Introduction of Architecturally Visible Storage in Instruction Set Extensions

Partha Biswas, *Member, IEEE*, Nikil D. Dutt, *Senior Member, IEEE*, Laura Pozzi, *Member, IEEE*, and Paolo Ienne, *Member, IEEE*

***Abstract*—Instruction set extensions (ISEs) can be used effectively to accelerate the performance of embedded processors. The critical and difficult task of ISE selection is often performed manually by designers. A few automatic methods for ISE generation have shown good capabilities but are still limited in the handling of memory accesses, and so they fail to directly address the memory wall problem. We present here the first ISE identification technique that can automatically identify state-holding application-specific functional units (AFUs) comprehensively, thus being able to eliminate a large portion of memory traffic from cache and the main memory. Our cycle-accurate results obtained by the SimpleScalar simulator show that the identified AFUs with architecturally visible storage gain significantly more than previous techniques and achieve an average speedup of 2.8× over pure software execution with a little area overhead. Moreover, the number of required memory-access instructions is reduced by two thirds on average, suggesting corresponding benefits on energy consumption.**

***Index Terms*—Application-specific processors, architecturally visible storage, instruction set extensions (ISEs).**

I. Introduction

THE DESIGN of embedded processors poses a great challenge due to a stringent demand for high performance, low-energy consumption, and low cost—a blend which is not often found in general purpose processors. On the other hand, since embedded processors are dedicated to a single application—or to a small set of them—unique possibilities arise for designers, who can exploit their knowledge of the application in order to achieve the aforementioned blend.

Generally, a cost-effective way to simultaneously speed up execution and reduce energy consumption is to delegate time-consuming tasks of the application to dedicated hardware, leaving less critical parts to traditional software execution. This can be achieved by adding application-specific functional units (AFUs) to the processor and instruction set extensions (ISEs) to the instruction set for executing the critical portions of the application on the AFUs.

Manuscript received April 17, 2006; revised August 1, 2006 and October 18, 2006. This paper was recommended by Guest Editor D. Sciuto.

P. Biswas is with The Mathworks, Inc., Natick, MA 01760 USA (e-mail: partha.biswas@gmail.com).

N. D. Dutt is with the Donald Bren School of Information and Computer Sciences, and Department of Electrical Engineering and Computer Science, University of California, Irvine, CA 92697 USA.

L. Pozzi is with the Faculty of Informatics, University of Lugano, 6904 Lugano, Switzerland.

P. Ienne is with the Ecole Polytechnique Fédérale de Lausanne, 1015 Lausanne, Switzerland.

Digital Object Identifier 10.1109/TCAD.2007.890582

Since time-to-market is an important feature for the success of embedded processors and manual selection of ISEs can be a very daunting task, automatic identification of ISEs for a given application is of extreme importance. Indeed, a few automated techniques that sometimes match the performance of an expert designer have been presented. However, limitations still exist, and in some cases, the proposed techniques are still far from achieving the desired results. In particular, an important limitation is the inability of dealing with memory operations and allowing internal storage inside AFUs. In fact, apart from some simple exceptions treated in [4], the existing techniques are not able to include operations that access memory—while it is well known that memory traffic reduction is always of vital importance for performance as well as energy efficiency.

In this paper, we present an innovative algorithm for automatic identification of ISEs with architecturally visible storage: We envision AFUs with small internal memory and propose a way to automatically detect and accelerate even those parts of the application that involve memory accesses. To show the effectiveness of our approach, we augment the SimpleScalar [26] processor with ISEs identified by our proposed algorithm on different applications and study the resulting improvements in performance and energy. Our cycle-accurate results show that adding architecturally visible storage to an AFU results in an increase in average application speedup over pure software execution from 1.4× to 2.8×. Furthermore, the number of accesses to cache and main memory is also reduced by 66%, which also hints a concomitant energy reduction. We also demonstrate an average energy reduction of 53% in a 32-KB data cache due to redirection of costly memory accesses into a tagless AFU-resident memory.

II. Architecturally Visible Storage

In the course of executing an application, data are copied across different storage units starting from main memory before finally residing in a register file [Fig. 1(a)]. A functional unit (or AFU in this case) reads the data from the register file for processing and then writes the result back into the register file. A scratchpad [as shown in Fig. 1(b)] with a much simpler design than a cache brings the data closer to the AFU. The closer the data resides with respect to a computational unit, the faster the processing time. This approach also reduces cache pollution by allowing the AFU to bypass the cache and read the data directly from the scratchpad. Consequently, this results in not only speedup but also energy reduction because majority of the cache accesses are redirected to a simpler energy-efficient scratchpad.

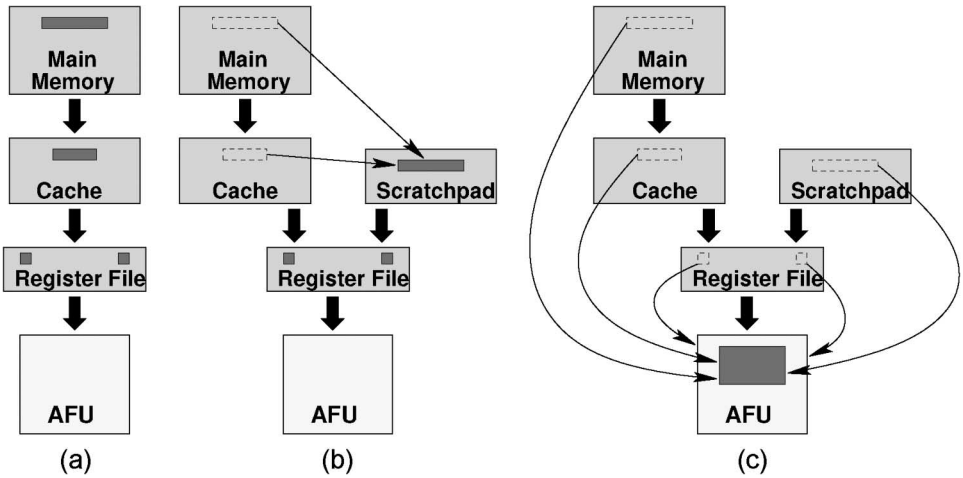


Fig. 1. (a) Data are copied from main memory and through cache and register file, before reaching the AFU. (b) Scratchpad helps reduce copies and pollution. (c) The local memory inside the AFU goes beyond previous achievements by bypassing even the register file and reducing copies and pollution to the minimum.

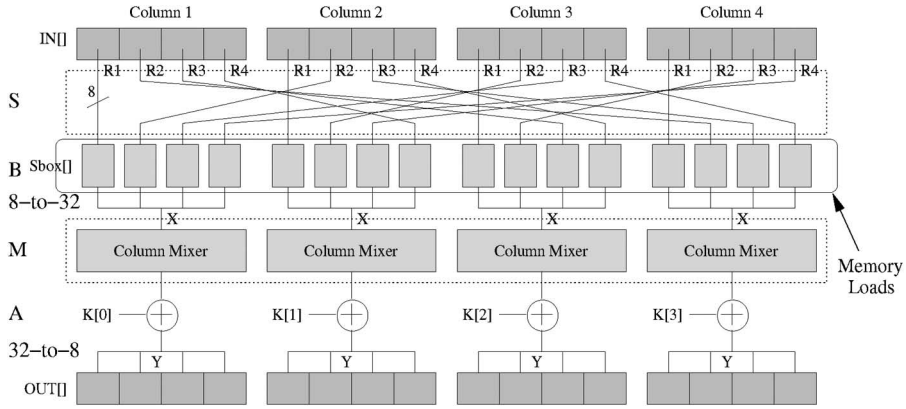


Fig. 2. AES main kernel.

We take this philosophy of bringing the data close to the site of computation to its limit—i.e., bring the data to be processed directly inside the AFU. By introducing a small internal memory inside the AFU, all the storage units are bypassed and we can get the dual benefit of speedup and energy reduction [Fig. 1(c)]. This internal memory is architecturally part of the AFU structure; therefore, we also call it a local architecturally visible storage.

III. Motivating Examples

The Advanced Encryption Standard (AES) benchmark is a Rijndael block cipher with a block/key size of 16 B. The stages involved in the AES encryption/decryption of a 16-B input are the following: 1) *shift rows (S)* as per a fixed scheme; 2) *byte substitution (B)* where each byte of the block is replaced by its substitute stored in a fixed 256-element array called *Sbox*[]; 3) *mix columns (M)* where each column stored in a 4-B block is multiplied with a constant matrix under some special rules involving multiply and XOR operations; and 4) *add round key (A)*, which also involves XOR operations.

