Way Stealing: Cache-assisted Automatic Instruction Set Extensions

Theo Kluter[†]
ties.kluter@epfl.ch

Philip Brisk[†]
philip.brisk@epfl.ch

Paolo Ienne[†]
paolo.ienne@epfl.ch

Edoardo Charbon[‡§]
e.charbon@tudelft.nl

Ecole Polytechnique Fédérale de Lausanne (EPFL)
[†]School of Computer and Communication Sciences
[‡]School of Engineering
CH–1015 Lausanne, Switzerland

Delft University of Technology
[§]Circuits and Systems Group
NL–2600 AA Delft, The Netherlands

ABSTRACT

This paper introduces *Way Stealing*, a simple architectural modification to a cache-based processor to increase data bandwidth to and from application-specific *Instruction Set Extensions (ISEs)*. Way Stealing provides more bandwidth to the ISE-logic than the register file alone and does not require expensive coherence protocols, as it does not add memory elements to the processor. When enhanced with Way Stealing, ISE identification flows detect more opportunities for acceleration than prior methods; consequently, Way Stealing can accelerate applications to up to $3.7\times$, whilst reducing the memory sub-system energy consumption by up to 67%, despite data-cache related restrictions.

Categories and Subject Descriptors

B.3.2 [**Memory Structures**]: Design Styles; D.3.4 [**Programming Languages**]: Processors—*Optimization*

General Terms

Design, Performance

Keywords

Application-Specific Processors, Instruction Set Extensions, Way Stealing, Memory Coherence, Automatic Identification

1. INTRODUCTION

The performance requirements of embedded systems must continuously grow within a stringent cost and energy envelope. As a consequence, algorithms to automatically identify application-specific custom *Instruction Set Extensions (ISEs)* have been proposed in recent years [5]. These ISEs effectively extract maximal performance from relatively simple cores.

Unfortunately, the data bandwidth between the main processor and the ISE logic is limited—indeed a classical formulation of the ISE identification problem uses register-port availability as a constraint [9]. To mitigate this problem, *Architecturally Visible Storage (AVS)* uses local memory elements to intrinsically increase the data bandwidth, effectively bypassing the processor itself [1]. AVS systems may use either flip-flops to store scalar variables or local memories to store arrays.

AVS suffers from classic coherence problems in cache based systems; correctness can be ensured by embedding the AVS memories in coherence protocols, such as those available in high-end embedded multiprocessor systems [8]. Using a coherence protocol to facilitate AVS in a single-processor system unfortunately imposes a high cost in terms of energy and silicon real estate: namely the tag array and a more complex cache state machine (see Figure 1).

Way Stealing, the contribution of this work, addresses the aforementioned concerns. Way stealing is a simple architectural modification to a cache-based processor (see Figure 2). Way stealing extends several existing paths in the processor data-cache interface (see Figure 3) to increase bandwidth to and from ISEs, similar in principle to coherent AVS, but without the overhead of a coherence protocol. Some small modifications to ISE identification algorithms to account for Way Stealing are presented as well.

The rest of the paper is organized as follows: Section 2 details the related work in the domain. Section 3 introduces our concept of Way Stealing, the specific problems that one could encounter by its introduction, and brings effective and efficient solutions to all of them. We show in Section 4 how Way Stealing can be integrated in a framework that automatically identifies custom instruction set extensions. In Section 6 we address several benchmarks, and show the effectiveness of ISE with Way Stealing, by using our FPGA-based emulation platform and ISE identification framework as described in Section 5. Section 7 concludes the paper.

2. RELATED WORK

Biswas *et al.* [1] introduced techniques to automatically identify ISEs that can read and write directly to an AVS local memory; this allowed for potentially larger ISEs with greater speedups than prior methods that forbade the inclusion of memory accesses into ISEs. The compiler inserted DMA transfers into the program to move data between main memory and the AVS. Kluter *et al.* [8] 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

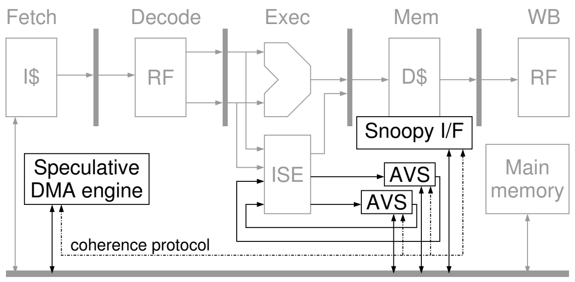

Figure 1: Current state-of-the-art Automatic Instruction Set Extension algorithms provide high bandwidth to the ISE-logic by adding Architecturally Visible Storage, however, they require extensive hardware added to a standard processor pipeline [1, 8].

