# Automatic Topology-Based Identification of Instruction-Set Extensions for Embedded Processors

Laura Pozzi, Miljan Vuletić, and Paolo Ienne
Swiss Federal Institute of Technology Lausanne
Processor Architecture Laboratory
IN-F Ecublens, 1015 Lausanne, Switzerland
{Laura.Pozzi, Miljan.Vuletic, Paolo.Ienne}@epfl.ch

Technical Report CS 01/377
December 2001

## Abstract

*Embedded processors increasingly need specialisation to meet the challenging performance requirements in semi-custom VLSI technologies under tight energy and area budgets. A key issue is being able to determine automatically useful specialisation features from high-level application code (e.g., C or C++). This paper addresses the design automation task for one of the possible forms of specialisation: the inclusion of ad-hoc functional units, possibly implemented on embedded reconfigurable logic. A formal approach to the identification of groups of operations for the ad-hoc unit is presented and a simple lightweight algorithm is shown; it partitions the application data-dependency graph into maximal regions with topological features important at microarchitectural level. Several benchmarks (mostly from* MediaBench *[9]) indicate, under generic microarchitectural assumptions, that even relatively simple operation groups can contribute significantly to speed-up complete applications (up to above $2\times$ in best cases). Several possibilities to enhance the current technique conclude the paper.*

## 1. Introduction

It is not new that the embedded processor market outnumbers the market of general-purpose processors. What is changing in the last few years is that the requirements placed on embedded processors—and especially on those for *Application-Specific Integrated Circuits* (ASIC)—are becoming increasingly similar to those of general-purpose processors in terms of computing power; and yet, the energy consumption and area cost remain at premium for embedded processors.

This need for high performance in ASIC CMOS technologies coupled with aggressive energy and area goals is pushing researchers and designers in industry toward aggressive forms of *processor specialisation* toward a given application-domain. Schematically, this can happen in essentially three mutually nonexclusive ways:

- **Ad-hoc Functional Units.** Inside a traditional architecture (either CISC, RISC, or VLIW) the designer adds special arithmetic/logic functional units which performs complex operations tightly related to a specific application domain. Several recent architectures display an interface for such units and the toolset fully supports them (e.g., [14, 6, 5, 4]).
- **Ad-hoc Coprocessors.** Most established embedded architectures, such as MIPS, ARM, and PowerPC, have the possibility of extending the capabilities of the processor with coarse-grained coprocessors in charge of large computational kernels. Essentially these differ from the previous category because of the management of part of the application control flow in the coprocessor.
- **Processor Customisation.** Most radically, all processor resources (registers, functional units, etc.) can be tuned to the needs of the application domain. Some parametrisable toolset and customisable architectures are emerging with different basic architectures and accordingly varied levels of flexibility (e.g., [10, 14, 6, 1, 5]).

The order of these three classes generally reflects the increasingly larger hardware design effort. This paper focuses on the design of lightweight ad-hoc functional units.

One of the main problems in processor specialisation is to achieve an automatic design flow: the design of ad-hoc extensions can be very lengthy and burdensome, hence affordable only on mature products for an application. Unfortunately, these are not the ASIC products where processor specialisation matters most: newer technology generations are available at later stages of an application life, making design less challenging. This papers investigates the possibility of identifying automatically and with very low effort some relatively small clusters of combinatorial operations which can be collapsed in single operations of ad-hoc functional units. The application of these techniques spans from pure ASIC designs with extensible hard-core processors to processors equipped with embedded reconfigurable arrays. To minimise microarchitectural and technological assumptions, the analysis of the results focuses on the former type of designs without excluding the viability for the latter implementation.

### 1.1. Related work

Most of the related work on the identification of groups of logic and arithmetic operations to be mapped on ad-hoc functional units has been performed in conjunction with reconfigurable hardware. One of the first attempts in developing an automatic identifier is in the PRISC compiler [13]. The basic identification is a greedy approach that clusters as many operations executable on the reconfigurable unit as possible, with the limitation of at most two inputs to the unit. This constraint is not relaxed in following work by other groups [8] which focuses on specific challenges related to the reconfigurability of the unit. In the present work the constraint on the number of inputs is relaxed and its impact investigated. Additional ports from the register file need not necessarily be implemented exclusively for the ad-hoc functional units, but they might be temporarily borrowed from other functional units [1]. The authors of Chimaera [16] relax the number of inputs up to 9 and use a similar technique as the one addressed here; apart from a stronger microarchitectural abstraction, the present paper contains results without any attempts of branch conversion—which would result in better speedups—and shows that the selection of a very limited number of operation groups, rather than the execution of all of them on reconfigurable hardware, is perfectly viable.