The sequence of operations involved in the AES encryption is: $A-(S-B-M-A)^9-S-B-A$, indicating that the sequence S-B-M-A is executed in nine rounds presenting itself as a hot spot for optimizations. The basic stages of a round as implemented in the benchmark are captured in Fig. 2. The input 16-B block (conceptualized as a 4×4 matrix with 1-B entries) is realized as four blocks of *unsigned integer* (4 B each), which is shown as IN[] array in the figure. Except for the B-stage, all other stages comprise scalar operations that do not access memory. Unfortunately, contemporary techniques for ISE selection choose only the sections having scalar operations. However, an experienced architect on careful analysis of the B-stage would conclude that the memory operations simply reads from a small fixed table (*Sbox*[] of size 256) and, thus, it makes sense to map the table into the hardware. With a little overhead in area, this introduction of a hardware table having short and deterministic latency would generate a large performance gain. It is important to note that these instances are common in cryptographic benchmarks. The main goal of this paper is to steer the ISE design space exploration for generating results close to those achieved manually by an architect.

Many applications access small portions of memory multiple times in a frequently executed part of code. While previous techniques have attempted to move such memory accesses closer to the computational core (e.g., using scratchpad memories to reduce cache pollution), it is clear that we can gain significant benefit from moving such memory accesses

```
for (k = 1; k ≤ n; k++)
  n1 = 1<<k;
  n2 = n1>>1;
  ...
  for (j = 0; j < n2; j++)
    ...
    for (i = j; i < 2^n; i += n1)
      l = i + n2;
      tRealData = (WReal * RealBitRevData[l])
                + (WImag * ImagBitRevData[l]);
      tImagData = (WReal * ImagBitRevData[l])
                - (WImag * RealBitRevData[l]);
      tRealData = tRealData >> SCALE_FACTOR;
      tImagData = tImagData >> SCALE_FACTOR;
      RealBitRevData[l] = RealBitRevData[i]
                        - tRealData;
      ImagBitRevData[l] = ImagBitRevData[i]
                        - tImagData;
      RealBitRevData[i] += tRealData;
      ImagBitRevData[i] += tImagData;
```

Fig. 3. *fft* kernel (Courtesy of EEMBC).

directly into the computation core—i.e., *directly into the AFUs* [Fig. 1(c)]. For example, consider a portion of the *fft* kernel from the Embedded Microprocessor Benchmark Consortium (EEMBC) suite [27] shown in Fig. 3. The innermost loop is run $2^n/2^k$—i.e., 2^{n-k} times. Therefore, for each k, there are $2^{k-1} \cdot 2^{n-k}$ or 2^{n-1} accesses to memory. For $n = 8$, k goes from 1 to 8, leading to $8 \cdot 127 = 1024$ memory accesses for each array variable in the critical region. Since there are six memory reads and four memory writes corresponding to array variables *RealBitRevData*[] and *ImagBitRevData*[], there are 6144 memory reads and 4096 memory writes in the *fft* kernel for $n = 8$.

Existing automatic ISE techniques would identify instructions composed of data-flow and nonmemory-access operations, such as the butterfly, leaving the memory accesses to the processor core. However, if the *fft* kernel executes in an AFU with a small local memory with a storage space for 256 elements, all 10 240 accesses to main memory can be redirected to the fast and energy-efficient AFU-resident local memory.

In general, the advantages of an AFU-resident memory are manifold: it lowers cache pollution, it increases the scope of ISE algorithms, it increases the resulting performance, and it reduces energy consumption. A previous work [4] exploited the presence of memory elements in ISEs, in only two special forms: 1) hardware tables and 2) architecturally visible state registers. We present a formal framework for automatically exploiting any kind of AFU-resident memory during ISE generation.

IV. Related Work

Most related research efforts in automatic ISE, such as [1]–[3], [5], [7], [8], [13]–[15], do not allow memory instructions to be selected in the acceleration section and thus do not consider either memory ports in AFUs or AFU-resident memory. Thus, they miss the speedup opportunities enabled for the first time in this paper. One recent work indeed considered memory inside AFUs [4] but only in very special cases, namely, in the cases of read-only memory and loop-carried scalars. Similar to this approach, a contemporary work [17] also incorporates read-only memory to increase the scope of ISEs. On the other hand, in this paper, we present a general formulation that considers *any kind of vector or scalar access* without restriction. Our solution, in fact, encompasses the special cases treated in [4].

One interesting approach [18], [19] that addresses the memory wall problem closely matches our approach. This approach increases the memory throughput of a computation by making the architecture communication-aware. This is achieved by either enhancing the memory subsystem with an application-specific hardware [18] or placing an arithmetic logic unit next to the static random-access memory (SRAM) [19] to speed up computations. However, our goal is to speed up application through complex ISEs that can now also include memory operations. Allowing memory operations inside ISEs increases the complexity of the ISE identification problem.

Program-In Chip-Out Nonprogrammable Accelerator [20] bears some similarity with this paper as its architectural model permits the storage of reused memory values in accelerators. However, it does not present a method for identifying the portions of application code to be mapped on the accelerator; that is left to a manual choice, while we present an automated approach. Another work in reconfigurable computing [9] considered automatically selected coprocessors with direct memory access. On the other hand, our technique identifies whole arrays or scalars to be loaded into an AFU and furthermore permits the processor to access the AFU memory directly (rather than the main memory) during inner loop execution. This is an innovative proposal, which was not considered in prior work; our experimental results prove its effectiveness.

Register Promotion [21] is a compiler technique that aims at reducing memory traffic by promoting memory accesses to register accesses. However, previous efforts have not used it in the context of ISEs, where memory accesses can instead be eliminated by AFU residency—i.e., both data flow computation and memory accesses are identified together and delegated to special computation units, bypassing even the register file. Finally, the contributions presented here bear some resemblance with a recent work on scratchpads [11] and with one using application-specific memories instead of caches [12]. We go beyond such approaches by bringing portions of storage closer to the core—directly inside the AFU that is going to use them [as shown in Fig. 1(c)]. In this paper, we not only discuss our memory-aware ISE generation approach in detail but also expand on our previous work [16] to present 1) an analysis to justify the scheme employed for maintaining consistency between the main memory and the internal memory inside the AFUs and 2) additional experimental results to demonstrate the energy benefits of including architecturally visible state in AFUs. Furthermore, our experimental results also show that incorporating local memories inside AFUs has a fairly low area overhead.

V. Memory-Aware ISE Identification

We first introduce a general formulation of the ISE identification problem [2], and then we list the differences required to

