Introduction of Local Memory Elements in Instruction Set Extensions

Partha Biswas[†*] partha@cecs.uci.edu
Vinay Choudhary[§] vinchr@cse.iitk.ac.in
Kubilay Atasu[§] atasu@cmpe.boun.edu.tr
Laura Pozzi[§] laura.pozzi@epfl.ch
Paolo Ienne[§] paolo.ienne@epfl.ch
Nikil Dutt[†*] dutt@cecs.uci.edu

[†]Center for Embedded Computer Systems
School of Information and Computer Science
University of California, Irvine, CA, USA

[§]Processor Architecture Laboratory
Swiss Federal Institute of Technology
Lausanne, Switzerland

ABSTRACT

Automatic generation of *Instruction Set Extensions (ISEs)*, to be executed on a custom processing unit or a coprocessor is an important step towards processor customization. A typical goal of a manual designer is to combine a large number of atomic instructions into an ISE satisfying microarchitectural constraints. However, memory operations pose a challenge for previous ISE approaches by limiting the size of the resulting instruction. In this paper, we introduce memory elements into custom units which result in ISEs closer to those sought after by the designers. We consider two kinds of memory elements for mapping to the specialized hardware: small hardware tables and architecturally-visible state registers. We devised a genetic algorithm to specifically exploit opportunities of introducing memory elements during ISE generation. Finally, we demonstrate the effectiveness of our approach by a detailed study of the variation in performance, area and energy in the presence of the generated ISEs, on a number of MediaBench, EEMBC and cryptographic applications. With the introduction of memory, the average speedup varied from 2.7X to 5X depending on the architectural configuration with a nominal area overhead. Moreover, we obtained an average energy reduction of 26% with respect to a 32-KB cache.

Categories and Subject Descriptors: C.1.3 [Processor Architectures]: Other Architecture Styles

General Terms: Algorithm, Design, Performance.

Keywords: Customizable processors, ASIPs, Instruction Set Extensions, Ad-hoc Functional Units, Coprocessors, Genetic Algorithm.

[*]This work was partially supported by NSF grants CCR-0203813 and CCR-0205712

DAC 2004, June 7–11, 2004, San Diego, California, USA.

1. INTRODUCTION

In the era of high-performance, low-power and cost-effective systems, automatic customization to the application requirement is the key to achieve fast turn-around time in a competitive market. *Ad-hoc Functional Units (AFUs)*, coupled with standard microprocessors, represent the hardware realizations of the critical sections of an application. In order to utilize these specialized AFUs, the instruction set of the base architecture is augmented with the corresponding application-specific *Instruction Set Extensions (ISEs).*

State of the art in automatic ISE generation typically is limited to complex instructions that exclude memory accesses [1, 2, 3]. Including generic memory accesses in ISEs creates a twofold problem. First, the resulting instruction has a non-deterministic latency, an undesirable characteristic especially in compile-time scheduled machines. Next, the architectural design of AFUs becomes significantly complicated requiring necessary synchronisation to memory. Hence, AFUs are typically designed without a dedicated access to memory, adhering to the register-to-register RISC philosophy. However, a method that can include state in the AFUs has the benefit of increasing the scope of ISE; and reducing memory and register-file accesses from the main processor.

The aim of this paper is to present a method for including architectural state in the AFUs, without adding a dedicated access to memory from the AFU itself. Our observations show that in the kernel of many embedded applications some memory accesses correspond to fixed tables such as logarithmic or cryptographic tables. Embedding these inside AFUs can lead to several advantages — (1) Speedup: memory access to a small memory, now within the AFU, is faster. (2) Energy reduction: access to smaller and tagless memories saves considerable energy. (3) Lowered cache pollution: the tables are accessed solely within the AFUs reducing the chances of evicting the highly-accessed cache lines. We also observed that some values are accumulated within a loop, opening possibilities for inserting internal registers within the AFUs and thereby lowering the input/output needs of the resulting instruction. If the problem of including memory state such as in the cases described above can be correctly formulated and solved, the scope of ISE can be tangibly increased leading to higher performance.

The rest of the paper is organized as follows. In Section 2, we discuss related research work. Section 3 presents our mo-

tivations, while Section 4 defines the problem at hand. We discuss our approach in Section 5. Section 6 presents a detailed analysis of the experimental results. Finally, Section 7 concludes the paper.

