Application Specific Processors and a special section on Computer Arithmetic for the IEEE TRANSACTIONS ON COMPUTERS .