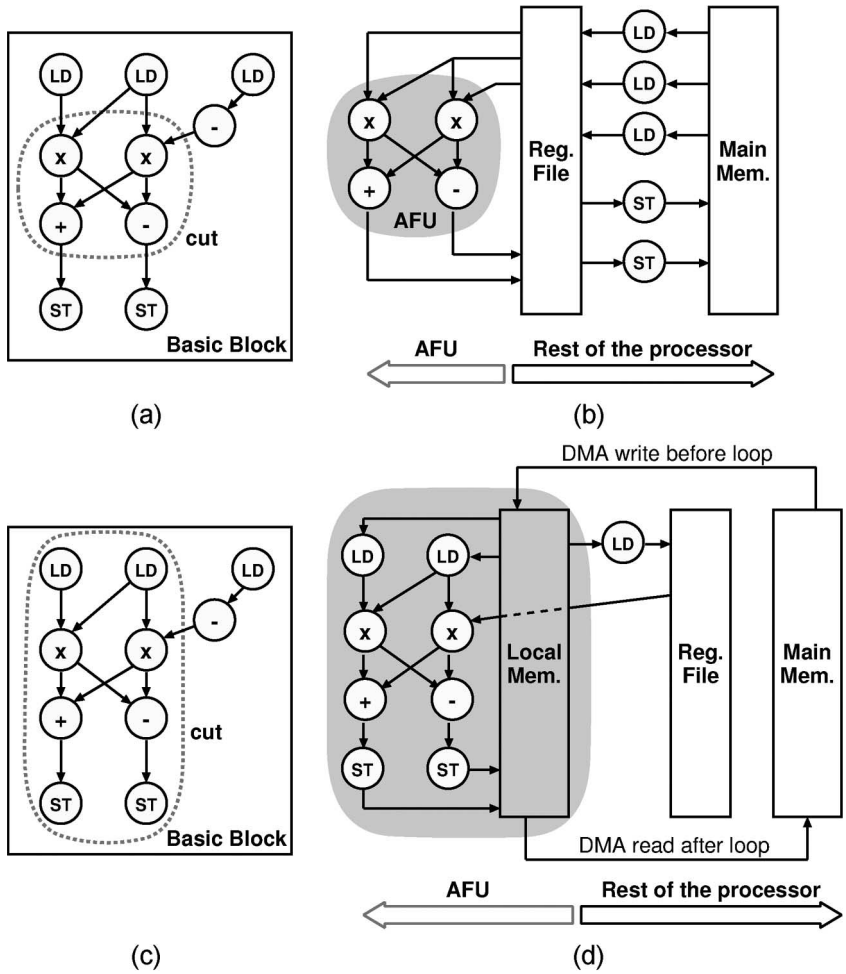


Fig. 4. In a previous work, (a) a cut could not include any memory access nodes (ld/st), and (b) the corresponding AFU did not hold a state; the AFU fetched all operands from the register file. Now, (c) a cut can include memory-access operations to a vector, and (d) the corresponding AFU has a copy of the vector in its internal memory; all memory operations in this basic block access the AFU internal memory instead of main memory.

identify memory holding ISEs. We call $G(V, E)$ the directed acyclic graphs representing the data flow of a critical basic block (cbb); nodes V represent primitive operations, and edges E represent data dependencies. A *cut* C is a subgraph of $G : C \subseteq G$. A function $M(C)$ measures the merit of a cut C and represents an estimation of the speedup achievable by implementing C as a special instruction.

We call IN(C) and OUT(C) the number of inputs and outputs, respectively, of cut C, while values N_{in} and N_{out} indicate the number of register-file read and write ports, respectively, which can be used by the special instruction. In addition, due to microarchitectural constraints, operations of a certain type might not be allowed in a special instruction. We call F (with $F \subseteq V$) the set of *forbidden* nodes that should never be part of C.

The identification problem is formally stated as follows.

Problem 1: Given a graph G and the microarchitectural features N_{in}, N_{out}, and F, find the cut C that maximizes $M(C)$ under the following constraints:

1) IN$(C) \leq N_{\text{in}}$;
2) OUT$(C) \leq N_{\text{out}}$;
3) $F \cap C = \emptyset$;
4) C is convex.

The first two constraints guarantee I/O feasibility, the third one disallows inclusion of forbidden nodes, and the last constraint ensures that all inputs are present at the time of issue. When considering state-holding ISEs (rather than purely combinational ones as in [2]), *two features must be adapted*, namely 1) the content of F and 2) the definition of $M(C)$. Ideally, all memory access nodes can now be excluded from set F, i.e., they can be included in a cut (in practice, we apply a compiler pass to exclude from F all accesses to vectors and loop-carried scalars; pointer accesses are still not treated at present). The merit function $M(C)$ must take into account the cost of transferring data between the AFU memory and the main memory.

A. Architectural Organization

If all memory accesses are forbidden in C, as in previous work, the envisioned situation is that of Fig. 4(a): A cut can only contain data flow operations. Fig. 4(b) describes the architectural side: The load/store unit of the processor affects the transfers between register file and main memory, and the AFU fetches its operands from the register file—like any functional unit. However, when memory accesses to some vector are allowed in a cut, as shown in Fig. 4(c), a state-holding AFU is taken into consideration. A state-holding AFU can also include

scalar accesses; these can be treated as a special case of vectors. Fig. 4(d) shows the AFU corresponding to the cut chosen: A copy of the vector is resident in the internal memory and *all* memory accesses to that vector in the basic block—whether included in the cut or not—access the local AFU memory rather than the main memory. Architecturally, in the most general case, the vector in question needs to be transferred from the main memory to the AFU local memory by *direct memory access (DMA)* before loop execution (i.e., before executing the cbb). As a result, all memory accesses to the vector are now transformed into internal accesses to the local memory instead. At the end of the loop (i.e., after executing the cbb), the vector is copied back to main memory—only if needed. The applicability of this approach is clearly limited by the number and size of the vectors accessed within the cbb. However, note the much decreased register file pressure (only one register read) and reduced main memory access (only two accesses) in the example of Fig. 4(d).

B. Merit Function and Problem Solution

In the following, cbb refers to the basic block for which an ISE is currently being identified.

The merit function $M(C)$ per unit execution of cbb is expressed as follows:

$$M(C) = \lambda_{\mathrm{sw}}(C) - \lambda_{\mathrm{hw}}(C) - \lambda_{\mathrm{overhead}}(C) \qquad (1)$$

where $\lambda_{\mathrm{sw}}(C)$ and $\lambda_{\mathrm{hw}}(C)$ are the estimated software latency (when executed natively in software) and hardware latency (when executed on an AFU) of cut C, respectively, and $\lambda_{\mathrm{overhead}}$ estimates the transfer cost. Consider a DMA latency of λ_{DMA} and suppose that the DMA write and read operations required will be placed in a write basic block (wbb) and a read basic block (rbb), whose execution counts are N_{wbb} and N_{rbb}, respectively (ideally much smaller than the execution count of cbb N_{cbb}, where the ISE is identified).

The transfer cost can be expressed as

$$\lambda_{\mathrm{overhead}} = \frac{N_{\mathrm{wbb}} + N_{\mathrm{rbb}}}{N_{\mathrm{cbb}}} \cdot \lambda_{\mathrm{DMA}}.$$

Note that all the aforementioned considerations are valid not only for vectors but also for inclusion of scalar accesses. However, in the case of scalar accesses, the transfer will be much cheaper as it does not involve DMA setup and transfer overhead. In the rest of this paper, we will use the term "memory transfer" for both vectors and scalars.

For a given cbb, the steps for generating ISEs that include architecturally visible storage are given as follows. 1) Find vectors and scalars accessed in cbb; for this, we can use some well-known static memory disambiguation techniques [23]. 2) Search for the most profitable code positions for inserting memory transfers between the AFU and the main memory—this is a fundamental problem, and the solution is discussed in the next section. 3) Run ISE identification. We use the ISE identification algorithm presented in [2], which is an improvement over [1] and can optimally solve problem 1 for basic blocks of approximately 1000 nodes. In the algorithm, we use the updated merit function $M(C)$ expressed in (1) to evaluate the merit of a selected cut. The pruning criterion in [2] had to be relaxed in order to handle memory awareness correctly.

C. Scheduling Data Transfers

To ensure profitability of memory inclusion in an AFU, memory transfer operations between main memory and the AFU must be performed in basic blocks with the least possible execution count. However, they must be performed in basic blocks that ensure semantic correctness of the program. Now, we will discuss how to optimize the insertion of a DMA write operation (transfers from main memory to AFU); insertion of DMA read (transfers from AFU to main memory) requires a very similar and dual procedure.

Intuitively, for correctness, a DMA write should be inserted in a basic block wbb 1) that is situated after any basic block that modifies the vector to be included in the AFU and 2) always reaches cbb in the control flow. Therefore, after identifying accessed vectors and scalars v in cbb, for every v, we execute the following steps to determine the basic block wbb—a node in the control flow graph (CFG) —where a DMA write should be placed.

1) Determine the set P (polluters) of nodes where v is written, excluding cbb, such that cbb is reachable from each node in P.
2) For each node $p \in P$, determine the set of nodes *reachable* by it including p, which is indicated as R_p. Such a set can be obtained in linear time by traversing the CFG.
3) Determine the set D_{cbb} of basic blocks that strictly dominate cbb. A node n_1 strictly dominates a node n_2 if every path from the procedure entry node to n_2 passes through n_1 and $n_1 \neq n_2$. This set can be computed in polynomial time by traversing the CFG [24].
4) Compute the intersection of all sets: $S_{\mathrm{wbb}} = D_{\mathrm{cbb}} \cap R_{p_1} \cap R_{p_2} \cap \ldots$, with $p_1, p_2, \ldots \in P$. This represents the set of nodes where it is correct to place a DMA write operation.
5) Choose the node in S_{wbb} with the least execution count—this is wbb. If $S_{\mathrm{wbb}} = \phi$, the DMA write is not required.

