Speculative DMA for Architecturally Visible Storage in Instruction Set Extensions

Theo Kluter, Philip Brisk, Paolo Ienne, and Edoardo Charbon
Ecole Polytechnique Fédérale de Lausanne (EPFL)
School of Computer and Communication Sciences
CH-1015 Lausanne, Switzerland
E-mail: {ties.kluter, philip.brisk, paolo.ienne, edoardo.charbon}@epfl.ch

ABSTRACT

Instruction set extensions (ISEs) can accelerate embedded processor performance. Many algorithms for ISE generation have shown good potential; some of them have recently been expanded to include *Architecturally Visible Storage (AVS)*—compiler-controlled memories, similar to scratchpads, that are accessible only to ISEs. To achieve a speedup using AVS, *Direct Memory Access (DMA)* transfers are required to move data from the main memory to the AVS; unfortunately, this creates coherence problems between the AVS and the cache, which previous methods for ISEs with AVS failed to address; additionally, these methods need to leave many conservative DMA transfers in place, whose execution significantly limits the achievable speedup. This paper presents a memory coherence scheme for ISEs with AVS, which can ensure execution correctness and memory consistency with minimal area overhead. We also present a method that speculatively removes redundant DMA transfers. Cycle-accurate experimental results were obtained using an FPGA-emulation platform. These results show that the application-specific instruction-set extended processors with speculative DMA-enhanced AVS gain significantly over previous techniques, despite the overhead of the coherence mechanism.

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, Architecturally Visible Storage, Instruction Set Extensions, Speculative Direct Memory Access

1. INTRODUCTION

The performance requirements of embedded systems are continuously growing under a very constrained cost and energy envelope. System designers must extract maximal performance from relatively simple cores through specific architectural decisions; processor customization is a possible framework for such architectural decisions [8] and instruction set extensions one of the main tools in the hands of designers.

Unfortunately, the power of customized instructions is often limited by the data bandwidth to and from the datapaths implementing them—indeed a typical formulation of the instruction-set extension identification problem has register-port availability as a critical constraint [10]. One way to moderate the problem is to add *Architecturally Visible Storage (AVS)*, which intrinsically provides the customized datapath with additional local bandwidth. Architecturally visible storage may simply mean scalar registers to hold local variables mostly used by the customized instruction. It can also mean complete data structures, such as local arrays, whose content is used over and over by the special instruction.

Some researchers have made first steps in introducing memory elements in automatically discovered instruction set extensions [4]. The basic idea is to enable selectively the inclusion of load and store operations (often simply excluded) in the custom instruction, assuming that such load and store operations are in fact implemented not from main memory but from a locally instantiated register or small ASIC memory; such memories are not fundamentally dissimilar from ad-hoc *Scratch Pad Memories (SPMs)*. The key of the methodology is to provide any classic instruction set extension algorithm (such as [10] in that case) with a good estimation of the cost of bringing in and out the required data, usually with *Direct Memory Access (DMA)* transfers. The authors developed an algorithm to compute such a $\lambda_{overhead}$ parameter by placing DMA transfers in those positions in the program that guarantee correct execution (coherent content between the custom memory and main memory) and have minimal cost (that is, are executed as seldom as possible). Our goal is to improve on the results of Biswas *et al*. [4] in two directions: (1) extend the applicability to more complex and typical systems where their results would be optimistic or even incorrect, and (2) improve the efficiency of their solution.

The rest of the paper is organized as follows: Section 2 details the related work in the domain. Section 3 discusses the specific problems that one could encounter in the introduction of architecturally visible storage inside custom instructions, and brings effective and efficient solutions to all of them. We prove this in Section 5 by addressing a complete application displaying all qualitative situations of interest, and by using the experimental environment described in Section 4. Section 6 concludes the paper.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CODES+ISSS'08, October 19–24, 2008, Atlanta, Georgia, USA.
Copyright 2008 ACM 978-1-60558-470-6/08/10 ...$5.00.