2. RELATED WORK

The problem of ISE generation for application specific processors has been studied for almost a decade. Loosely stated, the problem is to identify legal subgraphs/clusters of a set of *Data Flow Graphs (DFGs)* which are potential sources of speedup. Recently, efforts have been directed to automate the identification process by proposing heuristics to practically solve this problem of exponential complexity under microarchitectural constraints.

When the goal is speedup coupled with dynamic reuse, as in [3, 5, 6], the resulting subgraphs in most cases are generally small. On a similar note, the subgraphs generated starting from an output node by adding predecessor nodes resulting in a single-output ISEs [7] may result in limited speedup. One of our goals is to generate large clusters with higher potential for speedup.

All the related research work, such as [1, 2, 3], consider the occurrence of memory instructions implicitly as a pruning criterion in the clustering process, and thus exclude memory elements. Our method and results show that some kind of memory instructions, which act as stumbling blocks in the growth of clusters, can be included in clustering.

Extensible processors have been primarily studied in the light of performance in the context of both ASIPs [10] and reconfigurable computing [4, 11, 12]. Instead of giving the coprocessor access to the main memory, as in [4], we propose to have a special memory inside the AFU. Energy consumption along with performance was addressed for the first time in a recent work [3]. In this paper, we show the merit of having memory elements in AFUs in terms of performance and energy reduction. In addition, we analyze the feasibility of implementation in terms of area overhead.

The contributions presented in this paper bear some resemblance with a recent work on scratchpads [13]. We go beyond the scratchpad approach by bringing special portions of memories closer to the core — directly inside the (application-specific) Functional Unit that is going to use them. Our results on energy reduction closely match those obtained by mapping the frequently accessed data to an Application-Specific Memory instead of a cache [14].

3. MOTIVATING EXAMPLES

Consider the **Advanced Encryption Standard (AES)** benchmark, a Rijndael block cipher with a block/key size of 16 bytes. The stages involved in AES encryption/decryption of a 16-byte 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-byte block is multiplied with a constant matrix under some special rules involving multiply and XOR operations. (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 9 rounds presenting itself as a hot spot for optimizations. The basic stages of a round

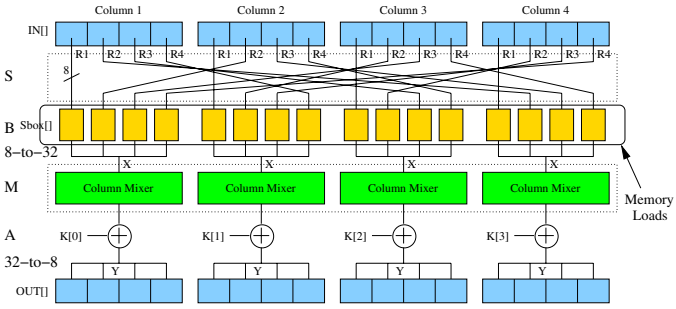

Figure 1: AES main kernel

as implemented in the benchmark are captured in Figure 1. The input 16-byte block (conceptualized as a 4X4 matrix with 1-byte entries) is realized as 4 blocks of *unsigned integer* (4 bytes 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 are 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 our work is to steer the ISE design space exploration for generating results close to those achieved manually by an architect.

Consider a second example: the **ADPCM-decode** benchmark from the *MediaBench* suites, which also exhibits reads from two monodimensional arrays (viz., *indexTable* and *stepsizeTable* of sizes 16 and 89 respectively) and therefore can benefit from the introduction of hardware tables.

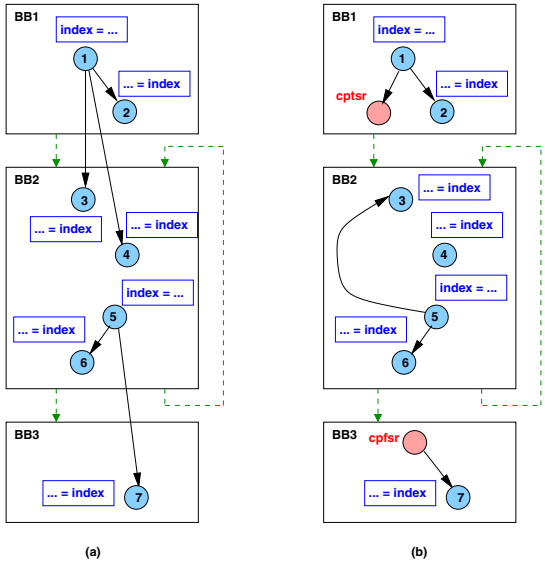

Figure 2: ADPCM-decode example: (a) Original DFG (b) State Register introduced into DFG