Fig. 5 illustrates the algorithm as applied to the *fft* example. Fig. 5(a) depicts the CFG of the application, where the entry node is 0, cbb was identified as node 10, and the set of polluter nodes P consists of node 3 only. Fig. 5(b) shows the set R_3, which contains nodes 2–16. Fig. 5(c) depicts the set D_{cbb}, which consists of nodes $\{0, 1, 2, 4, 5, 6, 7, 8, 9\}$. $S_{\mathrm{wbb}} = \{2, 4, 5, 6, 7, 8, 9\}$ represents the set of nodes where it is *correct* to insert a DMA write operation. The node where it is *correct and most beneficial* to insert the DMA write is the one with the least execution count, which in this case is node 4 (execution counts, which are not shown here, are gathered with profiling). Therefore, node 4 is finally chosen as wbb, and a DMA write is placed there.

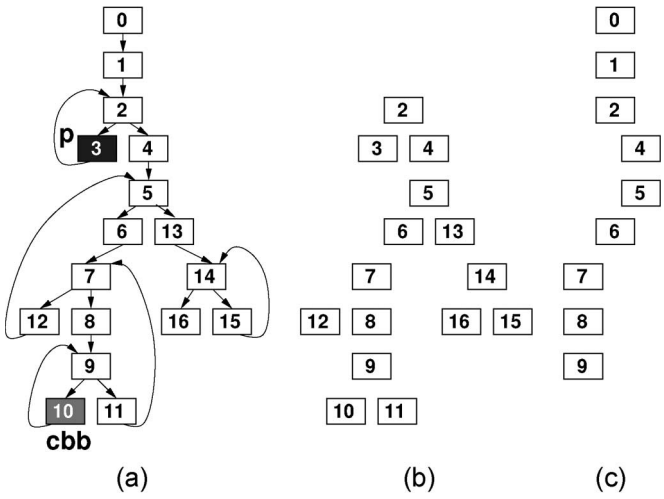


Fig. 5. For the *fft* example. (a) Control flow graph. (b) Set R_3: the set of nodes that can be reached by polluter node 3. (c) Set D_{10}: the set of nodes that strictly dominate node 10.

VI. Other Solutions: Further Analysis

In our analysis for vectors inside the AFU, we adopt a conservative approach by assuming that corresponding to a given array, each store operation creates a dependence for subsequent memory operations. When some memory accesses go to the AFU memory and others go to the main memory, special care is taken so that the contents of the internal AFU memory and that of the external main memory are consistent.

A. Maintaining Storage Consistency

As shown in Fig. 6, there are, in particular, three ways of maintaining this consistency.

1) *Consistent AFU memory* [Fig. 6(a)]: This corresponds to the case discussed so far in the previous section. Here, the consistency between the AFU memory and the main memory is achieved by assuring a consistent state within the AFU. This is done by transforming each store *st* in software into a special store *st_loc* from a register to the local memory (instead of the main memory). Likewise, each load *ld* in software is changed into a special load *ld_loc* into a register from the local memory (but not the main memory). These special load and store instructions are meant to be added as new instructions to the instruction set. In order to maintain a consistent state while entering the basic block, a DMA write into the local memory may be executed to copy all the array elements into the AFU memory. Similarly, a DMA read from the local memory into the main memory may be executed in order to maintain a consistent state while exiting the basic block. As explained before, such DMA operations of copying the whole array between the local memory and the main memory take place in the basic block having the least execution count. Other loads [*ld*] and stores [*st*] inside the AFU operate on a consistent AFU memory.
2) *Consistent main memory* [Fig. 6(b)]: The second way of maintaining storage consistency is to allow AFU access to the main memory. Because the AFU directly accesses data in the main memory, the loads *ld* and stores *st* that are in software get the consistent state of the main memory. In this case, there is no local storage inside the AFU.
3) *Consistent main memory and AFU memory* [Fig. 6(c)]: Another way of maintaining consistency is by always storing the data in both the main memory and the local memory. If there is a nonzero number of loads inside the AFU, an initial consistent state can be maintained by writing the whole array into the AFU before the loop.

B. Analysis

The effective speedup brought about by the memory part of the cut can be obtained by subtracting the overhead of data transfer from the speedup contribution of the memory part. Let the speedup contribution from the memory part of C be $S_{\mathrm{mem}}(C)$ and the overhead due to data transfer (by DMA) be $\lambda_{\mathrm{overhead}}(C)$. Thus, the effective speedup due to migration of memory operations into the AFU can be expressed as follows:

$$S_{\mathrm{mem}}(C) - \lambda_{\mathrm{overhead}}(C).$$

For the sake of analyzing the effective speedup possible in the three different configurations for maintaining storage consistency, we represent the load/store operations inside a cut C as enclosed within square braces. Let *ld* and *st* refer to load and store operations in software, and let [*ld*] and [*st*] refer to those inside the cut, respectively. Let the vector under consideration be A, and unless otherwise stated, all the memory references are with respect to A. In the following, we determine S_{mem} and $\lambda_{\mathrm{overhead}}$ for the aforementioned three configurations possible for maintaining storage consistency.

1) *Consistent AFU memory*: $S_{\mathrm{mem}}(C)$ has two components, namely 1) speedup due to moving the local memory inside the AFU, getting rid of the costly memory accesses and 2) speedup due to special memory operations (*ld_loc* and *st_loc*), which eliminate access to the main memory. Thus,

$$S_{\mathrm{mem}}(C) = \left(\left(\lambda_{\mathrm{ld}} \cdot N_{[\mathrm{ld}]} + \lambda_{\mathrm{st}} \cdot N_{[\mathrm{st}]}\right) - \lambda_{\mathrm{hw}}^{\mathrm{mem}}\right) + \left(\left(\lambda_{\mathrm{ld}} - \lambda_{\mathrm{ld\_loc}}\right) \cdot N_{\mathrm{ld}} + \left(\lambda_{\mathrm{st}} - \lambda_{\mathrm{st\_loc}}\right) \cdot N_{\mathrm{st}}\right).$$

In the preceding expression, $\lambda_{\mathrm{hw}}^{\mathrm{mem}}$ refers to the critical path contributions due to memory operations inside the cut (i.e., [*ld*] and [*st*]). Since the local memory is integrated within the AFU, $\lambda_{\mathrm{hw}}^{\mathrm{mem}} = 0$. Let *loads* and *stores* refer to all the loads and stores within the basic block, respectively. Therefore, $N_{\mathrm{loads}} = N_{\mathrm{ld}} + N_{[\mathrm{ld}]}$, and $N_{\mathrm{stores}} = N_{\mathrm{st}} + N_{[\mathrm{st}]}$. Simplifying the preceding expression, we get

$$S_{\mathrm{mem}}(C) = \left(\lambda_{\mathrm{ld}} \cdot N_{\mathrm{loads}} + \lambda st \cdot N_{\mathrm{stores}}\right) - \left(\lambda_{\mathrm{ld\_loc}} \cdot N_{\mathrm{ld}} + \lambda_{\mathrm{st\_loc}} \cdot N_{\mathrm{st}}\right).$$

As presented before, the overhead of data transfer is

$$\lambda_{\mathrm{overhead}}(C) = \frac{N_{\mathrm{wbb}} + N_{\mathrm{rbb}}}{N_{\mathrm{cbb}}} \cdot \lambda_{\mathrm{DMA}}.$$