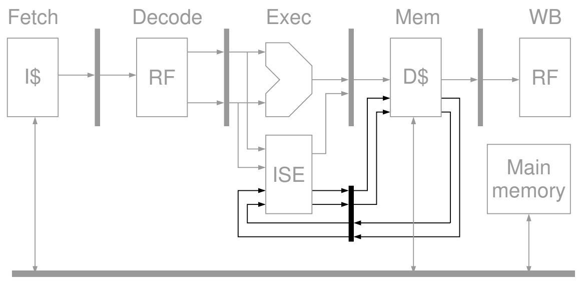

Figure 2: Automatic Instruction Set Extension utilizing Way Stealing, as proposed in this work, provide high bandwidth to the ISE-logic with only simple architectural modification to the data cache and insertion of pipeline registers.

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. *Way Stealing*, achieves coherence but without the overhead of a protocol. Its drawback is that the number of AVS memories is limited by the number of ways in the cache; the coherence-based approach, in contrast, may allocate as many AVS memories as desired.

Other methods, orthogonal to AVS, have been proposed to increase the bandwidth between the processor core and ISEs. Cong *et al.* [2], for example, introduced shadow registers, which are written by the processor's ALU in parallel with the main register file; however, this only increases ISE input bandwidth, with no affect on the output bandwidth. Their ISEs were k-input cones, with at most k inputs and one output; our custom instructions may have any number of inputs and outputs, such as a 3-input, 3-output ISE used for color space conversion in the CJPEG benchmark; their method could not identify this ISE as it is not a k-input cone.

Jayaseelan *et al.* [6] used the forwarding mechanism to resolve data hazards in a pipelined processor to increase the I/O bandwidth to ISEs from 2-inputs and 1-output (the register file) to 4-inputs and 2-outputs; Karuri *et al.* [7] proposed the use of VLIW-style clustered register files, where each bank has 2 inputs and 1 output. These approaches are compatible with Way Stealing, and could be used if the ISE reads and/or writes many scalar variables; Way Stealing should be used when ISE data originates from arrays.

Many reconfigurable cache architectures have been proposed in the past. Ranganathan *et al.* [11] argued that cache associativity should be configurable at runtime to meet varying needs across disparate workloads. One of their proposed configurations places the data/tag SRAM under compiler control, similar in principle to Way Stealing. Their evaluation, however, focused on instruction reuse, while ours focuses on increase performance through ISEs.

3. MORE BANDWIDTH: WAY STEALING

In an n-way set-associative data cache, each way contains separate data and tag memories. All memories are activated in parallel during a lookup. After *hit* detection, the valid data is selected by an n-input multiplexor, and is sent to the processor for storage in the register file (i.e. Figure 3). Prior to the multiplexor, there are n values read in parallel, which, in principle, could be rerouted to the ISE (see Figure 2).

Writing is similar; n values can be written at once. The write behavior of a data cache is normally implemented by a write decoder, which, in turn, is activated by the *hit* signals. By extending the write decoder, providing n data line from the ISE logic to the n data memories, and placing a multiplexor in front of each data memory, the write bandwidth can be extended similarly (see Figure 2).

3.1 Preload and Locking

Way Stealing needs a preload-lock and preload-unlock custom instruction to prevent extensive multiplexing and tag comparison in the data cache. At compile time, these instructions load and lock (unlock) a given array in a pre-determined way of the cache. These instructions require simple extensions to the cache state machine.

During execution of an application, the cache page replacement policy determines which way of the cache will store a given array; in many cases the array will be scattered across different ways. Thus, a compiler cannot feasibly predict the state of the cache prior to executing an ISE. The compiler, therefore, must statically select the ways that hold each array that will be read or written by an ISE at runtime. Additionally, the compiler must ensure that these arrays are not evicted until after the program section containing the ISEs finishes its execution.