2. RELATED WORK

Most of the research in automatic identification of instruction set

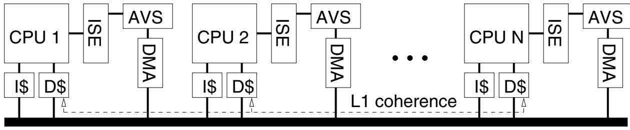

Figure 1: The basic system architecture that we target. We assume that a cache coherence protocol to support multiple processors is already implemented.

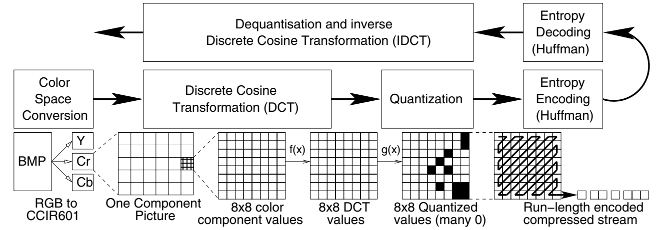

Figure 2: JPEG compression and decompression algorithm. Top: the six main kernels of the algorithm. Bottom: schematic representation of the compression algorithm. The decompression algorithm has the same steps, but in reverse order.

extensions has been carried out independently from the problem of storage [5, 10, 13, 2, 12]. In many cases load and store instructions would simply be omitted from custom instructions while in some cases they can be included allowing normal memory ports from the custom units. The problem of adding architectural visible storage to custom functional units has been addressed directly by Biswas *et al*. [4], in the various forms of constant tables, scalar variables, and arrays. In this paper we solve a few shortcomings of the system presented in [4] and exploit the architectural features of modern multiprocessor systems-on-chip to achieve significantly better results.

Although this work is positioned toward prior art in instruction set extensions much in the same way as Biswas *et al*. [4], it additionally relates to the body of literature on cache coherence in multiprocessors [6]. If both [4] and our work relate to the problem of synthesizing application-specific memories [3] or allocating data to scratch pads [11], doing so for custom functional units in systems equipped with data caches naturally raises problems of coherence between the various copies of the data (typically, main memory, cache, and storage in the custom instruction). To address the issue, we assume that our target platform is already equipped with a typical coherence protocol, such as MESI [9]. Although by no means inexpensive, such protocols are becoming frequent in high-end embedded systems because of the growing importance of multiprocessor systems-on-chip (e.g., ARM Cortex-A9 MPCore [1]). We seamlessly adapt our customized memories to standard coherence protocols, maintaining coherence protocol compatibility with special care to minimize the additional hardware cost. Figure 1 shows the type of system that we address in this paper.

3. THE IMPORTANCE OF COHERENCE

As mentioned, previous methods to include architecturally visible storage in instruction set extensions have shown very tangible potentials but also display a number of nonnegligible shortcomings. To illustrate the limitations, we will use a complete application composed of several heterogeneous functions and comprising examples of all problems we address. We will take them one by one in successive sections and show our solutions to overcome them. We focus on three contributions:

- **Ensuring coherence.** We will show that coherence is, in all but the most trivial cases, an issue that cannot be overlooked and must be considered while adding memory elements to instruction set extensions. We address the challenges of solving this issue efficiently without losing altogether the benefits of instruction set extensions with memory—a very concrete risk, as we will show.
- **Speculative DMA.** The naive introduction of DMA transfers in all positions that are indicated by standard program analysis [4] may result in suboptimal performance due to an overly conservative number of transfers to and from the instruction set extension memories. We show that one can leverage the investment in the coherence network to improve significantly on the results with very limited hardware cost.
- **Improved DMA overhead modeling.** One can obtain significantly suboptimal results by the use of a simple and local model of the data transfer overhead to get data into and out of the instruction set extension memory. We indicate how this model can be improved with no significant increase in algorithmic complexity.

Before addressing these three points, we will briefly discuss the sample application which we will use throughout the paper.

3.1 JPEG Encoding and Decoding

We target the kernels of the JPEG compression and decompression algorithm as very simple but still representative example of an application for the system of Figure 1. JPEG compression is taken from the EEMBC [7] test-bench suite and its kernels are depicted in Figure 2. From the six kernels presented in Figure 2, we use the De-quantization and Inverse Discrete Cosine Transformation as a motivational example throughout the remainder of the paper. This kernel will be referred to as IDCT.