Among the other identification techniques described in literature, one can mention Horizontal and Vertical Aggregation [7], which consists in a set of heuristic rules to grow large groups; the present work is similar in objective but uses a more formal methodology to achieve an algorithm which has linear complexity in the number of nodes. A rather different technique has been introduced based on the analysis of execution traces [2]: these are scanned to create a library of frequent operation patterns without specific constraints on the number of inputs and outputs. The technique is interesting but computationally very expensive compared to the one described here.

Finally it is worth mentioning that a significant amount of attention has been devoted in literature to the problem of designing optimal instruction sets for *Application Specific Instruction-set Processors* (ASIPs) (e.g., [11]) or in general to develop ad-hoc architectures (e.g., TTA [3]). Although not completely dissimilar, the present work has one fundamental difference: it is here assumed the availability of a complete standard processor to which the ad-hoc functional units are added to improve its performance. Hence, there is no need to cover the complete application with a minimal or otherwise optimal set of instructions: in the present work the goal is to indentify the smallest number of frequently executed kernels which most benefit when executed in closely coupled dedicated hardware.

## 2. Topological features

This paper addresses the problem of Instruction-Set Extension Identification under simple architectural conditions: the additional FUs do not have access to memory (i.e., no load/store instructions are considered in the search), and they have no storage (i.e., only data-flow computation is analysed). The proposed identification algorithm is applied to the Data Flow Graph of all basic blocks in the application. Topological features of the subgraphs representing the chosen instruction set extensions are naturally important for two reasons: Architecturally, the topology of the graph determines the number of read/write port needed, the latency of the resulting operation, etc. Algorithmically, the topology determines the algorithm needed for identification. It is believed that a subgraph to be collapsed into an instruction set extension should be:

1. Most suitable to be collapsed into one instruction only; in particular, architectural issues should be considered so that the solution is "viable".

2. Most easily identified, i.e., identified by an algorithm with acceptable complexity.

The above points lead to consider Multiple Inputs Single Output graphs, here called MISOs, and in particular MISO graphs of maximum kind. As shown theoretically[12] and by the experimental results, this kind of graph is an interesting candidate for automated instruction-set extension identification. In the following, some definitions are given together with some basic theoretical notions.

A formal definition of a MISO $M^i$ is as follows.

**Definition 1** *Denote by $G^i = \langle V^i, E^i \rangle$ a subgraph where $V^i$ is the set of nodes in $G^i$ and $E^i$ is the set of all edges departing from such nodes. An edge $e^i \in E^i$ is identified by its source node ($v_k^i \in V^i$) and its destination node $v_l^i$ and it is denoted by $e^i(k,l)$. If for all $v_k^i \in V^i$ excepting one node $v_O^i$ it is true that:*

$$\forall e^i(k,l) \in E^i, v_l^i \in V^i$$

*then G is MISO. A MISO is indicated as $M^i$, $v_O^i$ being its* output node.

In particular, the focus is on maximal MISOs, which are indicated as MaxMISOs. Their identification within a *Directed Acyclic Graph* (DAG) requires an algorithm of only linear complexity (described below).

**Definition 2** *A MaxMISO $MM^i$ is a MISO that is not completely included in any other MISO.*

MaxMISOs have an important property:

**Theorem 1** *Two MaxMISOs $MM^i, MM^j$ cannot partially overlap.*

**Proof.** The theorem is proved by contradiction. Assume $MM^i \cap MM^j \neq \emptyset$. This implies that at least one node $v^*$ in the DAG belongs to both $MM^i$ and $MM^j$. Then the alternative possibilities are:

1. Node $v^*$ is the output node for either $MM^i$ or $MM^j$ or both. Then, given the definition of MISO, there is a MISO $MM^k = MM^i \cup MM^j$. As a consequence, $MM^i$ and $MM^j$ are not maximum, contrary to the assumption.

2. Node $v^*$ is not an output node for either $MM^i$ or $MM^j$. Then, $v^*$ could not be the source of edges connecting to nodes in $MM^i$ (otherwise $MM^j$ would have more than one output and therefore would not be a MISO) nor to nodes in $MM^j$ ($MM^i$ would not be a MISO); as a consequence, $v^*$ could only belong to a subgraph not connected within both MISOs, contrary to initial assumption.

Hence $MM^i \cap MM^j = \emptyset$. □

The above property is used as the basis for the algorithm that extracts MaxMISOs from all basic blocks of an application.