The preload-lock and preload-unlock instructions are similar to cache prefetching and software cache-line locking [3]. Our preloading scheme, however, differs from prefetching, as the compiler, rather than the dynamic replacement policy, selects the ways that hold each array. Three distinct situations may occur:

1. Similar to prefetching, the data does not reside in the cache at all, and is loaded using the normal cache load/evict procedure (shown as step 1 in Figure 4). Contrary to prefetching, where the replacement policy selects the way dynamically, the preload instruction statically selects the way, and then locks it, possibly impeding the replacement policy.

2. Similar to prefetching, in case the data already resides in the cache, and in the correct way, nothing has to be done (shown as step 2 in Figure 4).

3. Contrary to prefetching, the data may already reside in the cache, but in a different way than specified by the preload instruction (shown as step 3 in Figure 4). For normal cache operation there may only be one copy of a given data structure in the cache. By loading the data in the specified way, the preload instruction would violate this rule, impeding correct cache behavior; moreover, if the data already in the cache was in the modified state, the preload operation would create an inter-way coherence violation. One approach to solve the problem is to evict the data in the target cache, copy it from the source way to the target way, and then invalidate the source way. A second approach, which we have adopted,

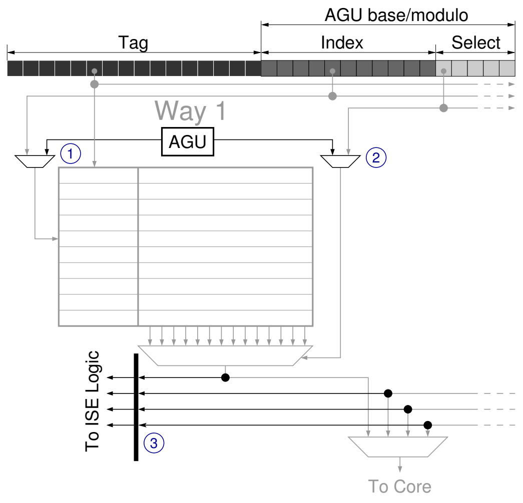

Figure 3: Block diagram of the read path of a standard n-way set-associative data cache. The Way Stealing modifications are shown in black. Multiplexor (1) may be on the critical path. Multiplexor (2) is out of the critical path. The pipeline registers (3) restrict the additional load due to long lines to the ISE logic—they only add an extra pipeline register input load to the read path. The write path is similar and omitted in this figure.

is to swap the contents of the source and destination ways, including the tags. Although the second approach seems more complex, it is faster than the first because it does not initiate a cache line writeback.

3.2 Address Generation Units

In normal cache behavior, all ways from the cache are addressed identically. The address provided by the process is divided into a *Tag*, an *Index* and a *Select*, as shown on the top of Figure 3. The *Tag* determines a hit or miss in a given way through a comparison against the tag memories; the *Index* selects a cache line; and the *Select* picks the correct data in the selected cache line. Using this addressing scheme for Way Stealing would reduce usability in two ways. Firstly, all arrays used in an ISE must be aligned such that the *Index* and *Select* parts of their start addresses are identical; secondly, the access pattern of the data structures must be identical as well. The first restriction can be enforced by padding; however the second restriction limits the algorithmic behavior which an ISE may exhibit.

To alleviate these restrictions, Way Stealing introduces an *Address Generation Unit (AGU)*, much like those found in DSPs, for each way of the cache. Each AGU contains a count register (A_{count}), a stride register (A_{stride}), a mask register (A_{mask}), and a base register (A_{base}). The preload instruction ensures that the array is already present in a given way. This reduces the total addressing of the AGU to the *Index* and *Select* parts, as shown in Figure 3. The preload instruction stores the start address *(Index, Select)* of the data structure into A_{base}. Figure 5 depicts the addressing scheme that the AGU can perform. Although simple, this addressing scheme is easy to automatically detect and is representative of most embedded applications.

3.3 Zero Overhead Loops

Way Stealing requires pipeline registers, shown in Figure 2 and Figure 3, which minimize the increase in the processor's critical path; however, they impose a multi-cycle execution phase of an ISE, as depicted on the left of Figure 6. The program execution, shown on the left hand side, can be accelerated using pipelining, as shown on the right hand side.