The *ADPCM-decode* benchmark presents another interesting scenario as depicted in Figure 2. The dotted edges show the control flow of the application. Figure 2 (a) shows the different points in the DFG where a variable *index* is

read (in nodes 2, 3, 4, 6 and 7) and written (in nodes 1 and 5). An ISE generation algorithm applied to basic block BB2 will naturally consider edges 1-3/1-4 and 6-7 as one of the external inputs and outputs respectively. However, we propose to introduce **architecturally-visible state registers** into AFUs and the following two instructions into the core instruction set:

- ❒ *Copy To State Register (cptsr)*:
  cptsr <*state_reg*> <*core_reg*> copies the core register content <core_reg> into the state register <state_reg>.
- ❒ *Copy From State Register (cpfsr)*:
  cpfsr <*core_reg*> <*state_reg*> copies back the content of state register <state_reg> into the core register <core_reg>.

The uses of *cptsr* and *cpfsr* are illustrated in Figure 2 (b) assuming that the data-flow shown in BB2 maps into an AFU. In basic block BB1, the value of index is copied into a state register using *cptsr*. The state register inside the AFU gets updated in successive iterations of the loop executing in the AFU. At the end of the loop, the *cpfsr* instruction is used to retrieve the value back into a core register as shown in BB3. With the introduction of state registers in AFUs, the selected ISE needs one less input and one less output corresponding to each state register. Our goal is to exploit the presence of both kinds of memory elements: hardware tables and architecturally visible state registers through effective ISE generation for higher performance.

4. PROBLEM DEFINITION

We call $G(V, E)$ the *Directed Acyclic Graph (DAG)* representing the dataflow within each basic block; the nodes V represent primitive operations and the edges E represent data dependencies. Each graph G is associated to a graph $G^+\left(V \cup V^+, E \cup E^+\right)$ which contains additional nodes V^+ and edges E^+. The additional nodes V^+ represent input and output variables of the basic block. The additional edges E^+ connect nodes V^+ to V, and nodes V to V^+(Figure 3(a)).

We now define a cyclic graph $G^{'}\left(V, E \cup E^{'}\right)$ (Figure 3(b)). Graph $G^{'}$ contains the same nodes as graph G, and additional edges $E^{'}$. The additional edges $E^{'}$ represent (loop-carried) data dependencies across basic blocks between nodes V. An edge in $E^{'}$ exists from a node $u \in V$ to another node $v \in V$ if all the following conditions are true:

- ➤ graph G represents a basic block within a loop,
- ➤ v is connected to some $v^+ \in V^+$ in graph G^+,
- ➤ u is connected to some $u^+ \in V^+$ in graph G^+,
- ➤ nodes v^+ and u^+ represent the same variable as input and output to G respectively, and
- ➤ this variable is neither read nor written in other basic blocks within the same loop body.

Similar to graph G, each graph $G^{'}$ is associated to a graph $G^{'+}\left(V \cup V^{'+}, E \cup E^{'} \cup E^{'+}\right)$ which contains additional nodes $V^{'+}$ and edges $E^{'+}$. The additional nodes $V^{'+}$ represent input and output variables of the basic block (V^+) which have not undergone the conditions above. The additional edges $E^{'+}$ connect nodes $V^{'+}$ to V, and nodes V to $V^{'+}$. $V_{\text{load}} \subset V$ and $V_{\text{store}} \subset V$ in $G^{'}$ (and G) represent memory-load and memory-store instructions respectively. For a node $v_l \in V_{\text{load}}$, memloc$[v_l]$ represents locations that v_l can read from. Similarly, a node $v_s \in V_{\text{store}}$ can write into locations, memloc$[v_s]$. Let V_{stores} be the union of all V_{store} in the application. A memory-load instruction, $v_l \in V_{\text{load}}$ is a **read-only** access to memory if $\forall v_s \in V_{\text{stores}}$, memloc$[v_l] \bigcap$ memloc$[v_s] = \phi$.