It is worth mentioning that our interest is in fully automated practical solutions. We have taken, for our example, a relatively optimized version of the JPEG encoding and decoding application: it contains a few optimizations for improving the software performance, as does most production code. For instance, in the IDCT kernel the software designer, aware of the fact that most of the AC-components of the entropy-decoded Discrete Cosine Transformed (DCT) values are zero, has implemented in the IDCT function a fast-track in case of all zero AC-values, only performing a complete one-dimensional IDCT otherwise. The pseudocode of the software-optimized IDCT is given on the left side of Figure 3. We will see that these software optimizations can actually prevent or worsen the impact of hardware optimization through instruction set extensions including architecturally visible storage. To ensure realistic results, we will not undo such optimizations but we will show how to work with them, albeit somehow sub-optimally.

3.2 Ensuring Coherence

By applying the algorithms from [4] on the IDCT, the *Control Flow Graph (CFG)* shown to the right of Figure 3 is found. *Basic Block 1* (*BB1*) and *BB5* represent the **if** constructs of the pseudocode, *BB3* and *BB7* the fast paths discussed in the previous section, *BB2* and *BB6* the 1-dimensional IDCTs, and *BB4* and *BB8* the loops. The algorithms of [4] detects that *BB2* and *BB6* can be sped up by using ISEs with architecturally visible storage. The memories included into the ISEs are, amongst others, the buffer holding the 64 IDCT values (read-write). In this work we consider the data

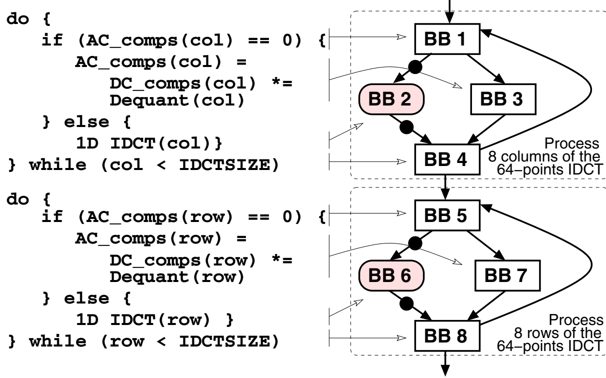

Figure 3: Left: The Pseudo code of the software-optimized IDCT kernel. Right: The Control Flow Graph of the IDCT kernel. Applying the algorithm of [4] will detect BB2 and BB6 having ISEs with AVS. The filled circles denote the DMA-in/DMA-out actions.

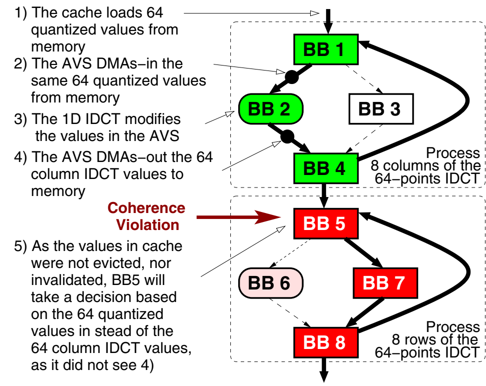

Figure 4: Example of a coherence violating path through the IDCT kernel. A coherent path would be *BB1-BB3-BB4- BB5-BB7-BB8*.

structures included in the AVS-search to be processor local. This assumption implies that the data structures reside in the cache in either *exclusive* or *modified* state. Therefore, these data structures will never be requested by other processors, allowing us to perform the experiments on a single processor system without losing generality. Although this assumption may be arguably too restrictive, dealing with shared data structures will be left for future work.