Pipelining the path from the data cache to the ISEs is similar to zero overhead loops instructions. The branch and increment phase of a program are removed to amortize their cost on each iteration. Using zero overhead loops in conjunction with Way Stealing amortizes the read delay on each iteration, and only adds one initial read delay at the beginning of each loop. Figure 6 illustrate the benefits of this approach.

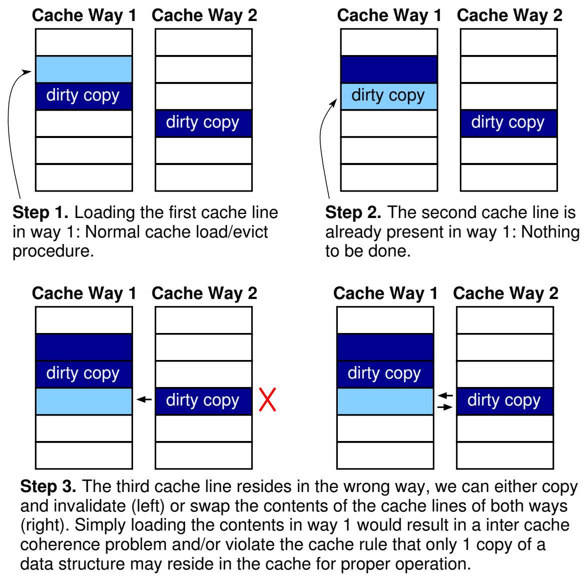
Cache Way 1
Cache Way 2
dirty copy
dirty copy
Step 1. Loading the first cache line in way 1: Normal cache load/evict procedure.
Cache Way 1
Cache Way 2
dirty copy
dirty copy
Step 2. The second cache line is already present in way 1: Nothing to be done.
Cache Way 1
Cache Way 2
dirty copy
dirty copy
X
Cache Way 1
Cache Way 2
dirty copy
dirty copy
Step 3. The third cache line resides in the wrong way, we can either copy and invalidate (left) or swap the contents of the cache lines of both ways (right). Simply loading the contents in way 1 would result in a inter cache coherence problem and/or violate the cache rule that only 1 copy of a data structure may reside in the cache for proper operation.

Figure 4: The different steps of preloading a data structure in way 1 having a potential risk for inter cache coherence problems.

3.4 Coherence

Way Stealing is intrinsically coherent, unlike AVS [1, 8], as it introduces no new memory elements into the processor. Coherence protocols in multiprocessor systems can be extended to accommodate Way Stealing as well, but doing so is beyond the scope of this paper.

3.5 Way Stealing Restrictions

To ensure proper cache operation, only one copy of an array may reside in the cache; therefore, only one data lane, as depicted in Figure 2 can be reserved for each array. Parallel reads and writes are only possible for distinct data structures. Secondly, as the capacity of a way is fixed, special care must be taken for data structures that exceed this size.

4. AUTO ISE WITH WAY STEALING

This section describes a Way Stealing-aware ISE identification flow. ISE selection algorithms, in general, need to model the cost of operations on a normal RISC pipeline in contrast to the cost of the operations executing on an ISE-enhanced pipeline. The concept of Way Stealing introduces extra costs normally not considered in ISE identification flows. The primary challenge is to model the costs of preload and lock/unlock instructions when evaluating an ISE, while accounting for the restrictions listed in the preceding section. Furthermore, Way Stealing requires some specific compiler analyses in the identification process.

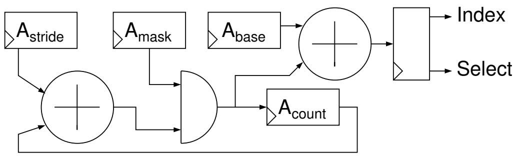

Figure 5: The functional block diagram of the Address Generation Unit (AGU) shown in Figure 3.