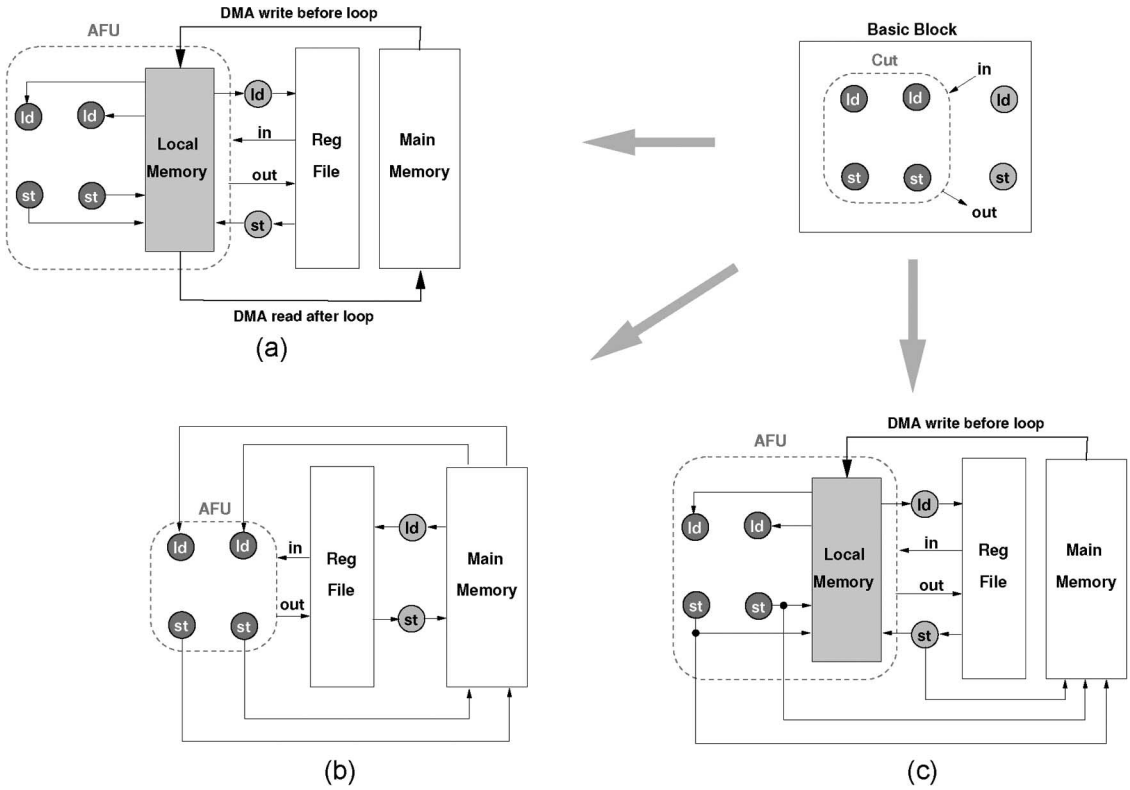


Fig. 6. Three configurations for maintaining consistency: (a) Consistent AFU memory. (b) Consistent main memory. (c) Consistent main memory and AFU memory.

2) *Consistent main memory*: If the main memory is accessed from the AFU in the same way as it is from software, both $S_{\text{mem}}(C)$ and $\lambda_{\text{overhead}}(C)$ are zero. Therefore, the effective speedup due to the memory part in this case is also zero.
3) *Consistent main memory and AFU memory*: Here, only the load operations can contribute to speedup by being inside the cut because all the store operations (whether inside or outside the cut) write into both the local memory and the main memory. Thus,

$$S_{\text{mem}}(C) = \left(\lambda_{\text{ld}} \cdot N_{[\text{ld}]} - \lambda_{\text{hw}}^{\text{mem}}\right) - \lambda_{\text{st\_loc}} \cdot N_{\text{stores}}.$$

Since $\lambda_{\text{hw}}^{\text{mem}} = 0$,

$$S_{\text{mem}}(C) = \left(\lambda_{\text{ld}} \cdot N_{[\text{ld}]}\right) - \left(\lambda_{\text{st\_loc}} \cdot N_{\text{stores}}\right).$$

Clearly, the speedup contribution due to the local memory inside the cut is reduced by the stores that also write into the local memory along with the main memory. The DMA overhead is only because of the DMA writes before entering the cbb. Thus,

$$\lambda_{\text{overhead}}(C) = \frac{N_{\text{wbb}}}{N_{\text{cbb}}} \cdot \lambda_{\text{DMA}}.$$

Having discussed the effective speedup contributions in the three possible scenarios for maintaining storage consistency, we should adopt a scheme that results in the greatest speedup. The second case does not yield any speedup; therefore, we compare only the first and the third cases.

Since $N_{\text{loads}} > N_{[\text{ld}]}$, and $N_{\text{stores}} > N_{\text{st}}$, $S_{\text{mem}}(C)$ in the first case is evidently greater. Because the overhead due to DMA transfer in most cases is hidden by other instructions, the first case yields the highest effective speedup in the memory part of the cut. Therefore, we adopted the first scheme of maintaining AFU-memory consistency for introducing architecturally visible state inside the AFU.

VII. Experiment

We implemented our memory-aware ISE generation algorithm on the MACHSUIF [25] framework. We used six benchmarks to demonstrate the effectiveness of our approach: adaptive differential pulse code modulation decoder (ADPCM decoder, *adpcm-d*), ADPCM encoder (*adpcm-e*), fast Fourier transform (*fft*), finite-impulse response filter (*fir*), Data Encryption Standard (*des*), and Advanced Encryption Standard (*aes*), which are taken from Mediabench, EEMBC, and cryptography standards. We chose the cycle-accurate SimpleScalar simulator [26] for the ARM instruction set and modified it as follows: For vectors, we introduced a DMA connection between the local memory inside an AFU and the main memory by adding four new instructions to the instruction set, namely: 1) set source address (*ssa*); 2) set command register for transferring data to the AFU memory (*scmda*); 3) set destination address (*sda*); and 4) set command register for transferring data to main memory (*scmdm*); two instructions are for setting the source and

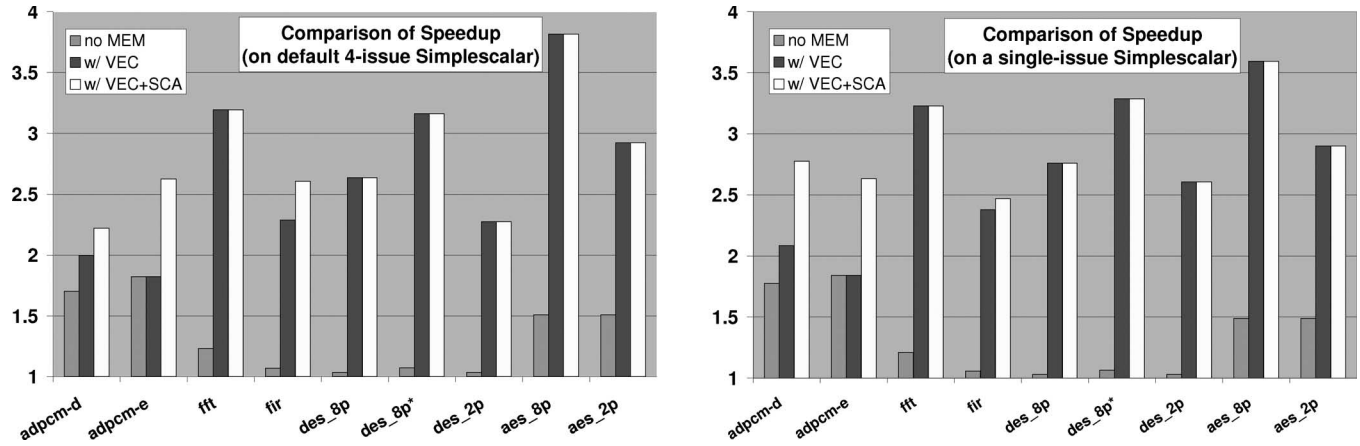


Fig. 7. Comparison of speedup for I/O constraints of 4/2 obtained on a four-issue (default) and a single-issue SimpleScalar with the ARM instruction set.