Executing the IDCT as presented in Figure 3 will result in incorrect program behavior in the presence of one or more data caches. To demonstrate incorrect execution, we look at the situation depicted in Figure 4. The processor starts by performing a column based one-dimensional IDCT. To perform the IDCT, the processor takes the path through *BB2*, which contains an ISE with architecturally visible storage. The processor performs a DMA-in transfer and the values present in memory are copied into the architecturally visible storage. Here already the first problem arises: the memory may not contain the latest copy of the data as the latter can be dirty in the cache; the DMA-in transfer not being coherence aware, this results in incorrect execution. If by chance the memory contains the correct values, the execution of the column-based IDCT continues correctly. The column-based values are updated into memory through the DMA-out action following *BB2*. After one iteration, *BB1* will check the next column, that is stored in the cache, and continues operation. As all columns are mutually exclusive, no problem arises for *BB1* in obtaining the latest copy of the data. However, when reaching *BB5*, the execution of the program is guaranteed to be incorrect: (i) *BB1* has forced the cache to load local copies of the data structure into the cache. (ii) The DMA-in and DMA-out actions surrounding *BB2* have not informed the cache on changes of the data; thus the cache will keep its old copy. (iii) When reaching *BB5*, the processor will use incorrect values.

Radical ways to avoid the coherence problem include flushing the data cache prior to DMA transfers or removing the data cache altogether. We will show in the results that, although these methods do resolve the coherence problem, their influence on performance is simply inacceptable and a different strategy is needed. Given the fact that our architecture already provides a coherence protocol between the caches, the simplest solution is to include the architecturally visible storage into such coherence protocol. However, the following problems may arise:

- **Stall requirement.** As the coherency protocol can invalidate the architecturally visible storage at any time, or request at any time a write-back, a stall mechanism must be integrated in the ISE, and indirectly into the processor.

- **Granularity.** To work properly, the coherence protocol expects a burst transfer of the size of a cache line. As, in general, the architecturally visible storage is not of the size of a single cache line, we have to take precautions to make a granularity adaptation.

- **Alignment.** As we do not know if the data structure contained in the architecturally visible storage is cache-block aligned, we have to take precautions on misaligned data.

Although the stall requirement problem appears to be the most difficult and costly to solve, it is actually the easiest one: We are guaranteed by the algorithm in [4] that, between any pair of DMA-in and DMA-out transfers, all references to the data structure contained in the architecturally visible storage will be done through the custom instructions, and not through the cache. Therefore, making sure that a DMA-in transfer will request the write-back of all polluted data contained in the caches, we are guaranteed that no coherence invalidation or write-back request will occur during the execution of the custom instruction. The coherence protocol enforces this behavior, resulting in no stall requirement for the custom instructions. The granularity and alignment problems can easily be solved by splitting up the architecturally visible storage into chunks of the size of a cache line, or smaller. Each of these chunks are then burst individually in or out of the architecturally visible storage. Although these chunk-based DMA transfers require more cycles than normal DMA transfers, they provide correctness of execution through the existing and unmodified coherence protocol.

3.3 Speculative DMA Transfers

Revisiting Figure 4, one can notice that, during the processing of the eight columns of the IDCT, the processor may, in the worst case, request eight DMA-in and eight DMA-out transfers. As discussed before, the correctness of the IDCT can be guaranteed without the cache having the latest version of the data structure used in *BB2*. Hence, the number of required DMA transfers is in fact not sixteen but two. However, as it cannot be guaranteed that the processor will always go through *BB2*, the DMA-in and DMA-out transfers cannot just by moved before *BB1* and after *BB4*, respectively. This is an example, as anticipated in Section 3.1, of a difficulty that was

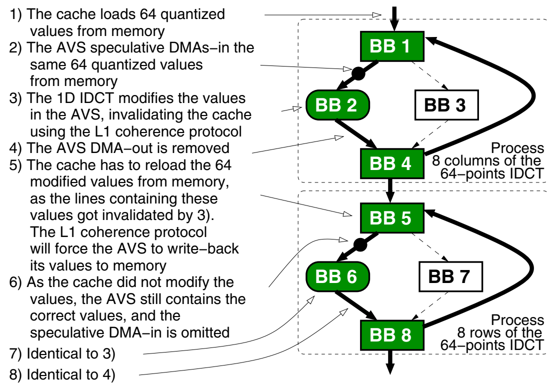

Figure 5: The CFG of the IDCT after applying speculative DMA transfers. Correctness of execution is guaranteed, although all DMA-out transfers are removed and all DMA-in transfers are only performed when actually required.