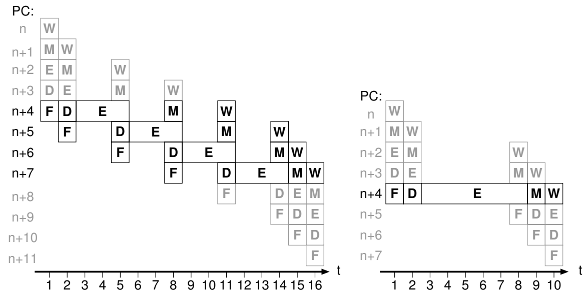

Figure 6: (Left) The execution of a program segment with a unrolled loop of four custom instructions applying Way Stealing. Due to inserted pipeline registers, the read of the data-cache takes 2 cycles, extending the execution phase to 3 cycles. (Right) By introducing the concept of zero overhead loops in the custom instruction, the data-cache read can be pipelined and the same program segment executes faster.

4.1 Modeling the Cost of Way Stealing

Way Stealing has tangible benefits, but also incurs some costs, which must be modeled accurately in order to determine when and where it is best applied.

The first cost occurs when a preload instruction swaps cache lines, as depicted in step 3 of Figure 4, which we assume takes N_{swap} cycles. If the arrays occupy α cache lines, then the maximum preload cost is defined by $\lambda_{\text{preload}} = \alpha \cdot N_{\text{swap}} + 2\alpha$. This cost is only incurred when a swap occurs; the cost of a cache-line load is identical for both normal operation and Way Stealing, and can thus be omitted. At compile time, we cannot determine how many cache lines must be swapped; the estimate, conservatively, is based on worst-case assumptions. The extra factor 2α accounts for cycling through all cache lines during the preload instruction, and unlocking of α cache lines after executing an ISE. The preload cost λ_{preload} replaces the λ_{DMA} in the algorithm described in [1]; however, λ_{preload} is orders of magnitude smaller than λ_{DMA}, as no memory accesses are required.

The second cost, λ_{store}, is the cost of store instructions. RISC based processors execute arithmetic operations on the contents of registers. To move the result to the memory hierarchy, the contents of a register needs to be moved from the register file back to the data cache by using a store instruction. The cost related to this store instruction is $\lambda_{\text{store}} = 1$ processor cycle. However, if the store instruction(s) is/are selected to be part of an ISE with Way Stealing and takes place on one of the data lanes depicted in Figure 2, than λ_{store} reduces to 0. In this case the arithmetic operation is directly performed on the data cache contents, bypassing the register file.

The final cost, λ_{load}, is the cost of load instructions. Similar to the store instruction the contents of the data cache has to be moved to the register file by (a) load instruction(s) with an associated cost of $\lambda_{\text{load}} = 1$ processor cycle. However, when the load instruction(s) is/are selected to be part of an ISE with Way stealing and are allocated to the data lanes depicted in Figure 2, than the λ_{load} should reduce to 0. However, λ_{load} does not become zero, because the pipeline registers inserted in the path between the data-cache and ISE logic increases λ_{load} from 1 cycle to 2.

4.2 ISE Identification Flow

The flow to find ISEs in conjunction with Way Stealing consists of ten phases:

1. **Data structure disambiguation**: The compiler attempts to disambiguate all data structures in the program [1]. All load instructions to arrays that are not disambiguated are marked as forbidden; no ISE may include a forbidden operation.

2. **Loop detection**: The loop hierarchy is computed [10]. All loops with a data-dependent loop count and those that cannot be perfectly nested are marked as forbidden.

3. **Zero overhead loop detection**: The innermost (leaf) loops in the hierarchy identified in the preceding step are examined for compatibility with zero overhead Way Stealing, described in Section 3.3. If the loop has a speedup, this loop is selected as ISE candidate and marked with a *zero overhead* annotation. To prevent the ISE identification step 5 from selecting parts of the loop as an ISE candidate, all operations in the loop are marked as forbidden.

4. **Access pattern detection**: For each loop, the access pattern to each disambiguated data structure is analyzed and matched with the AGU's capability. Load instructions that follow an incompatible access pattern are marked as forbidden.

5. **ISE identification**: ISEs are identified using a known algorithm, e.g., [9]. Loads and stores may be included in the search, and their costs are annotated with λ_{load} or λ_{store} as described in Section 4.1. All ISE candidates including load and/or store instructions are marked with a *Way Stealing* annotation.

6. **Way Stealing Filtering**: All ISE candidates whose data lane or array requirements exceed the number of cache ways, or whose arrays exceed the capacity of a way, are removed from consideration. Transformations such as loop unrolling must be made aware of these restrictions to be used successfully in conjunction with Way Stealing. The loops selected in phase 3 are added as potential ISE candidates.