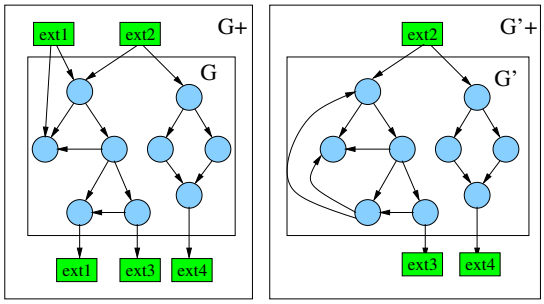

Figure 3: An example of the graphs described: (a)Graphs G and G^+(b)Graphs $G^{'}$and $G^{'+}$. Note that the two nodes of V^+corresponding to variable 'ext1' do not belong anymore to $G^{'+}$, while new edges in $G^{'+}$ have appeared.

A *cut* S is a subgraph of $G^{'}(S \subseteq G^{'})$ representing a potential ISE. There are $2^{|V|}$ possible *cuts*, where $|V|$ is the number of nodes in $G^{'}$. A function $\mathrm{M}(S)$ measures the merit of a cut S as an estimation of the speedup achievable by implementing S as a special instruction.

We call $\mathrm{IN}(S)$ the number of predecessor nodes of those edges which enter the cut S from the rest of the graph $G^{'+}$. They represent the number of input values used by the operations in S. Similarly, $\mathrm{OUT}(S)$ is the number of predecessor nodes in S of edges exiting the cut S. They represent the number of values produced by S and used by other operations, either in $G^{'}$ or in other basic blocks. We call the cut S *convex* if there exists no path containing forward edges from a node $u \in S$ to another node $v \in S$ which involves a node $w \notin S$. If a cut is not convex, the input operands of the ISE respresented by the cut will not be available at the time of issue. We now define the problem as follows:

PROBLEM 1. *Given graphs $G^{'}$ and $G^{'+}$, find the cut S in $G^{'}$ which maximises $\mathrm{M}(S)$ under the following constraints:*

1. $\mathrm{IN}(S) \leq N_{\text{in}}$,
2. $\mathrm{OUT}(S) \leq N_{\text{out}}$,
3. *S is convex,*
4. *A node $v \in V_{\text{load}}$ can be part of S if v is a read-only access to memory.*

The user-defined values N_{in} and N_{out} refer to the number of register-file read and write ports respectively, which can be used by the cut S in the form of an ISE. The merit function $\mathrm{M}(S)$ should be a simple function estimating the speedup achievable by implementing S as an ISE. The above formulation of the problem is a modified version of the one presented in [1]. The modifications are aimed at including architectural state in the AFUs in two ways: (1) The additional edges $E^{'}$ of $G^{'}$ represent **back-edges** of a loop. The variables associated with these edges can be kept inside the AFU and do not need to be read from or written back into the register file at every iteration of the loop. Therefore, if

a cut S contains one or more of these edges, one or more state registers can be inserted in the AFU in order to hold the corresponding values. (2) Read-only memories can now be included inside an AFU using constraint 4. The read-only values of such memories can be loaded into the AFU either at fabrication time or at load time, depending on the technology used. Note that the envisioned architecture still does not require the AFU to address main memory directly.

5. ISE GENERATION ALGORITHM

The optimal algorithms for ISE generation [1] no longer applies to graph $G^{'}$ because in the presence of back-edges, the node numbers are not monotonic in the topological ordering. We use a genetic algorithm (originally proposed by Holland [8]) for ISE exploration and profitably introducing small read-only memories and architecturally-visible state registers in AFUs. The genetic algorithm has the advantage of searching multiple areas in solution space in parallel to find a globally-optimal solution. The genetic algorithm (depicted in Figure 4) starts with a population of possible solutions (subgraphs in our case) and evolves into a set of solutions that maximize a fitness function. A selection mechanism based on the fitness values decides which individuals will survive to the next generation. Genetic recombination operators are applied to the surviving individuals to produce offsprings for the next generation. Repeated selections and recombinations between generations result in a continuous evolution of the population till a termination criterion is satisfied.