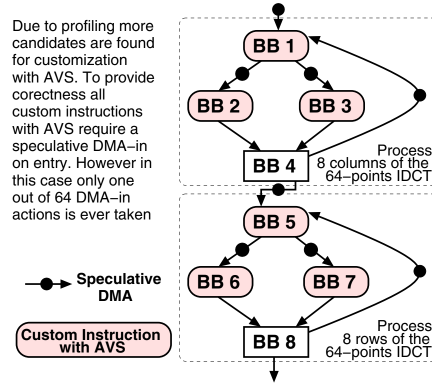

Figure 7: The CFG of the IDCT kernel after applying profile-based speculative DMA transfers.

artificially created by the software optimizations of the code. However, by keeping track of the state of the data structure contained in the architectural visible storage, we can speculatively remove redundant DMA-in transfers. Keeping track of the state of the data structure can be realized by adding a single valid bit to the architecturally visible storage of a particular data object. A real DMA-in transfer will set this valid bit. We can simply use the snooper of the available coherence protocol to invalidate the content of our architecturally visible storage. A DMA-in transfer is now only performed in case the valid bit is inactive—we call this *Speculative DMA Transfer*. Its cost is minimal: one added bit in the hardware and one cycle delay for each redundant DMA-in transfer. In the case of Figure 4 we can save in the best case seven times the DMA-in transfer delay minus seven cycles. In a critical loop in the program this can result in drastic speedups.

Performing a similar speculation on the DMA-out transfer is not as simple as the speculation on the DMA-in. To ensure coherence, we must send out a coherence invalidation message for each chunk of the architecturally visible storage of the size of a cache line. To perform speculative DMA-out we have to add a dirty bit to the architecturally visible storage which is activated whenever the custom instruction writes in the local memory. If the dirty bit was reset prior to this writing by the custom functional unit, we have to send out invalidation messages for all chunks of the architecturally visible storage. At the moment of the speculative DMA-out transfer, we have to DMA-out the data contained inside the architectural visible storage only when the dirty bit is active. The DMA-out transfer resets the dirty bit. The problem with this scheme is that it is extremely likely that the ISE with architectural storage writes

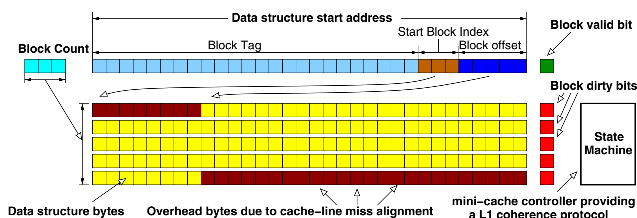

Figure 6: Speculative DMA memory architecture. The architecturally visible storage is transformed from a SPM-like structure into a mini-cache-like structure.

to it: no DMA-out transfer is prevented as the dirty bit is always active when hitting the DMA-out transfer, and we introduced extra hardware overhead with no real use. An interesting alternative is to remove the DMA-out transfers from the program altogether and introduce the dirty bit into the standard coherence protocol. Due to the resulting coherence invalidation messages, all caches are guaranteed to have no local copy of the data structure contained in the architectural visible storage (we are the exclusive owner). In case the processor wants to access the data structure through the cache, a cache-miss will occur and the cache will request the data from main memory. Hence, by introducing the dirty bit into the coherence protocol, we can detect this request and use the coherence protocol to safely write-back our dirty copy to main memory before the cache gets its copy. This ensures correctness for speculative lazy DMA-out transfers. Figure 5 shows our example CFG after introducing speculative DMA transfers.

