Automating Processor Customisation: Optimised Memory Access and Resource Sharing

Robert Dimond, Oskar Mencer and Wayne Luk
Department of Computing, Imperial College,
180 Queens Gate, London. SW7 2RH
{*rgd00,oskar,wl*}*@doc.ic.ac.uk*

Abstract

We propose a novel methodology to generate Application Specific Instruction Processors (ASIPs) including custom instructions. Our implementation balances performance and area requirements by making custom instructions reusable across similar pieces of code. In addition to arithmetic and logic operations, table look-ups within custom instructions reduce costly accesses to global memory. We present synthesis and cycle-accurate simulation results for six embedded benchmarks running on customised processors. Reusable custom instructions achieve an average 319% speedup with only 5% additional area. The maximum speedup of 501% for the Advanced Encryption Standard (AES) requires only 3.6% additional area.

1. Introduction

State-of-the-art embedded processing requires optimisation and balancing of requirements such as performance, power consumption and area cost. Typically, an acceptable execution speed must be achieved at the lowest possible power consumption and within a specified area budget. Conventionally, the designer either selects an off-the-shelf instruction processor, or else develops a customised hardware accelerator for the task.

Application Specific Instruction Processors (ASIPs) provide an alternative solution by allowing the designer to customise a base processor to a specific application. An ASIP can be customised by changing the instruction set to directly implement frequently performed operations. Such 'custom instructions' provide the speed and efficiency of hardware for selected program segments, while supporting the rapid software design flow of a fixed processor. ASIP vendors provide tools that automate the synthesis of a processor, for example XTensa (Tensilica) and LISAtek (CoWare). However, the designer still selects custom instructions. High level processor description languages [5] facilitate rapid integration, but do not help with selecting instructions.

ASIP customisation is automated by searching [6] for an optimal partition of an application between hardware (as custom instructions) and conventional software (base instructions). However, regarding custom instruction selection as a partitioning problem has a disadvantage: partitioning does not consider reuse of instructions until after they are selected. Techniques based on regularity extraction [2, 10], similar to that used for hardware resource sharing seek to reuse instructions but are unable to exploit algebraic relationships for reuse over non-identical code.

This paper introduces a novel technique, Similar Sub-Instructions (SSI) for automatic customisation of an extensible processor. The SSI technique identifies structurally similar computations that can be implemented using the same hardware custom instruction. We claim three advantages over previous regularity extraction techniques. 1) We exploit commutativity in expressions so that instructions can be reused across expressions that are non-identical. 2) We permit resource sharing for non-identical operators that can be implemented using the same hardware. 3) Computations may include reads from look-up tables that the compiler identifies as read-only and can allocate to dedicated custom instruction memories.

2. Related Work

We classify existing techniques for automatic processor customisation as either partitioning or clustering. Partitioning methods [6, 8] divide computation between custom and base instructions, then optionally perform tests to identify identical custom instructions that can be reused. Clustering approaches [2] are driven by regularity extraction and build instructions to implement frequently occurring segments. Sun et al. propose a method that combines elements of both ideas [10].

Our system is similar to clustering approaches [2] although we operate on incidence matrices rather than dags to exploit commutativity and reuse across non-identical computations. Unlike existing clustering approaches, we also al-

3-9810801-0-6/DATE06

$$J = A \oplus (B >> 6)$$
$$K = C \oplus D \oplus ((E >> 5) + 1)$$
$$L = (F >> 6)$$

(a) System of assignments.

$\oplus$	A	(B>>6)	C	D	((E>>5)+1)	(F>>6)
J	1	1	0	0	0	0
K	0	0	1	1	1	0
L	0	0	0	0	0	1

(b) Incidence matrix and chains representation.

Figure 1. Incidence matrix with input chains example. The $\oplus$ symbol denotes any binary commutative operator.

low access to read-only data within custom instructions by incorporating dedicated memory elements, a technique first proposed in [8].

Exploiting datapath reuse in an ASIP is first discussed in Fauth et al. [4]. Their Hardware Modelling Cell (HMC) serves a similar purpose to our generic operator (*genop*) construct that models a hardware block. However, their work assumes that instructions are manually pre-designed and only considers reuse at the synthesis stage. Our technique considers reuse in the context of automatic design.