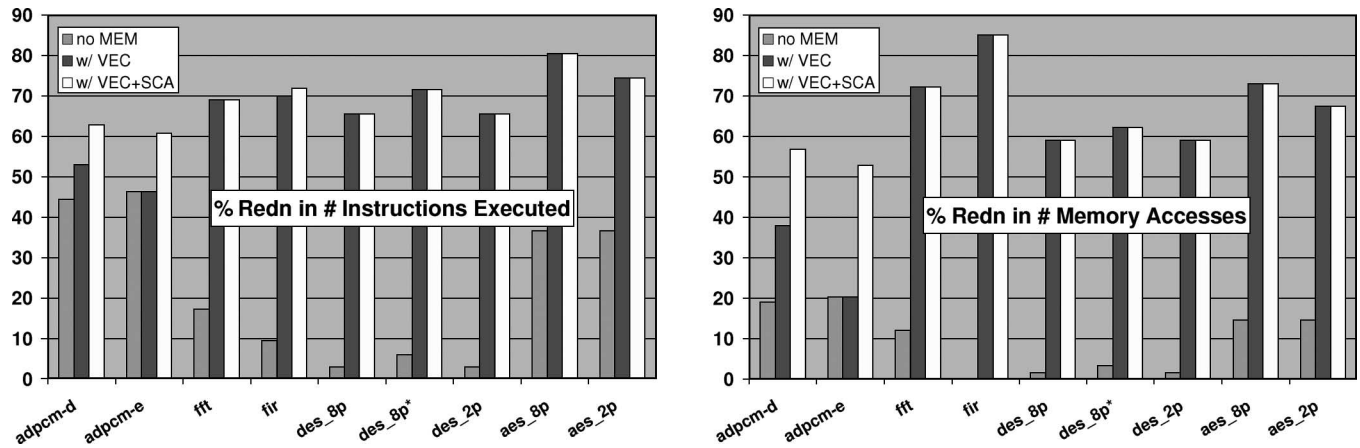


Fig. 8. Percentage reduction in the number of instructions executed and the number of memory accesses.

destination addresses, and two are for setting the command registers to transfer data from main memory to the AFU memory and vice versa. For handling scalars, two additional instructions were added to set and get local registers inside the AFU. Of course, we also added the application-specific ISEs identified by our ISE generation algorithm.

The hardware latency for each instruction was obtained by synthesizing the constituent arithmetic and logic operators on the Artisan UMC 0.18-μm CMOS process using the Synopsys Design Compiler. The access latency of the internal memory (modeled as an SRAM) was estimated using the Artisan UMC 0.18-μm CMOS process SRAM Generator. The default SimpleScalar architecture has four integer ALUs, one integer multiplier/divider, four floating-point adders, one floating-point multiplier/divider, and a three-level memory hierarchy for both instruction and data. The sizes of L1 and L2 data caches are 2 and 32 kB, respectively. The main memory has a latency of 18 cycles for the first byte and 2 cycles for subsequent bytes. The same latency is also used when transferring data between main memory and AFU by DMA.

A. Performance Gain

Our baseline case is pure software execution of all instructions. We set the I/O constraints to four inputs and two outputs and generated *a single cut* to be added as an ISE to the SimpleScalar architecture. First, we generated the cut without allowing memory inclusion ("no MEM"). Then, we allowed local memory inside with vector accesses ("w/VEC") and subsequently with scalar accesses also ("w/VEC+SCA"). For these three cases, we show in Fig. 7 a comparison of speedup on several applications obtained on the default SimpleScalar architecture (four-width out-of-order issue) as well as on the single-issue SimpleScalar architecture.

Observe that: 1) the speedup is raised tangibly when state-holding AFUs are considered (1.4× on average for the case with no memory to 2.8× for the "w/VEC+SCA" case, on the default architecture) and 2) the trend of speedups obtained on the two different configurations of the SimpleScalar architecture is the same. The label $des*$ indicates the results for *des* with three ISEs rather than with a single one (*des* is the only benchmark where a single cut was not enough to cover the whole kernel). The suffixes "_8p" and "_2p" are used for *des* and *aes* to indicate that the speedups correspond to the internal AFU memory having eight ports and two ports, respectively.

The corresponding speedups on adding local memory and then state registers inside the AFU on the single-issue SimpleScalar are 2.65× and 2.9×. Fig. 8 shows the reduction in the number of instructions executed and in the number of memory accesses. Interestingly, there is an average 9% reduction

TABLE I
SUMMARY OF LOCAL MEMORIES SELECTED FOR THE DIFFERENT BENCHMARKS

Benchmarks	Array Identifier	Size (bytes)	# read-ports
adpcm-d	stepsizeTable	356	1
	indexTable	64	1
adpcm-e	stepsizeTable	356	1
	indexTable	64	1
fft	RealBitRevData	512	2
	ImagBitRevData	512	2
fir	inbuf16	528	2
	coeff16	128	2
des	des_SPtrans	2048	8
aes	Sbox	256	8

in memory operations even before incorporating memory inside the AFU. This is because the ISEs generally reduce register need (multiple instructions are collapsed into one) and therefore reduce spilling. With the incorporation of memory inside the AFU, the average reduction in memory instructions is a remarkable two thirds, hinting a very tangible energy reduction. Note that by handling vectors and scalars in a unified manner, our results clearly subsume the merits of including read-only memory and state registers as presented in [4]. The benchmarks *adpcm-d*, *adpcm-e*, and *fir* very clearly show the advantage of including scalars by exhibiting a marked increase in speedup due to the increased scope of ISE.

Table I shows that the sizes of the vectors incorporated in the AFU for the given benchmarks are fairly limited. The maximum number of read ports in the AFU internal memory is determined by the number of parallel loads from the memory that take place inside the AFU. The last column of Table I shows the number of read ports required for the chosen application in the worst case. As evident from Fig. 9, the area overhead of the local memories introduced for the chosen benchmarks is minimal—on average, only about 5% of the area occupied by a 32-kB direct-mapped cache. Using the Artisan UMC 0.18-μm SRAM Generator, we evaluated the area overhead, which correctly accounts for the number of ports in the AFU memory. The AFU memory uses Artisan single-port or dual-port SRAM. When more than two ports are needed, the memory reads are sequentialized using an approach similar to register file port sequentialization, as described in a previous work [6]. The corresponding reduction in performance due to such sequentializations is clearly depicted in Fig. 7 (*des*_8p to *des*_2p and *aes*_8p to *aes*_2p).

Note here that if more memory ports are available, one can easily avoid sequentialization of memory reads. In the case of multiple read ports, we can use a design scheme similar to that used in the Alpha 21624 processor [22] for increasing the read ports of its integer register file. The basic idea is to make more read ports available by using several replicas of a memory, with all write ports in parallel (allowing all the memory replicas to have the same content at all times). With this scheme, the performance of *des* and *aes* would increase from 2.2× (*des*_2p) to 2.6× (*des*_8p) and from 2.9× (*aes*_2p) to 3.8× (*aes*_8p), respectively, for the default SimpleScalar (four-issue) configuration. However, this advantage comes at a considerable cost because the area overhead for *des* and *aes* grows from 12.7% to 51% (of the cache area) and from 3.5% to 14% (of the cache area), respectively.

B. Energy Reduction

Our technique is also effective in reducing energy consumption because of three primary reasons: 1) A significant number of data-cache accesses are redirected to small tagless AFU-resident memory. 2) The number of fetches is reduced as a result of compaction of a large number of instructions into an ISE. 3) There is lesser number of instructions executed after encapsulating a set of instructions as ISE. The latter two reasons are sufficient for expecting energy reduction in any ISE generation approach. However, the first reason pertains only to the inclusion of memory within an AFU. Therefore, we will analyze the energy reduction, taking only the first feature into account. We consider a 32-kB direct-mapped data cache (with 2048 lines of 16 B each) as used in typical implementations of ARM for energy efficiency.

After mapping the state-holding ISEs to AFUs, let the number of accesses to the AFU local memory be N_A and the number of loads/stores directed to the cache be N_C. Thus, the number of accesses to the cache when there are no AFUs or for an AFU that does not contain local memory is $(N_A + N_C)$. If we represent the energy per access for the AFU memory and cache as E_A and E_C, respectively, the energy saving due to the AFU memory can be expressed as

$$\frac{(N_A + N_C) \cdot E_C - (N_A \cdot E_A + N_C \cdot E_C)}{(N_A + N_C) \cdot E_C}.$$

We characterized both the cache and the AFU-resident memory using Artisan UMC 0.18-μm technology and found the ratio $(E_C - E_A)/E_C$ to be 0.795 for the average size (1 kB) of the local memory in the chosen applications. Hence, the preceding expression simplifies into $(0.795 \cdot N_A)/(N_A + N_C)$. We present N_A, N_C, and percentage energy saving due to redirection of the data-cache accesses to the AFU memory in Table II.

Because the energy estimations are done under a number of conservative assumptions, we can safely say that the average energy reduction in the cache due to AFU-resident memory is at least 53%. Since the cache is a significant contributor of system energy, using local memory in AFUs would result in a perceptible overall system energy reduction. Note that our energy estimations do not include leakage energy consumption.