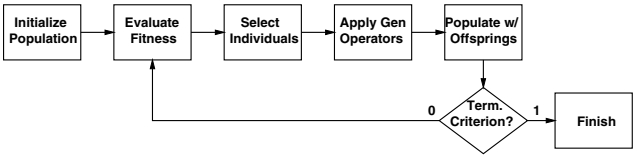

Figure 4: Genetic Algorithm

We adopted a natural encoding of solutions as bitstrings. A bitstring is of length $|V|$, with each node $v \in V$ assigned a bit position based on its fixed topological order in $G^{'}$. The *1* or *0* value of a bit indicates the presence or absence of the corresponding node in the subgraph respectively. The initial population is built using MaxMISO algorithm [7] which extracts all the disjoint maximal-input single-output subgraphs from $G^{'}$. Our strategy is similar to that of messy genetic algorithms [9]: we explicitly search for low-order high-fitness solutions as building blocks in the initial stages and then combine them in the later stages using recombination.

A population here consists of subgraphs of $G^{'}$, (1) obeying convexity constraints and (2) having any node $v \in V_{\text{load}}$ as read-only. The fitness function measuring the fitness of an individual (i.e., cut S) in the population is defined as follows:

$$F(S) = F_I(S) \cdot F_P(S) \cdot F_M(S),$$

$$F_I(S) = (1 - \alpha \cdot ((IN(S) - N_{in}) + (OUT(S) - N_{out}))),$$

$$F_P(S) = min(M(S), M_{best}),$$

where M_{best} = the best merit value among the existing feasible solutions,

$$F_M(S) = \beta \cdot \prod_{v \in V_b(S)} \gamma^{EQ(v,S,G')} \cdot \frac{|out\_adj[v][S]|}{|out\_adj[v][G']|}, \text{where}$$

$$EQ(v, S, G') = \begin{cases} 1, \text{if } |out\_adj[v][S]| \text{ equals } |out\_adj[v][G']|, \\ 0, \text{otherwise} \end{cases}$$

and $V_b(S)$ is the set of all vertices in S that are sources of back-edges in S, $out\_adj[v][S]$ are the nodes in S that are in the adjacency list of v w.r.t. the outgoing edges and similarly $out\_adj[v][G']$ are the corresponding nodes in $G^{'}$.

The multipliers F_I, F_P and F_M respectively capture I/O constraints, performance metric and architecturally visible state considerations. If $IN(S) > N_{in}$ or $OUT(S) > N_{out}$, the I/O constraints are violated and so we penalize the subgraph heavily so that the evolution process will eventually discard it. The value of penalty parameter α has been empirically determined to be 0.05. The factor F_P ensures that the best merit value is always owned by a feasible solution and associates scaled merit values with infeasible solutions. The fitness contribution from F_M rewards inclusion of backedges in S with the help of positive parameters β (=5.00) and γ (=6.00). The parameter β gives a positive merit to the presence of some back-edges and γ gives more weight to the inclusion of all the back-edges (i.e., when $|out\_adj[v][G']$ equals $|out\_adj[v][S])$. This scheme ensures that the algorithm eventually converges into a feasible solution (if any) including all the back-edges. The solutions including read-only loads in S are implicitly favored because of the corresponding increase in $M(S)$.

The fitness values for all individuals in the population are calculated and a roulette wheel selection technique [8] is employed to select fit individuals for recombination. The genetic operators used for recombination are mutation and crossover. Mutation causes random alterations of the bits in the bitstring representation at a rate called mutation rate. The crossover operator chooses random pairs of selected subgraphs (or bitstrings) and exchanges nodes (or bits) with the aim of producing better subgraphs. The probability with which the crossover operation is performed is called crossover rate. The offsprings produced by mutation and crossover are tested for convexity constraints and violating individuals are discarded. In the acceptable set of individuals, any memory-load that is not read-only is removed (without violating the convexity constraints) resulting in a valid population for the next generation. The termination criterion has been chosen as obtaining the best fitness value for last 20 generations. Our experiments produced best results with the following parameters: population size = 400, crossover rate = 0.95, mutation rate = 0.001.

6. EXPERIMENTS

We chose benchmarks mostly in the telecommunication domain from EEMBC (*autcor*, *viterbi* and *bezier*) and MediaBench (*adpcm_coder* and *adpcm_decoder*) suites. In addition, we chose a cryptographic application viz. *AES*. We