We use extended versions of techniques from computer algebra. Our structural recognition technique is inspired by Van Hulzen [9], which briefly suggests topological sorting of incidence matrix columns to recognise structure. We also extend Breuer's grow factor algorithm [1], originally devised for common sub-expression elimination. Our version uses a modified 'figure of merit' heuristic, adapted to the problem of finding custom instructions.

3. Similar Sub-Instructions

Our novel technique, Similar Sub-Instructions (SSI) automatically selects custom instructions within a high level language compiler. SSI operates by finding repeated sub-expressions that can be computed using the same hardware. The technique has two outputs: software including custom instructions at the appropriate locations and datapaths to implement these custom instructions.

The key principle of our approach is to fit the expressions in a program region to a model composed of two structures: Incidence matrices and Chains. Incidence matrices represent expressions of binary, commutative operators. We use incidence matrices to identify common structures of expressions in the input code. Chains are sequences of unary operators that represent inputs to incidence matrices. A unary operator is a table lookup, negation or a constant input binary operator such as an add or shift. We use chains to find opportunities to reuse hardware blocks. Any code that does not fit the incidence matrix and chains model (e.g. function calls) is assigned to a temporary variable and left unchanged. There are four major steps involved that we detail in the remainder of this section.

Algorithm 1: Structural recognition algorithm

```
   Data: Incidence matrix I and input unary operators Itop
   Result: New incidence matrix O and input unary operators Otop
 1 begin
 2     O ← Empty matrix with same number of rows as I
 3     foreach column I_i do
 4         placed ← false
 5         foreach column O_j do
 6             if ¬∃k : I_ik = 1 ∧ O_jk = 1 then
 7                 if topchain ← Merge(Itop_i, Otop_j) then
 8                     placed ← true
 9                     Otop_j ← topchain
10                     Vector sum of columns
11                     Oj ← O_j + I_i
12                 end
13             end
14         end
15         if ¬placed then
16             Add as new column in output matrix
17             O_{j+1} ← I_i
18         end
19     end
20 end
```

3.1. Step A: Structural recognition

The input to the structural recognition algorithm is a set of incidence matrices. A separate incidence matrix is required for each binary commutative operator in the program region being examined. An incidence matrix relates expression outputs (the row headings) to expression inputs (the column headings), where a '1' entry implies that the input is contained in the sum for the corresponding output. For example, the set of expressions in Figure 1a can be represented by the incidence matrix of Figure 1b. The advantage of this representation is that the commutativity of the operator is explicit. In contrast, a conventional directed acyclic graph (DAG) or expression tree obscures this information. Exploiting commutative properties of arithmetic increases the possibility of finding opportunities for instruction reuse.

In our system, the inputs to an incidence matrix consist of *chains* which are unary expression sequences. A chain can contain any sequence of unary operations, such as manipulation by a constant, table lookup or read of a temporary variable. A chain contains at least a read of some input variable which might be external to the program region, or the output of an incidence matrix. The chain generalisation allows similar structures to be found in expressions that contain unary and non-commutative operators and to integrate table-lookups. We find common structures by merging columns of an incidence matrix subject to the following restrictions. Firstly, the chains at the input of each column (represented by *topchains* in Algorithm 1) must be *sim-*