C. Expanded ISEs Identified for *fft* and AES

Fig. 10 shows the kernel of *fft*, with the omission of address arithmetic. Our memory-aware ISE identification algorithm found it profitable to include a small internal memory with a size of 1 kB (containing RealBitRevData and ImagBitRevData) inside the AFU. This AFU memory is filled using a DMA write before entering the *fft* kernel. With the local storage inside the AFU, a single cut now covers the entire *fft* kernel, which results in almost doubling the speedup obtained without memory in the AFU.

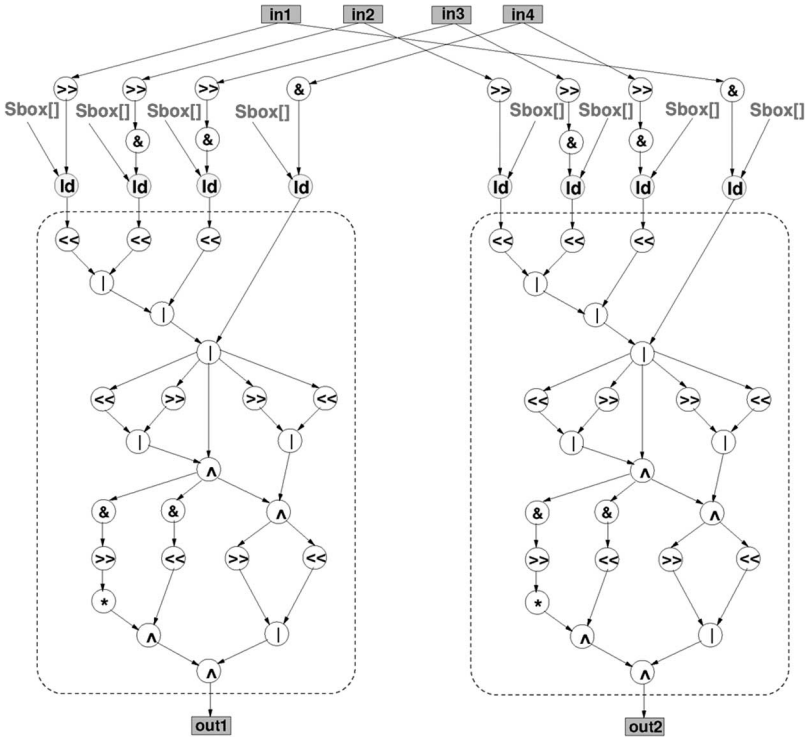


Fig. 9. Area overhead of the local memories introduced (presented as a percentage of area occupied by a 32-kB direct-mapped cache).

TABLE II
PERCENTAGE ENERGY SAVINGS ESTIMATED FOR A 32-kB CACHE WITH THE INTRODUCTION OF LOCAL MEMORY IN THE AFU FOR A FOUR-ISSUE AND ONE-ISSUE SIMPLESCALAR

BMs	4-issue SimpleScalar			1-issue SimpleScalar		
	N_A	N_C	% En Sav	N_A	N_C	% En Sav
adpcm-d	884960	673742	45.14	884518	820866	41.23
adpcm-e	836823	747771	41.98	887645	673213	45.21
fft	18426062	7096660	57.39	18442219	6790148	58.11
fir	25793	4565	67.55	25598	4244	68.19
des	127497	88670	46.89	128002	86700	47.4
aes	33220	12279	58.05	33217	11814	58.64

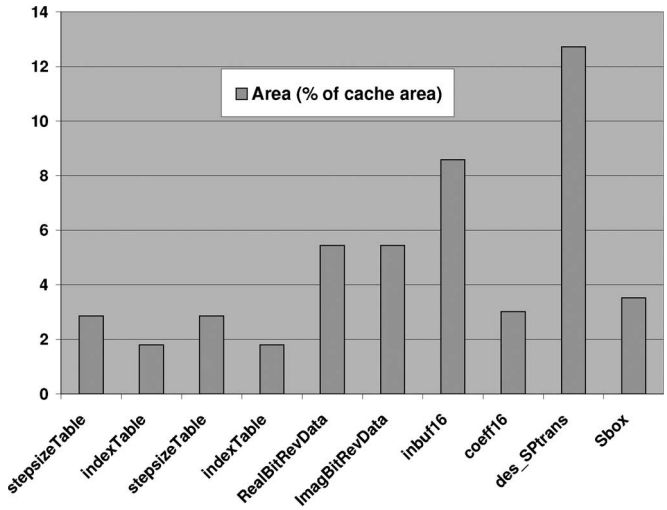


Fig. 10. Data flow graph of *fft*. The whole kernel is chosen when architecturally visible storage is allowed; only the cut in a dotted line is chosen otherwise.

In Fig. 11, we show the cut chosen for AES when memory operations are allowed in ISE generation. Because of going beyond memory barriers, a single cut encompasses a portion more than the portion chosen, with two cuts lacking memory inside the AFU. This graphically illustrates an increase in the scope of ISE because of including an architecturally visible state inside the AFU.

VIII. CONCLUSION

Embedded processors can be accelerated by augmenting their core with application-specific ISEs. Traditionally, memory access operations were either not included in automatically generated ISEs or were limited to special classes of memory operations; thus, ISEs were so far limited by the "memory wall" problem. This is the first comprehensive effort to overcome the memory wall problem for ISEs. Specifically, the main contributions presented in this paper are given as follows. 1) We show an architectural modification to include architecturally visible storage inside AFUs, clearly encompassing the special cases addressed by the previous work. 2) We introduce an algorithm to profitably identify code positions for inserting data transfers between the main memory and the local storage in ISEs. We demonstrate the effectiveness of our automated approach by incorporating the generated ISEs in the cycle-accurate SimpleScalar simulator.

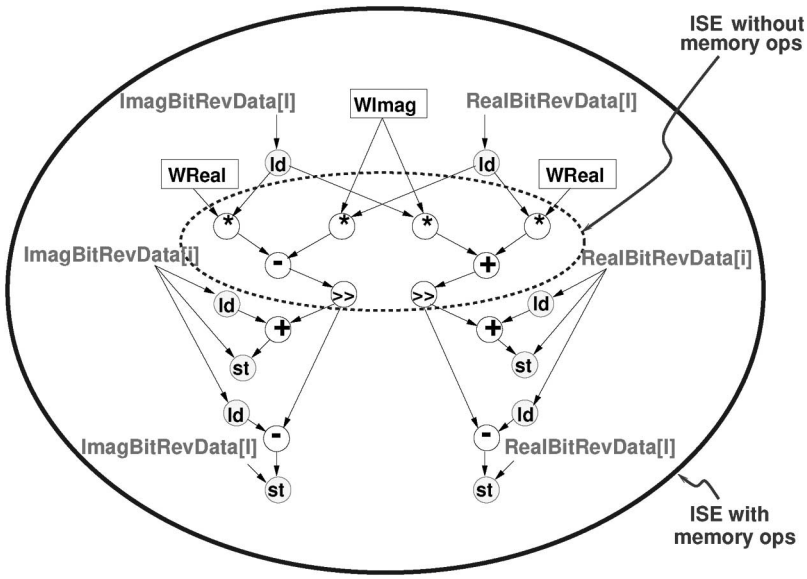


Fig. 11. Cut chosen in *aes*. The shown DFG portion is completely chosen when architecturally visible storage is allowed; otherwise, only the cut within a dotted line is chosen.

Our results show that the average speedup on a number of benchmarks increases from 1.4× to 2.8× by including architecturally visible storage in ISEs. Furthermore, an accompanied reduction of costly memory accesses by two thirds clearly highlights a concomitant reduction in energy. We showed an average energy reduction of 53% in a 32-kB data cache due to redirection of costly memory accesses into a tagless AFU-resident memory. We also showed that the area overhead of introducing these local memories is very moderate.

REFERENCES

[1] K. Atasu, L. Pozzi, and P. Ienne, "Automatic application-specific instruction-set extensions under microarchitectural constraints," in *Proc. DAC*, 2003, pp. 256–261.

[2] L. Pozzi, K. Atasu, and P. Ienne, "Optimal 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.

[3] J. Cong, Y. Fan, G. Han, and Z. Zhang, "Application-specific instruction generation for configurable processor architectures," in *Proc. FPGA*, 2004, pp. 183–189.

[4] P. Biswas, V. Choudhary, K. Atasu, L. Pozzi, P. Ienne, and N. Dutt, "Introduction of local memory elements in instruction set extensions," in *Proc. DAC*, 2004, pp. 729–734.