To support speculative DMA transfers, Figure 6 lists the additions to the basic architecturally visible storage. In Figure 6, *Block Count* is required for the number of cache-line sized and aligned chunks contained in the architectural visible storage (not necessarily a power of 2). The data structure start address is the source/target address required by a DMA controller to know where to copy from/to the data. However, in the speculative DMA the data structure start address is also used as the tag for the coherence protocol. Figure 6 furthermore shows the valid bit and one dirty bit for each cache-line sized chunk of the architectural visible storage. We added one dirty bit for each chunk, but this is not necessarily required: only one dirty bit is sufficient. However, depending on the access patterns, multiple dirty bits might improve the performance. Figure 6 suggests that essentially we have transformed the architecturally visible storage from a SPM-like structure into a peculiar and relatively cheap coherent cache: it contains a single oversized cache line with a single tag, segmented dirty bits, and a segment count. The state machine is similar to the state machine of a cache with coherence protocol, and the coherence states are *Shared(S)* (dirty bits are inactive and valid bit is active), *Invalid(I)* (all bits are inactive), and *Modified(M)* (valid bit is active and at least one dirty bit is active).

3.4 Improved DMA Overhead Modeling

Each transformation of the DMA behavior as described in the previous sections has a drastic impact on the $\lambda_{overhead}$ mentioned in [4] and representing the additional cost induced by the DMA transfers. Such value is an essential input to the instruction set extension search algorithm, as it is a nonnegligible part of the speedup

Table 1: Speedup results for two selected kernels of the JPEG algorithm, applying the different optimizations.

Kernel	DCT	IDCT
Atasu	1.1	0.9
Biswas coherence violating	9.0	1.3
Biswas Data Cache Flush	1.3	0.2
Biswas no Data Cache	6.2	0.7
Biswas corrected *M(C)*	1.1	0.9
Coherent DMA	6.3	1.0
Speculative DMA	9.6	9.0

metric that such algorithm tries to optimize. However, due to the dynamic nature of the speculative DMA transfers, the value calculated statically for $\lambda_{overhead}$ as in [4] may be extremely imprecise. A better estimation can only be obtained by profiling all data accesses in a given CFG for all data structures contained in this CFG. Although these data traces are dependent on the used data set, they make it possible to compute a $\lambda_{overhead}$ for a given data structure in a given CFG, where all *BB*s, using this data structure, are converted into ISEs with architectural visible storage. This statistical $\lambda_{overhead}$ can be used in the same way as the static value of [4] and a ISE search can be performed. The resulting CFG for the IDCT is shown in Figure 7. Although our approach is still simplistic here, we intend to perform more research and improve the statistical modeling the influence of the speculative DMA transfers on $\lambda_{overhead}$.

4. EXPERIMENTAL SETUP

We implemented our speculative DMA transfers by implementing the algorithms of [4] in our own environment for automatic generation of instruction set extensions. We have augmented the algorithms of [4] with data profiling support to achieve the goals discussed in Section 3.4. Finally, we modified the merit function optimized by the instruction set extension selection algorithm [10] to account for the D$-flushing overhead and speculative DMA transfers. We used the complete JPEG encoding/decoding chain (as described in Section 3.1) as our benchmark. Our environment for automatic generation of instruction set extensions has generated five architectural versions with corresponding modified C-code and VHDL-models of the ISEs (with or without local memories). We merged all the generated VHDL models of the ISEs to generate one module to augmented our OpenRISC-compatible platform running on FPGA. The different versions of DMA controller have been implemented by hand in VHDL, giving software control over normal DMA, coherent DMA and speculative DMA. All the modified C-code has been cross-compiled using a gcc 3.4.4 toolchain based on "newlib" for the OpenRISC.

Our FPGA-based multi-processor platform allows us to emulate cycle-accurately embedded systems consisting of 1 to 7 OpenRISC processors. The platform has software controllable 16 kB instruction and data caches. Consistently with our architectural assumption of Figure 1, the data caches implement a MESI Level 1 coherence protocol. For our experiments we used the single processor version with a 16 kB 2-way set-associative instruction cache with LRU-replacement policy. We used a 16 kB 4-way set-associative data cache with LRU-replacement policy and MESI Level 1 coherence protocol. For the experiment without the data cache, we disabled it in software.