⊕	A	(B>>6)	C	D	((E>>5)+1)	(F>>6)
J	1	1	0	0	0	0
K	0	0	1	1	1	0
L	0	0	0	0	0	1

(a) Initial matrix I

⊕	A
J	1
K	0
L	0

(b) O Step 1

⊕	A	(B>>6)
J	1	1
K	0	0
L	0	0

(c) O Step 2

⊕	A‖C	(B>>6)
J	1	1
K	1	0
L	0	0

(d) O Step 3

⊕	A‖C	(B>>6)	D
J	1	1	0
K	1	0	1
L	0	0	0

(e) O Step 4

⊕	A‖C	(B>>6)	D	((E>>5)+1)
J	1	1	0	0
K	1	0	1	1
L	0	0	0	0

(f) O Step 5

⊕	A‖C	((B‖F)>>6)	D	((E>>5)+1)
J	1	1	0	0
K	1	0	1	1
L	0	1	0	0

(g) O Step 6

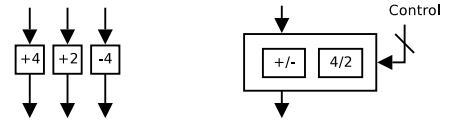
Figure 2. Walk-through of Algorithm 1 starting from Figure 1b. Each step moves a single column from the input matrix I to the output matrix O. Column merging occurs at steps 3 (d) and 6 (g). The symbol ‖ indicates a hardware block with multiple modes of operation.

ilar and thus capable of being merged. Secondly, merging two columns must not create overlap between non-zero entries. This means that columns can not be merged if they are both inputs to the same row. The first test is a target specific check, implemented in a *Merge* function, that the two chains can be implemented using the same hardware. The second test is to prevent hardware reuse within the same instruction, to enable independent parts of that instruction to be computed in parallel.

Pseudocode for structural recognition is shown in Algorithm 1. Tests 1 and 2 above are implemented at lines 7 and 6 respectively. The function call *Merge* at line 7 attempts to merge two chains and, if successful, returns the merged result. The merged result is represented by a chain of *genops* that we define as:

Genop — A *generic operator*, or model of a hardware block that can implement multiple basic operations.

For example, Figure 3a shows three chains each containing a single operation. Figure 3b shows a possible merge result: the generated chain contains a single genop that models a



(a) Input chains. (b) Output chain after two merge operations.

Figure 3. Example *Merge* function (Algorithm 1) on three sequences of unary operators. The result models an abstract hardware adder/subtracter. The control input selects an original operator sequences.

hardware block capable of implementing all three merged chains. The *Merge* function implements a target specific set of rules that control the extent of functionality that can be merged into a single genop. For the above example Figure 3, a rule for a hardware adder block will specify that a constant add and constant subtract can be merged. Other targets might impose different restrictions to allow more or less general hardware data-paths to be generated.

Figure 2 shows an example walk through of the algorithm starting from the example of Figure 1. In this case, two merges of columns take place at steps 3 and 6, giving a four column wide result.

3.2. Step B: Breuer grow factoring

After structural recognition, the next step is to select which columns of the matrix to implement within a custom instruction. The input to this stage is the incidence matrix from the structural recognition stage, the output is boolean for each column of the matrix to indicate whether the column is calculated either inside or outside of the instruction. We refer to this process as *factoring* where a custom instruction is used to implement the sum (using the appropriate operator) of the column set selected as the factor.

Selecting a set of columns is a compromise: adding additional terms to the factor means that more work is done within a custom instruction. However, only expressions that contain the entire factor will benefit from the generated instruction. The objective is to find an optimal solution that lies somewhere between these points. To accomplish this we use the heuristic grow factor algorithm, proposed by Breuer [1]. Breuer's grow factor algorithm finds an optimal combination of columns (the factor) to maximise a heuristic *figure of merit*. Breuer uses a greedy algorithm that adds the column with the highest individual merit at each stage which makes it practical to operate on very large matrices.

The heuristic figure of merit favours selection of instructions that (1) contain a large amount of computation and so will give a significant speedup above a base instruction se-

quence, (2) can be reused to compute multiple expressions. We calculate figure of merit using Eq. 1 and Eq. 2, where the *factored rows* term gives the number of matrix rows containing the entire factor including the current column c.

$$FoM(factor) = \sum_{column \in factor} FoM(column) \quad (1)$$

$$FoM(column \quad c) = Weight(chain) \times \text{factored rows} \quad (2)$$

The chain weight is an estimate of the computational load of a chain associated with a matrix column. It is computed as the sum of the weight of all operations in that chain. The weight of each operation is a metric for how worthwhile it is to implement that operation in a custom instruction. A high relative weight is given to operations that translate well into hardware, for example, table lookups where the table can be allocated to dedicated RAM or fixed shifts/rotates.