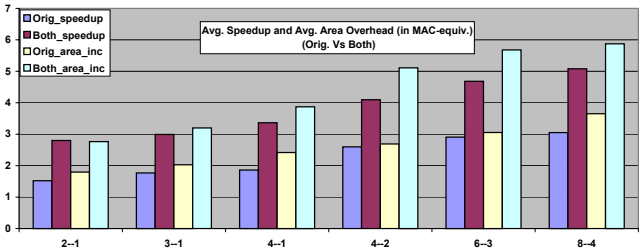

Figure 5: Comparison of Average Speedup and Area Overhead

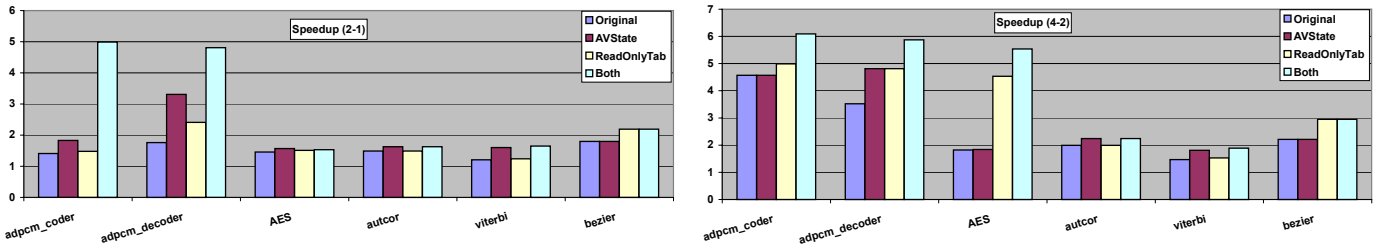

Figure 6: Comparison of Speedup (Left: $N_{in} = 2, N_{out} = 1$; Right :$N_{in} = 4, N_{out} = 2$)

integrated our algorithm in the MachSUIF framework [15]. The read-only tables are generally implemented as global arrays in the applications; any load or store into a global array is represented through a special instruction in SUIF [16]. We used this feature to find the read-only accesses to the tables. We manually verified that there were no malicious accesses to the global arrays using pointers in the given applications. This could have been done automatically using any well-known technique of pointer disambiguation [17].

The performance metric for a cut S, $M(S)$ estimates the speedup achievable by executing the cut as an ISE in an AFU, which is expressed as: $M(S) = \lambda_{sw}(S)/\lambda_{hw}(S)$. The software latency $\lambda_{sw}(S)$ is estimated as the execution time on a single-issue processor. This is calculated as the accumulated latencies of the nodes in S. The hardware latency $\lambda_{hw}(S)$ is the ceiling of the sum of hardware latencies of instructions in the critical path of S. The hardware latencies for each instruction is obtained by synthesizing the constituent arithmetic and logic operators on a common $0.18\mu m$ CMOS technology and then normalized to the delay of a 32-bit *multiply-accumulate (MAC)*. The area occupied by an AFU is measured by adding the area contributions from the hardware realizations of the constituent instructions and normalizing to the area of a MAC. We will refer to a unit area as a **MAC-equivalent**.

The base configuration for our experiments is an architecture having a core processor but no AFU. The AFUs added to the architecture are of the following kinds — (1) *Original*: No memory elements inside, (2) *AVState*: Having state registers only, (3) *ReadOnlyMem*: Having hardware tables only and (4) *Both*: Having both kinds of memory elements. We compare architectures having different kinds of AFUs with different constraints on the number of register file input-output ($N_{in} - N_{out}$) ports. Figure 6 plots the speedups obtained over the base configuration for 2-1 and 4-2 as I/O constraints. In all the plots, the speedup meteric is evaluated for the whole kernel. Unless otherwise stated, the maximum number of AFUs allowed is chosen as 8 and I/O constraints as 4-2. The benchmark, *autcor* exhibits an instance of *AVState* while *bezier* and *AES* present cases of *ReadOnlyTab*; all other benchmarks incorporate both kinds of memory elements. It is interesting to note that in *adpcm_coder* and *adpcm_decoder*, the multiplied effect of *AVState* and *ReadOnlyTab* results in a speedup of the order of 5X for 2-1 and 6X for 4-2 respectively.

We show a plot of speedup and area overhead averaged over all the applications for different I/O constraints in Figure 5. The metrics are shown for *Original* and *Both* configurations. As expected, the worst case area overhead ($\sim 5.5 MAC - equivalents$) is obtained for an 8-4 AFU with a maximum average speedup of 5X. Note that in general the area overhead is very reasonable: even clusters including hundreds of nodes, as in the case of AES, result in AFUs with a limited area overhead ($\sim 2 MAC - equivalents$).