For all the experiments we used the same 24-bit RGB encoded picture of 1024x768 pixels, similar to the resolution of current high-end web-cams and standard portable phones. Furthermore we

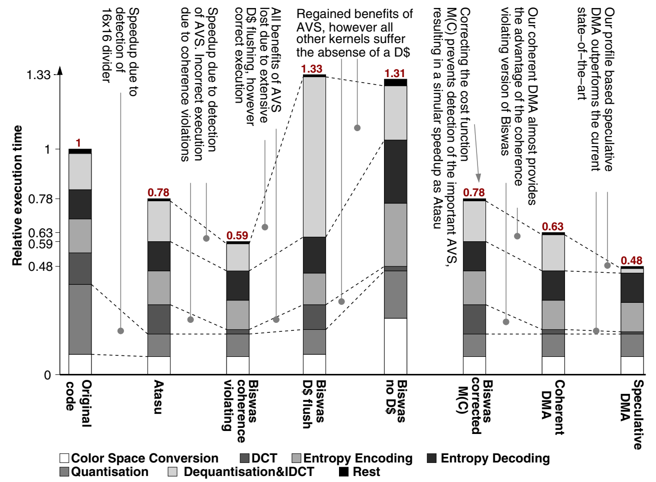

Figure 8: Relative run-time for the JPEG kernels of the different approaches to speedup the program applying ISEs with and without AVS.

gave a 4:2 input-output constraint to the algorithms for automated instruction set extension generation [10].

5. EXPERIMENTAL RESULTS

Firstly, we ran the original code without ISEs on our emulation platform to confirm the run-time models used in our own environment for automatic generation of ISE. The run-time breakdown for the different kernels are shown in the first column of Figure 8 and are normalized to the total runtime (about 290 million cycles) of the original code. In all the experiments there are only compulsory instruction-cache misses (about 300 misses in 210 million accesses). The data-cache miss rate varies over the different configurations due to the coherence overhead and is listed together with the *Instructions Per Cycle (IPC)*, speedup, and relative execution time in Table 2.

Subsequently, we used the code with ISEs without AVS (using the Atasu algorithm [10]). The kernel breakdown is seen in Figure 8. Overall, the Atasu algorithm gains over the original code. The Atasu algorithm achieves the most gain by detecting a 16x16-bit divider in the quantization kernel (providing a speed-up of 3.18 for this kernel). Looking a bit closer at the different kernels, we can detect a speed-down for the IDCT kernel (as shown in Table 1). The speed-down in this kernel is due to the compiler not being able to optimize the code as in the previous experiment. In our profile, we find 15% more executed instructions compared to the original code. This increase offsets the predicted 6% speed-up in this kernel causing slow-down.

Next, we investigated the code generated by the Biswas algorithm. Although the speed-up given by the detection of AVS is impressive (as can be seen in Figure 8 and Table 1), the program did not execute correctly, because of coherence violations. We confirmed the correctness of the program by disabling the data cache. Furthermore we issued a data-cache flush before any DMA-in transfer, and thus the program ran correctly. The kernel breakdown for the three different runs is depicted in Figure 8. As expected, the program with data cache flush exhibited degraded performance due to extensive delays of the data cache flush and the resulting reloading of required data into the data cache. On the other hand, the program with the data cache disabled retains mostly the increased performance in the custom instructions with AVS, how-

Table 2: Experimental results for the complete JPEG encoding and decoding chain of a 24-bit 1027x768 RGB picture.

Parameter:	Original code	Atasu	Biswas coherence violating	Biswas Data Cache flush	Biswas no Data Cache	Biswas corrected *M(C)*	Coherent DMA	Speculative DMA
IPC	0.66	0.6	0.47	0.21	0.21	0.6	0.42	0.53
D$ miss-rate [%]	0.29	0.33	0.52	7.20	100	0.33	2.77	1.32
Speedup	1	1.29	1.71	0.75	0.77	1.29	1.58	2.08
Execution time	1	0.78	0.59	1.33	1.31	0.78	0.63	0.48

ever drastic performance degradation for all other kernels is caused by the absence of the data cache.