Algorithm 2 gives pseudocode for our version of the grow factor algorithm. The operation has four iterative steps:

1. The *figure of merit* is calculated individually for each matrix column (Eq. 1) not currently in the factor. Only rows that contain all of the current factor are included (the test for this is on line 10).
2. The column with the highest merit computed in the previous step is added to the current factor (line 14).
3. If the current factor has the highest total figure of merit (Eq. 2), it is recorded as the best factor and the highest count updated (line 21).
4. The process is repeated until the factor becomes larger than the number of inputs (register ports) available to the custom instruction (line 5), or until every column is in the factor.

3.3. Step C: Heuristic instruction selection

The heuristic selection stage gives a simple yes/no response as to whether each custom instruction is worthwhile. In our implementation, this test prevents generating instructions close to, or very similar to those that exist in the basic instruction set. Without this check, the algorithm will find 2 input add, exclusive OR or other basic operations as custom instructions.

We use two basic tests. The first automatically approves instructions with more than a minimum number of inputs. This allows instructions to be selected that are versions of basic instructions with more inputs; a three input AND for example. The second is a *mean input weight* heuristic that is intended to select instructions that perform significant computation relative to the number of inputs (to guard against only selecting wide versions of basic operators).

Algorithm 2: Breuer grow factor algorithm for selecting terms to implement within a custom instruction. FOM is the figure of merit heuristic.

```
   Data: An incidence matrix I and topchains Itop
   Result: A set of column numbers factor
 1 begin
 2     inputs ← 0
 3     maxmerit ← 0
 4     currentfactor ← ∅
 5     while inputs < maximum do
 6         maxcolsum ← 0
 7         foreach columnI_i do
 8             colsum ← 0
 9             foreach elementI_ij do
10                 if ¬∃x ∈ currentfactor : I_xj = 0 then
11                     colsum ← colsum + Weight(Itop_i)
12                 end
13             end
14             if colsum > maxcolsum then
15                 maxcolsum ← colsum
16                 maxcol ← i
17             end
18         end
19         inputs ← inputs + inputs added to factor
20         currentfactor ← currentfactor ∪ maxcol
21         if FOM(currentfactor) > maxmerit then
22             maxmerit ← FOM(currentfactor)
23             factor ← currentfactor
24         end
25     end
26 end
```

Mean input weight is defined as:

$$Weight = \frac{\text{figure of merit}}{\text{width} \times \text{factored width}} \quad (3)$$

The figure of merit is as calculated in Breuer factorisation (Section 3.2, Eq. 2). The width is the number of inputs to the custom instruction. The factored width is the number of rows in the incidence matrix that contain the entire factor, equal to the number of uses of the instruction. The rationale is to benefit instructions that are more than just basic commutative operations with multiple inputs. Instructions that do significant computation in the chains will have a high mean input weight and are thus likely to be selected. We use a fixed threshold parameter, below which an instruction is rejected.

3.4. Step D: Hardware generation

Given the incidence matrix (Section 3.1) and factor set (Section 3.2), we use simple rules to generate the final code. For every matrix row that contains the factor, we insert the custom instruction. If the matrix row contains elements not contained within the factor, we generate code to calculate the respective input chains and then 'sum' (using the appropriate binary operator) with the result generated by the custom instruction. For matrix rows that do not contain the entire factor, we generate conventional code. The process is best illustrated by the example of Figure 4. Here, the three

$\oplus$	A‖C	((B‖F)>>6)	D	((E>>5)+1)
J	1	1	0	0
K	1	0	1	1
L	0	1	0	0
$\sum$	2	4	1	2
$factor$	0	1	0	0

(a) Example factoring reproduced from Figure 2.

$$J = A \oplus custom()$$
$$K = A \oplus D \oplus ((E >> 5) + 1)$$
$$L = custom()$$

(b) Resulting code

Figure 4. Example operation of the hardware generation (Section 3.4) starting from the result of Figure 2

cases are illustrated. Both J and L contain the factor implemented by the custom instruction (Case 1). J differs from L in that it contains an additional term (Case 2). K does not contain the factor, and so has to be implemented without the custom instruction (Case 3). We generate custom instruction hardware to implement the sum of the input chains for each column in the factor.