Table 1: Percentage energy saving using hardware tables instead of data caches

BMs	N_A	N_C	% Energy Saving
adpcm_coder	295188	443892	31.75
adpcm_decoder	295188	591412	26.47
AES	16408	21859	34.09
viterbi	2160000	72183000	2.38
bezier	48004004	61208005	34.94

Our technique is also effective in reducing the energy consumption because of two 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. We show the energy reduction taking only the first feature into account. We consider a 32-KB direct-mapped data-cache (with 2048 lines of 16-bytes each) as used in typical implementations of ARM for energy efficiency. If the constituent instructions in an ISE are executed in software, we conservatively assume that all memory operations map into the data cache.

After mapping the ISEs to AFUs, let the number of accesses to AFU-resident memory be N_A and the number of loads/stores directed to the cache be N_C. So, the number of accesses to cache when there are no AFUs or for an AFU that does not contain hardware tables 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 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 $0.18\mu m$ technology and found

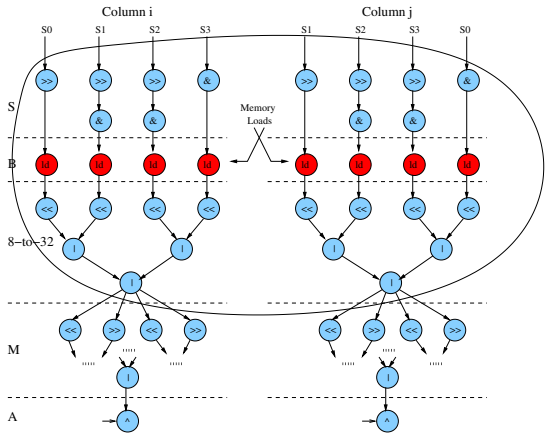

Figure 7: Clustering in AES

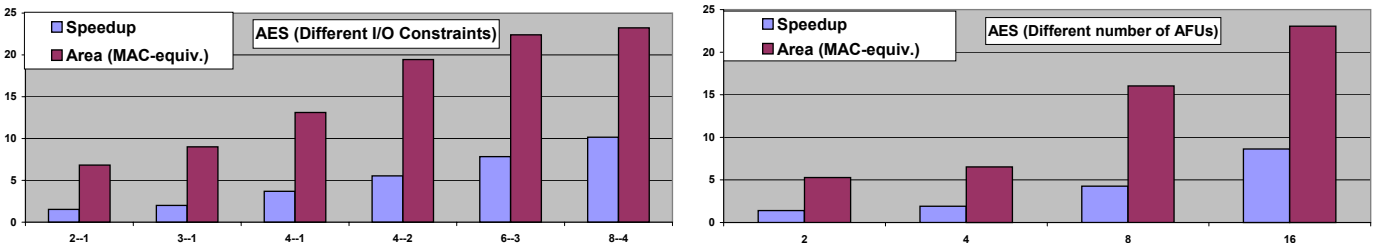

Figure 8: Impact of variation in the I/O constraints and the maximum number of AFUs for AES

the ratio $(E_C - E_A)/E_C$ to be 0.795 for the maximum size (1 KB) of the hardware tables in the chosen applications. Hence, the above expression simplifies into: $\frac{0.795 \cdot N_A}{N_A + N_C}$. We present N_A, N_C and percentage energy saving due to redirection of the data cache accesses to AFU memory in Table 1.

Because the above calculations are done under a number of conservative assumptions, we can safely say that the average energy reduction in cache due to AFU-resident memory is at least 26%. Since the cache is a significant contributor of system energy, using hardware tables in AFUs would result in a perceptible overall system energy reduction.

Among the chosen benchmarks, AES is the one having the largest number of nodes in identified ISEs (of the order of 150). We study ISE generation for AES in greater detail by varying the number of I/O ports and the number of AFUs chosen. We show the effect of increasing the number of ports on speedup and area overhead in Figure 8. Figure 8 also shows the effect of increasing the number of maximum AFUs allowed. It is interesting to note that increasing the number of I/O ports to 8-4 gains higher speedup ($\sim 10X$) than increasing the number of AFUs to 16 with almost equal area overhead in both cases.