As the current algorithm of Biswas does not provide us with a correct solution in presence of a data cache, we modified the merit function *M(C)* to take into account the overhead of data cache flushing. As expected, the algorithm now only detects constant tables for ISE inclusion, and the speed improvement is negligible if compared to Atasu's algorithm. To regain the benefits of Biswas algorithm, we introduced our coherent DMA, and profiled the corresponding program. Our coherent DMA scheme re-introduces the AVS into the custom instructions, resulting in similar speed-up as the Biswas algorithm presented in [4]. Due to the fact that our coherent DMA requires more cycles to transfer the data into and out-of the AVS, we loose some of the speed-up compared to Biswas. As can be seen in Table 2 the influence of the coherence protocol is clearly reflected in the data cache miss rate.

As the DMA placement of Biswas algorithm gives room for improvement, we introduced our profile based speculative DMA into the algorithm. The kernel-breakdown for this architectural version is shown in column eight of Figure 8. As can be seen in Table 1, we successfully removed redundant DMA transfers in the DCT-kernel, resulting in a 60% improvement in speed-up over Biswas original algorithm. More interesting however is the speed-up seen for the IDCT. As all algorithms fail to speed-up the IDCT kernel due to its software optimization, our profile based speculative DMA is able to capture all memories inside custom instructions due to a reduced cost function. Not only does our profile based speculative DMA provide program correctness, but it also gives us a speed-up of 9x for the IDCT kernel, and a speed-up of 2x over the complete application. Despite the coherence traffic overhead, as reflected in the data cache miss rate in Table 2, the profile based speculative DMA successfully outperforms all previous proposed algorithms, whilst providing program correctness in presence of one or more data caches.

6. CONCLUSIONS

We have shown that, in all but the most trivial cases, coherence is a serious concern when developing instruction set extensions with architecturally visible storage, and this problem has been often underestimated in prior art. Indeed, there is no miracle solution, and the best solution is to leverage existing cache coherence protocols which are now available in high-end embedded processors. Nevertheless, we show that adding coherence to the local memories is feasible with very limited hardware resources and that a number of optimizations are enabled by the coherence network. These optimizations achieve tangible speedup compared to conventional methods, including those which actually incur into coherence problems and are actually not guaranteed correct.

7. REFERENCES

[1] ARM Ltd. *ARM Cortex-A9 MPCore*. `http://www.arm.com/products/CPUs/`.

[2] K. Atasu, G. Dündar, and C. Özturan. An integer linear programming approach for identifying instruction-set extensions. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis*, pages 172–77, Jersey City, N.J., Sept. 2005.

[3] L. Benini, A. Macii, E. Macii, and M. Poncino. Synthesis of application-specific memory for power optimization in embedded systems. In *Proceedings of the 37th Design Automation Conference*, pages 300–303, Los Angeles, Calif., June 2000.

[4] 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.

[5] N. T. Clark, H. Zhong, and S. A. Mahlke. Automated custom instruction generation for domain-specific processor acceleration. *IEEE Transactions on Computers*, C-54(10):1258–70, Oct. 2005.

[6] D. E. Culler, J. P. Singh, and A. Gupta. *Parallel Computer Architecture: A Hardware/software Approach*. Morgan Kaufmann, San Mateo, Calif., 1999.

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

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

[9] M. S. Papamarcos and J. H. Patel. A low overhead coherence solution for multiprocessors with private cache memories. In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, pages 348–54, Ann Arbor, Mich., Jan. 1984.

[10] 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.

[11] S. Steinke, L. Wehmeyer, B.-S. Lee, and P. Marwedel. Assigning program and data objects to scratchpad for energy reduction. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, Paris, Mar. 2002.

[12] A. K. Verma, P. Brisk, and P. Ienne. Rethinking custom ISE identification: A new processor-agnostic method. In *Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*, pages 125–34, Salzburg, Sept. 2007.

[13] P. Yu and T. Mitra. Scalable custom instructions identification for instruction set extensible processors. In *Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*, pages 69–78, Washington, D.C., Sept. 2004.