7. **Preload Annotation**: λ_{preload} is added to the cost of each ISE candidate with a *Way Stealing* annotation. The cost for configuring the AGUs is calculated and added to each candidate as well. After the update, all ISEs no longer having any speedup are removed.

8. **Custom instruction selection**: The best m ISE candidates are selected; m is a designer-specific parameter.

9. **Preload instruction insertion**: Preload and unlock instructions are inserted, always outside the body of leaf loops. In all other situations, they are directly placed before and after the ISEs. More sophisticated placement algorithms are beyond the scope of this paper. AGU setup instructions are inserted similarly.

10. **Custom instruction insertion**: ISEs are inserted at the appropriate locations.

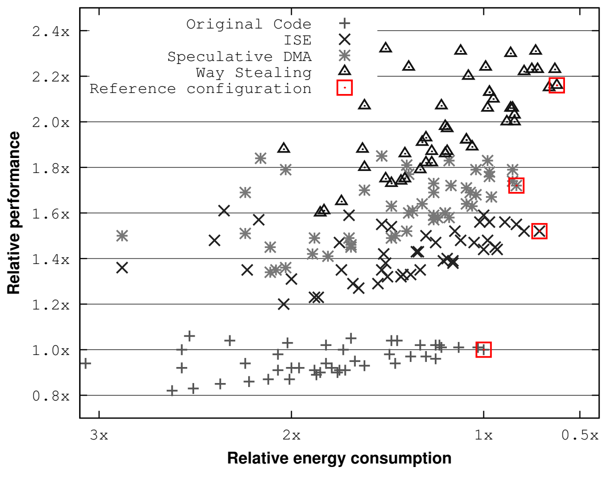

Figure 7: Design space exploration of the CJPEGV2-DATA1 testbench for the different processor specializations. The cache configuration delivering the best performance per energy for the original system is chosen as reference point.

4.3 Example of Generated Code

To demonstrate the overhead introduced by Way Stealing, following code snippet is used:

```
for (i=0; i<10; i++)
   b[i] = ise1(a[i],c[i]);
```

After the complete identification phase, the code-snippet is transformed into the following Way Stealing assembly code:

```
preload b,w1,10*4    // load b[] into way 1
preload a,w2,10*4    // load a[] into way 2
preload c,w3,10*4    // load c[] into way 3
setall  Amask,-1     // all Amask regs = 0xFF
setall  Acount,0     // all Acount regs = 0
setall  Astride,1    // all Astride regs = 1
ise1    w1,w2,w3,10  // perform ISE
unlock  b,w1,10*4    // unlock way 1
unlock  a,w2,10*4    // unlock way 2
unlock  c,w3,10*4    // unlock way 3
```

5. EXPERIMENTAL SETUP

We used modified an internally developed research compiler to perform ISE identification using Way Stealing. We also integrated prior relevant algorithms [1, 8] into the same environment. Our target was an OpenRISC-compatible FPGA-based emulation platform with all of the architectural changes required to support Way Stealing, as described in Section 3.

We selected several applications from the EEMBC benchmark suite [4] to evaluate Way Stealing. For each benchmark, the compiler generated VHDL models of the ISE, with or without Way Stealing, which were then added to the FPGA-based emulation platform. The compiler also generates modified C-code that includes calls to the appropriate ISEs, including prefetching and lock (unlock) instructions. The modified C-code was cross-compiled using a gcc 3.4.4 toolchain based on "newlib" for the OpenRISC.

The FPGA-based emulation platform has software controllable 16 kB instruction and data caches. Our experiments used an 2, 4, 8, and 16 kB 1-way, 2-way, and 4-way set associative instruction cache with a *Least Recently Used (LRU)* replacement policy. We used a 2, 4, 8, and 16 kB 4-way set associative data cache with an LRU replacement policy that was enhanced to support Way Stealing.