In order to study the efficacy of our clustering, we show a part of the data-flow graph in Figure 7, essentially covering two of the columns of Figure 1. One of the cuts (or ISEs) selected by our algorithm under 4-2 constraints is shown in the figure. If the memory loads (as shown in the figures) are not considered in AFUs, the largest cut generated has around 60 nodes. However, in the presence of AFU-resident memory, the number of atomic instructions included in this cut rises to the order of 150. Since these instructions mostly represent very simple bitwise operations, the resulting ISE (critical-path) latency has been found to be as little as the delay of 2 MAC operations. In general, we envision long-latency ISEs to be run as multi-cycle operations causing processor stalls but not affecting the processor cycle time. With the worst-case implicit register addressing, it is possible to fit the generated ISEs in the instruction encoding space.

7. CONCLUSIONS

Three main contributions were presented in this paper. Firstly, we introduced local memory (or state) into Ad-hoc Functional Units (AFUs) in the form of hardware tables and architecturally-visible state registers. Secondly, we guided a genetic algorithm to exploit the presence of memory elements in AFUs. Finally, we studied in detail the impact on performance, area and energy consumption, clearly demonstrating the advantages of including state into AFUs. For the cryptographic application AES, we obtained a speedup of the order of 10X in the best case with a maximum area overhead of only 2 MAC-equivalents for the largest cut. We also indirectly showed that speedup and energy are not conflicting goals in ISE generation by obtaining a maximum average speedup of 5X as well as 26% average reduction in cache energy.

We believe that automation of ISE generation is of utmost importance, and our contributions further bridge the gap between automatic solutions and those manually designed by expert engineers. In particular, the problem formulation and the proposed algorithm resulted in very large clusters in the order of 150 nodes and at the same time having limited area and delay overhead — a highly desirable and novel result in the state of the art.

8. REFERENCES

[1] K. Atasu, L. Pozzi and P. Ienne. Automatic Application-Specific Instruction-Set Extensions under Microarchitectural Constraints. In *Proc. of DAC*, 2003.

[2] N. Clark, H. Zhong and S. Mahlke. Processor Acceleration through Automated Instruction Set Customization. In *Proc. of MICRO*, 2003.

[3] F. Sun, S. Ravi, A. Raghunathan and N.K. Jha. Synthesis of Custom Processors based on Extensible Platforms. In *Proc. of ICCAD*, 2002.

[4] T. Callahan and J. Wawrzynek. Instruction-Level Parallelism for Reconfigurable Computing. In *Proc. of FPL*, 1998.

[5] M. Arnold and H. Corporaal. Designing Domain-specific Processors. In *Proc. of CODES*, 2001.

[6] H. Choi, I.C. Park, S.H. Hwang and C.M. Kyung. Synthesis of Application Specific Instructions for Embedded DSP Software. *IEEE TC*, 1999.

[7] C. Alippi et. Al. A DAG based Design Approach for Reconfigurable VLIW Processors. In *Proc. of DATE*, 1999.

[8] J.H. Holland. Adaptation in Natural and Artificial Systems. *U. Mich. Press*, 1975.

[9] D.E. Goldberg, B. Korb and K. Deb. Messy Genetic Algorithm: Motivation, Analysis and First Results. *Complex Systems*, 1989.

[10] A. Wang, E. Killian, D. Maydan and C. Rowen. Hardware/Software Instruction Set Configurability for System-on-chip Processors. In *Proc. of DAC*, 2001.

[11] R. Razdan and M.D. Smith. A High-performance Microarchitecture with Hardware-programmable Functional Units. In *Proc. of MICRO*, 1994.

[12] Z.A. Ye, A. Moshovos, S. Hauck and P. Banerjee. CHIMAERA: A High-performance Architecture with a Tightly-coupled Reconfigurable Functional Unit. In *Proc. of ISCA*, 2000.

[13] S. Steinke, L. Wehmeyer, B.S. Lee and P. Marwedel. Assigning Program and Data Objects to Scratchpad for Energy Reduction. In *Proc. of DATE* 2002.

[14] L. Benini, A. Macii, E. Macii, M. Poncino. Synthesis of Application-Specific Memory for Power Optimization in Embedded Systems. In *Proc. of DAC*, 2000.

[15] Machine SUIF. http://www.eecs.harvard.edu/hube/software/software.html.

[16] R.P. Wilson et. al. SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers. *SIGPLAN Notices*, 1994.

[17] R.P. Wilson and M. Lam. Efficient Context-sensitive Pointer Analysis for C programs. *SIGPLAN*, 1995.