[5] P. Biswas, S. Banerjee, N. Dutt, L. Pozzi, and P. Ienne, "ISEGEN: Generation of high-quality instruction set extensions by iterative improvement," in *Proc. DATE*, 2005, pp. 1246–1251.

[6] L. Pozzi and P. Ienne, "Exploiting pipelining to relax register-file port constraints of instruction-set extensions," in *Proc. CASES*, 2005, pp. 2–10.

[7] N. Clark, H. Zhong, and S. Mahlke, "Processor acceleration through automated instruction set customization," in *Proc. MICRO*, 2003, pp. 129–140.

[8] F. Sun, S. Ravi, A. Raghunathan, and N. K. Jha, "Synthesis of custom processors based on extensible platforms," in *Proc. ICCAD*, 2002, pp. 641–648.

[9] T. Callahan and J. Wawrzynek, "Adapting software pipelining for reconfigurable computing," in *Proc. CASES*, 2000, pp. 57–64.

[10] R. Razdan and M. D. Smith, "A high-performance microarchitecture with hardware-programmable functional units," in *Proc. MICRO*, 1994, pp. 172–180.

[11] S. Steinke, L. Wehmeyer, B. S. Lee, and P. Marwedel, "Assigning program and data objects to scratchpad for energy reduction," in *Proc. DATE*, 2002, p. 409.

[12] L. Benini, A. Macii, E. Macii, and M. Poncino, "Synthesis of application-specific memory for power optimization in embedded systems," in *Proc. DAC*, 2000, pp. 300–303.

[13] N. Kavvadias and S. Nikolaidis, "Automated instruction-set extension of embedded processors with application to MPEG-4 video encoding," in *Proc. ASAP*, 2005, pp. 140–145.

[14] J. Cong, Y. Fan, G. Han, A. Jagannathan *et al.*, "Instruction set extension with shadow registers for configurable processors," in *Proc. FPGA*, 2005, pp. 99–106.

[15] K. Atasu, G. Dundar, and C. Ozturan, "An integer linear programming approach for identifying instruction-set extensions," in *Proc. CODES-ISSS*, 2005, pp. 172–177.

[16] P. Biswas, N. Dutt, P. Ienne, and L. Pozzi, "Automatic identification of application-specific functional units with architecturally visible storage," in *Proc. DATE*, 2006, pp. 1–6.

[17] R. Dimond, O. Mencer, and W. Luk, "Automating processor customisation: Optimised memory access and resource sharing," in *Proc. DATE*, 2006, pp. 1–6.

[18] C. Huang, S. Ravi, A. Raghunathan, and N. K. Zha, "High-level synthesis using computation-unit integrated memories," in *Proc. ICCAD*, 2004, pp. 783–790.

[19] P. Jain, G. E. Suh, and S. Devadas, "Intelligent SRAM (ISRAM) for improved embedded system performance," in *Proc. DAC*, 2003, pp. 869–874.

[20] R. Schreiber, S. Aditya, S. Mahlke, V. Kathail *et al.*, "PICO-NPA: High-level synthesis of nonprogrammable hardware accelerators," *J. VLSI Signal Process. Syst. Arch.*, vol. 31, no. 2, pp. 127–142, 2002.

[21] K. D. Cooper and J. Lu, "Register promotion in C programs," in *Proc. PLDI*, 1997, pp. 308–319.

[22] R. E. Kessler, "The alpha 21264 microprocessor," *IEEE Micro*, vol. 19, no. 2, pp. 24–36, Mar. 1999.

[23] D. M. Gallagher, "Memory disambiguation to facilitate instruction-level parallelism compilation," Ph.D. dissertation, Univ. Illinois at Urbana-Champaign, Urbana, IL, 1995.

[24] S. S. Muchnick, *Advanced Compiler Design and Implementation*. San Mateo, CA: Morgan Kaufmann, 1997.

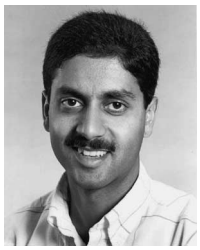
[25] *Harvard Machine SUIF*. [Online]. Available: http://www.eecs.harvard.edu/hube/software/software.html

[26] "The SimpleScalar Tool Set, Version 2.0," *Comput. Architecture News*, pp. 13–25, 1997.

[27] The EEMBC TeleBench, Version 1.1. [Online]. Available: http://www.eembc.org/benchmark/telecom.asp

Partha Biswas (M'01) received the B.Tech. degree in computer science and engineering from Indian Institute of Technology, Kharagpur, India, in 1998, and the M.S. and Ph.D. degrees in computer science from the University of California, Irvine, in 2002 and 2006, respectively.

He is currently with The Mathworks, Inc., Natick, MA. He was a Software Engineer and member of Technical Staff at Cadence Design Systems, Noida, India, from 1998 to 2000. His research interests include architectures and compilers for embedded systems, application-specific processor design, and high-level synthesis.



Nikil D. Dutt (S'84–M'89–SM'96) received the Ph.D. degree in computer science from the University of Illinois at Urbana-Champaign in 1989.

He is currently a Professor of computer science and electrical engineering and computer science with the University of California, Irvine, where he is with the following centers: CECS, CPCC, and CAL-IT2. His research interests include embedded systems design automation, computer architecture, optimizing compilers, system specification techniques, and distributed embedded systems.

Prof. Dutt received Best Paper Awards at CHDL89, CHDL91, VLSIDesign2003, CODES+ISSS 2003, CNCC 2006, and ASPDAC-2006. He currently serves as Editor-in-Chief of the *ACM Transactions on Design Automation of Electronic Systems* (TODAES) and as an Associate Editor of the *ACM Transactions on Embedded Computer Systems* (TECS). He was an ACM SIGDA Distinguished Lecturer during 2001–2002 and an IEEE Computer Society Distinguished Visitor for 2003–2005. He has served on the steering, organizing, and program committees of several premier CAD and embedded system design conferences and workshops, including ASPDAC, CASES, CODES+ISSS, DATE, ICCAD, ISLPED, and LCTES. He serves or has served on the advisory boards of ACM SIGBED and the ACM SIGDA, and is the Vice-Chair of IFIP WG 10.5.

Laura Pozzi (M'01) received the M.S. and Ph.D. degrees in computer engineering from the Politecnico di Milano, Milan, Italy, in 1996 and 2000, respectively.

From 2001 to 2005, she was a Postdoctoral Researcher with the School of Computer and Communication Sciences, Swiss Federal Institute of Technology Lausanne (EPFL), Lausanne, Switzerland. Previously, she was a Research Engineer with STMicroelectronics, San Jose, CA, and an Industrial Visitor with the University of California, Berkeley. She is currently an Assistant Professor with the Faculty of Informatics, University of Lugano (USI), Lugano, Switzerland. Her research interests include automating embedded processor customization, high-performance compiler techniques, and reconfigurable computing.

Prof. Pozzi received the Best Paper Award in the embedded systems category at the Design Automation Conference in 2003. She serves on the Technical Program Committee of the Conference on Compilers, Architectures and Software for Embedded Systems.

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

From 1990 to 1991, he was an Undergraduate Researcher with Brunel University, Uxbridge, U.K. From 1992 to 1996, he was a Research Assistant with the Microcomputing Laboratory (LAMI) and the MANTRA Center for Neuro-Mimetic Systems of the EPFL. In December 1996, he joined the Semiconductors Group of Siemens AG, Munich, Germany (which later became Infineon Technologies AG). After working on data path generation tools, he became the Head of the embedded memory unit in the Design Libraries division. In 2000, he joined EPFL, where he is currently a Professor and heads the Processor Architecture Laboratory (LAP). He is also a Cofounder of Mimosys, a company providing tools for the automatic customization of embedded processors. His research interests include various aspects of computer and processor architecture, reconfigurable computing, on-chip networks and multiprocessor systems-on-chip, and computer arithmetic.

Prof. Ienne was a recipient of the DAC 2003 Best Paper Award. He is or has been a member of the program committees of several international conferences and workshops, including Design Automation and Test in Europe (DATE), the International Conference on Computer Aided Design (ICCAD), the International Symposium on High-Performance Computer Architecture (HPCA), the ACM International Conference on Supercomputing (ICS), the IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC), and the Workshop on Application-Specific Processors (WASP).