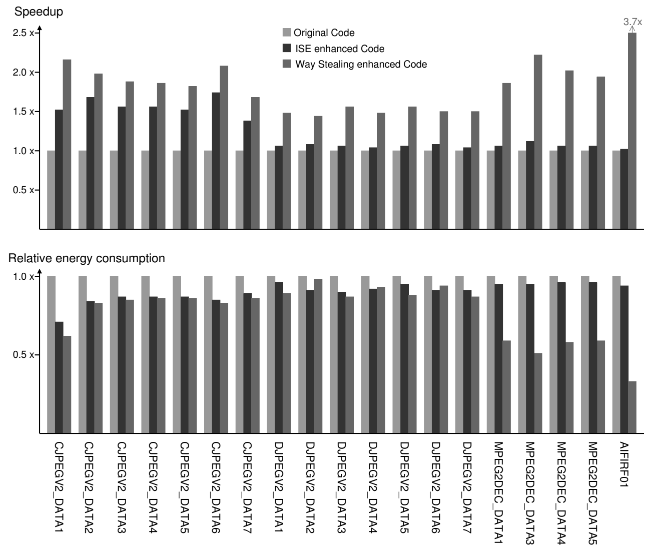

Figure 8: (Top) The speed up of several EEMBC benchmarks and data sets applying traditional ISEs and Way Stealing relative to the run-time of the original source code. (Bottom) The corresponding relative energy consumption graph of the same benchmarks as shown on the top.

For all benchmarks, we imposed a 4:2 input-output constraint on the ISE identification algorithm when Way Stealing was not used, which is consistent with the register file restrictions; when Way Stealing was used, we increased the input-output constraint to 6:5 to account for the greater bandwidth.

We used CACTI [12] to determine the read/write energy consumption for different cache configurations in a 90 nm technology. The energy consumed by the Way Stealing read/write accesses is conservatively overestimated by assuming that each read/write access consumes the same amount of energy as a normal cache read access. The external memory and bus-access read/write energy consumption is estimated to be 792pJ per access. The energy values reported here only include the dynamic energy consumed in the memory sub-system; this model does not include processor and leakage energy.

For all experiments we performed a cache design space exploration on the original system (as shown in Figure 7). We determined the cache configuration with the best performance-energy product as the reference configuration for our experiments.

6. EXPERIMENTAL RESULTS

Figure 8 shows the relative execution time and energy consumption of several EEMBC benchmarks augmented with Way Stealing custom instructions with respect to the original code. Figure 9 shows a detailed graph of the CJPEG benchmark, shown as the first set of bars in Figure 8, and includes comparison with traditional ISEs [9] and speculative DMA [8]. Both Figure 8 and Figure 9 only represent the results relative to the reference configuration.

Figure 8 shows clearly that Way Stealing consumes significantly less energy and achieves tangible speedups compared to the baseline. The reduction of energy consumption due to four effects, the first of which is common to all ISE methods, and the latter three of which are specific to Way Stealing: (1) replacing several instruc-

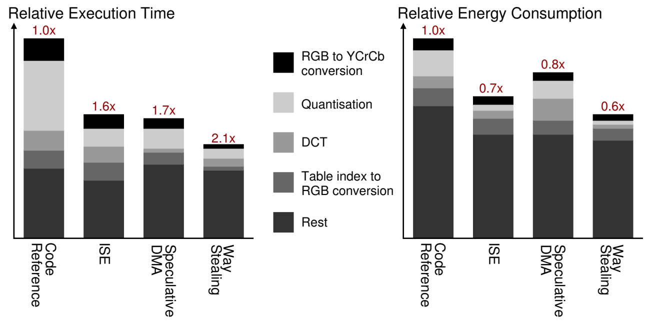

Figure 9: Detailed graph of the CJPEG benchmark shown in Figure 8. (Left) Way Stealing provides more opportunities for ISE detection due to its reduced overhead cost; however, on some kernels (like the DCT) the state-of-the-art outperforms Way Stealing. (Right) Way Stealing does not require expensive data structure moves to and from an AVS, consuming significantly less energy.

tions by one ISEs reduces the number of instruction cache fetches; (2) allocating one data structure to each way reduces data cache thrashing and capacity misses for data structures that exceed the capacity of a single way; (3) parallelizing cache read/write accesses reduces the total number of cache access; and (4) the use of zero overhead loop instructions removes a significant number of instruction cache accesses.