4. Implementation

To evaluate our technique, we implemented: (1) An optimising C compiler containing the algorithms of Section 3, (2) A 32-bit pipelined processor that can be extended with custom instructions, (3) A cycle-accurate simulator.

(1) Our C compiler is implemented using the CoSy framework and includes a Similar Sub-Instructions phase together with two groups of conventional optimisation techniques. The first, pre-optimisation group helps to expose potential custom instructions. The second post-optimisation group, improves the quality of the code containing custom instructions. Pre-optimisation includes loop unrolling, so that the regularity of looping structures is exposed and detection of read-only arrays (look-up tables), so that table look-ups can be implemented within a custom instruction. Post-optimisation includes scheduling base and custom instructions to minimise pipeline stalls.

(2) Our ASIP processor is generated to support the custom instructions selected by the compiler and the MIPS integer instruction set. To obtain area and clock-rate results, we place and route the ASIP on a Xilinx XC2V2000 FPGA.

(3) Cycle accurate simulation is provided by extending the SimpleScalar [3] framework to support our processor. The simulator includes a memory system model which is configured for a typical embedded MIPS processor with 2KB of instruction cache and a data cache size of 1 or 2KB. The caches are direct mapped with a random replacement

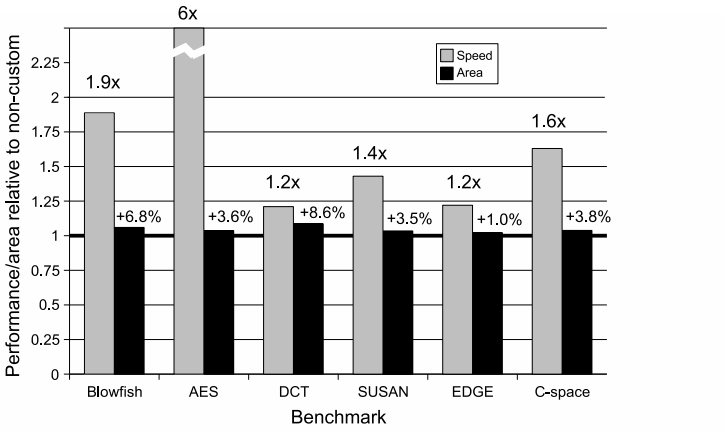


Figure 5. Area and overall performance for processors with and without custom instructions, scaled by maximum clock rate of the custom processor.

policy and 16 byte block size, with all other parameters set to SimpleScalar defaults.

5. Results

To demonstrate our methodology, we use our C compiler, generated ASIP processor and cycle-accurate simulator described in Section 4 to obtain results for six application benchmarks. We select benchmarks from the cryptography (AES, Blowfish) and image processing (Colorspace conversion, Laplace edge detection, SUSAN edge detection, Discrete Cosine Transform) domains. AES, Blowfish and SUSAN are part of the MiBench [7] suite. In all cases the code was not hand optimised with the exception of adding `static` to the definition of certain arrays so that they could be recognised as read-only by the compiler.

Table 1 shows example results for an automatically customised and a non-customised processor for each benchmark. We observe that the custom instruction implementations require significantly fewer execution cycles compared to the same benchmark running on the non-customised processor. This means that we can achieve the same performance at a lower clock rate, or a higher overall performance. In addition, there are fewer instructions executed, so instruction fetch and decode overhead will be reduced. Adding the custom instructions results in a small clock rate penalty, although in every case a large performance improvement is still achieved, from 21% (DCT) up to 501% (AES) with a mean of 319% (Figure 5).