Figure 9 compares Way Stealing to both traditional ISEs and ISEs augmented with AVS [1] and speculative DMA to ensure coherence [8]. The *DCT* kernel benefits significantly from storing an 8×8-byte matrix in an AVS with eight read and eight write ports to the pipelined DCT logic (an ISE). Unfortunately, since each data structure may reside in just one way of the cache, Way Stealing can only provide a single read and write port, regardless of associativity; this sequentializes the loads and stores, inhibiting the acceleration of this particular kernel compared to coherent AVS.

Second, among all kernels shown in Figure 9, Way Stealing finds more opportunities for acceleration compared to existing methods, with or without coherent AVS. This results from the reduced overhead of λ_{preload} compared to the λ_{DMA} required for DMA transfers when AVS is used. The use of zero overhead loops in conjunction with Way Stealing provides additional advantages that traditional ISEs, whose execution bodies are restricted to convex data flow subgraphs, cannot.

Lastly, Figure 9 shows that *Speculative DMA* consumes significantly more energy in the quantization kernel than the other three scenarios. *Speculative DMA* increases the miss rate in the data cache due to the DMA transfers required to facilitate coherent AVS; this is typical of other benchmarks as well. Way Stealing avoids these extra misses as it access the data directly from the cache.

7. CONCLUSIONS

Way Stealing is a simple architectural modification to a cache-based embedded processor that significantly increases data bandwidth to and from ISEs. Our results, derived by a cycle-accurate FPGA-based emulation platform, shows that ISEs enhanced with Way Stealing improve performance and reduces power consumption compared to the state-of-the-art. Due to restrictions imposed by the cache, Way Stealing does not achieve the best performance on all kernels compared to AVS, but ensures coherence at a much cheaper cost, and does not require the instantiation of new AVS memories. For these reasons, we believe that Way Stealing should be used instead of coherent AVS for modern embedded systems where energy efficiency and reduced cost trump raw performance.

8. ACKNOWLEDGMENTS

The authors thank Bhargava Shastry for his extensive work in generating the experimental results, and Xilinx for their generous donation of FPGA devices through their *XUP* program.

9. REFERENCES

[1] P. Biswas, N. Dutt, L. Pozzi, and P. Ienne. Introduction of architecturally visible storage in instruction set extensions. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, CAD-26(3):435–46, Mar. 2007.

[2] J. Cong, G. Han, and Z. Zhang. Architecture and compiler optimizations for data bandwidth improvement in configurable embedded processors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 14(9):986–97, Sept. 2006.

[3] J. A. Fisher, P. Faraboschi, and C. Young. *Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools*. Morgan Kaufmann, San Francisco, Calif., 2005.

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

[5] P. Ienne and R. Leupers, editors. *Customizable Embedded Processors—Design Technologies and Applications*. Systems on Silicon Series. Morgan Kaufmann, San Mateo, Calif., 2006.

[6] R. Jayaseelan, H. Liu, and T. Mitra. Exploiting forwarding to improve data bandwidth of instruction-set extensions. In *Proceedings of the 43rd Design Automation Conference*, pages 43–48, San Francisco, Calif., July 2006.

[7] K. Karuri, A. Chattopadhyay, M. Hohenauer, R. Leupers, G. Ascheid, and H. Meyr. Increasing data-bandwidth to instruction-set extensions through register clustering. In *Proceedings of the International Conference on Computer Aided Design*, pages 166–71, San Jose, Calif., Nov. 2007.

[8] T. Kluter, P. Brisk, P. Ienne, and E. Charbon. Speculative DMA for Architecturally Visible Storage in Instruction Set Extensions. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis*, pages 243–48, Atlanta, Ga., Oct. 2008.

[9] L. Pozzi, K. Atasu, and P. Ienne. Exact and approximate algorithms for the extension of embedded processor instruction sets. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, CAD-25(7):1209–29, July 2006.

[10] G. Ramalingam. On loops, dominators, and dominance frontiers. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(5):455–90, Sept. 2002.

[11] P. Ranganathan, S. V. Adve, and N. P. Jouppi. Reconfigurable caches and their application to media processing. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 214–24, Vancouver, June 2000.

[12] D. Tarjan, S. Thoziyoor, and N. P. Jouppi. CACTI 4.0. Technical Report HPL-2006-86, Hewlett-Packard Development Company, Palo Alto, Calif., June 2006.