The benefit of implementing table look-ups within custom instructions is shown by the significantly lower performance penalty of halving the cache size for the Blowfish,

Benchmark	Base MIPS CPU				With Custom Instructions							
	1kb Cache		2kb Cache		1kb Cache		2kb Cache		Area cost		Custom Insts.	
	KC	Miss	KC	Miss	KC	Miss	KC	Miss	RAM[b]	Logic[Slices]	# Selected	# Reuse
Blowfish	2190	26439	2030	18207	931	1270	930	1240	1024	1455	2	2,2
Colourspace	32.8	29	32.8	29	18.3	26	18.3	26	32	1415	1	1
DCT	737	390	737	390	594	130	594	130	64	1380	2	65,65
Edge Detect	36.1	1	36.1	1	28	1	28	1	64	1377	1	3
SUSAN	32.1	71	32.1	41	22.3	41	22.3	41	516	1410	1	31
AES	8.73	142	8.06	107	1.3	15	1.3	15	1024	1412	1	64

Table 1. Simulation and synthesis results showing customised and non-customised processors for six benchmarks. Core area comprises block RAM usage (bytes) and XC2V2000 SLICES. Execution cycles/1000 (KC) and number of cache misses for 1kb and 2kb data cache.

SUSAN and AES benchmarks. For example, halving the size of the data cache for the non-customised processor increases execution time of Blowfish by 8% but has negligible effect on the customised processor. Supporting table look-ups within a custom instruction incurs some area penalty (the RAM column in Table 1), although in all cases this is significantly less than an additional 1K of cache when tag/-valid bits and cache logic are considered.

The area efficiency advantage of reusing hardware datapaths (number of static instruction uses reported in the '# uses' column) is shown by the low additional area requirement, a mere 5% average across all benchmarks. For AES, the maximum acceleration of 501% is achieved with only 3.6% additional area and only 1kb of block RAM. There is significant reduction in cache misses, up to 90%, which constitutes a significant power saving if misses result in access to off-chip RAM. We anticipate that the energy saving of a customised processor will be at least as good as the performance improvement.

In addition, Similar Sub-Instructions reduces compile time by an average of 10% across our benchmarks. Incorporating multiple operations within custom instructions reduces execution time of the compiler back-end while adding only a small contribution to the compilation time itself.

6. Conclusion

We present 'Similar Sub-Instructions', a technique for finding reusable custom instructions that incorporate limited memory access in hardware look-up tables. Our custom architectures result in high speedup compared to the fixed processor at a competitive price in additional area. Much of the speedup is attributable to the table look-ups that reduce costly cache misses, indicating that memory system customisation is a fruitful direction for ASIP research. In our future work, we intend to exploit the short solution time of our technique by targeting a processor configurable at run-time (e.g. Stretch).

Acknowledgement We are indebted to ACE Associated Compiler Experts, Celoxica, the EPSRC, Tensilica and Xilinx who provided tools and/or funding to support our work.

References

[1] M. A. Breuer. Generation of optimal code for expressions via factorisation. *Communications of the ACM*, 12(6):333–340, June 1969.

[2] P. Brisk, A. Kaplan, R. Kastner, and M. Sarrafzadeh. Instruction generation and regularity extraction for reconfigurable processors. In *Proc. CASES 2002*, pages 262–269, October 2002.

[3] D. C. Burger and T. M. Austin. The SimpleScalar tool set.

[4] A. Fauth et al. Generation of hardware machine models from instruction set descriptions. In *Proc. IEEE Workshop VLSI Signal Proc., Veldhoven (Netherlands)*, pages 242–250, October 1993.

[5] G. Braun et al. A novel approach for flexible and consistent ADL-driven ASIP design. In *Proc. DAC*, pages 717–722, June 2004.

[6] K. Atasu et al. Automatic application-specific instruction-set extensions under microarchitectural constraints. In *Proc. DAC*, June 2003.

[7] M.R. Guthaus et al. MiBench: A free, comercially representative embedded benchmark suite. In *Proc. IEEE 4th Annual Workshop on Workload Characterisation, Austin, TX.*, December 2001.

[8] P. Biswas et al. Introduction of local memory elements in instruction set extensions. In *Proc. DAC*, pages 729–734, June 2004.

[9] V.V. Goldman and J.A. van Hulzen. Automatic code vectorisation of arithmetic expressions by bottom-up structure recognition. In *Computer Algebra and Parallelism*, pages 119–132. 1989.

[10] F. Sun, S. Ravi, and N.K. Jha. Custom-instruction synthesis for extensible-processor platforms. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 23(2):216–228, February 2004.