## 2.1. Algorithm for MISO extraction

Figure 1 shows the pseudocode for the identification of all MaxMISOs within a DAG. The algorithm operates in two steps: first, a node is chosen to be the *exit_node*, then the program calls a function which builds the MaxMISO related to such *exit_node*. Exit nodes are chosen *upwards*, i.e., starting from the exits of the DAG. Initially, the set of *Nodes_to_be_analysed* coincides with the set of nodes of the DAG; afterwards, when a MaxMISO has been generated, its nodes are removed from the *Nodes_to_be_analysed* set. The function `Generate_MaxMISO` starts from the chosen *exit_node* and recursively tries to include its parents. Recursion ends when the encountered node is nonlegal (e.g., it is a load instruction) or has a nonreconvergent fanout.

```
$\forall Node \in Nodes\_to\_be\_analysed$ do
{
  Generate_MaxMISO(Node)
  $Nodes\_to\_be\_analysed\ -= Nodes\_in\_MaxMISO$
}

Generate_MaxMISO(Node)
{
  $\forall Parent\_of\_Node$ do
  {
    if $(fanout\_Parent\_of\_Node) == 1)$ {
      $include(Parent\_of\_Node)$
      Generate_MaxMISO(Parent_of_Node)
    }else
      $fanout\_Parent\_of\_Node --$
  }
}
```

**Figure 1. Pseudocode of the algorithm for MaxMISO identification**

The proposed algorithm shows a complexity linear with the number of nodes in the examined graph. This follows from Theorem 1: since MaxMISOs cannot overlap, each node is visited only once.

## 3. Experimental setup

The described technique for extension of the instruction set has been implemented and applied to almost all benchmarks of the *MediaBench* suite. The developed consists of the following stages: (1) A first pass of compilation into the SUIF [15] intermediate representation is applied, which reduces the C-code into a DFG/CFG representation closer to assembler, although still largely architecture independent. SUIF Data Flow nodes resemble generic assembler operations. (2) Profiling is then performed and annotated in the SUIF representation. (3) At this point, code analysis for instruction identification is performed. The identification algorithm described in Section 2.1 is applied to all basic blocks, and all candidates found are ranked in terms of

| | **1** opcode | **2** opcodes | **4** opcodes | **8** opcodes | **16** opcodes | **32** opcodes |
|---|---|---|---|---|---|---|
| **epic** | 24.4% (12.2%) | 42.7% (24.4%) | 44.6% (26%) | 47.2% (27.5%) | 51.5% (30%) | 54.7% (31.8%) |
| **unepic** | 4% (2.6%) | 7.9% (5.3%) | 15% (9.4%) | 25.9% (14.9%) | 43.4% (23.9%) | 54.2% (30.6%) |
| **g721encode** | 17.6% (8.8%) | 23% (13.5%) | 31% (20.3%) | 37.3% (23.7%) | 42.3% (27.5%) | 49.4% (33%) |
| **g721decode** | 17.2% (8.6%) | 22.9% (13.6%) | 31.2% (20.6%) | 37.8% (24.1%) | 43.1% (27.9%) | 50.1% (33.4%) |
| **gsmdecode** | 8.4% (5.6%) | 16.8% (11.2%) | 33.5% (22.4%) | 59.7% (37.2%) | 63% (39.8%) | 63.9% (40.4%) |
| **gsmencode** | 9.2% (7.7%) | 18.4% (15.4%) | 34.2% (24.9%) | 50.9% (35.5%) | 59.8% (41.9%) | 66% (46.1%) |
| **jpegtran** | 9.7% (9%) | 14.8% (11.6%) | 21.9% (15.1%) | 28.9% (18.9%) | 34.3% (22.9%) | 39% (25.8%) |
| **djpeg** | 3.6% (3%) | 6.4% (4.9%) | 10.4% (8.3%) | 18.1% (13.6%) | 27.1% (18.5%) | 37.2% (24.4%) |
| **cjpeg** | 6.6% (4.4%) | 10.8% (7.8%) | 16% (11.5%) | 24.5% (16.4%) | 35.1% (23%) | 47.8% (30.3%) |
| **mpeg2encode** | 19.4% (15.6%) | 31.7% (21.7%) | 56.2% (33.9%) | 60.6% (37.4%) | 61.2% (37.8%) | 61.7% (38.1%) |
| **mpeg2decode** | 14.9% (11.2%) | 29.8% (22.4%) | 48.4% (33.6%) | 55% (38.6%) | 59.7% (41.8%) | 63.3% (44.2%) |
| **pegwit** | 10.1% (8.1%) | 15.7% (12.8%) | 27.6% (22.4%) | 43.6% (35.2%) | 60.5% (47.1%) | 72% (55.4%) |
| **adpcmdecode** | 25% (16.6%) | 41.6% (29.1%) | 50% (35.4%) | 50% (35.4%) | 50% (35.4%) | 50% (35.4%) |
| **adpcmencode** | 14.5% (12.6%) | 28.9% (23.5%) | 38% (29.6%) | 38% (29.6%) | 38% (29.6%) | 38% (29.6%) |
| **rasta** | 5% (2.5%) | 8.7% (4.3%) | 13.3% (6.9%) | 20.5% (11.2%) | 28.9% (16.4%) | 41.9% (23.2%) |
| **md5** | 18.2% (16.6%) | 26.5% (24%) | 44.7% (38.9%) | 68.2% (54%) | 70% (54.9%) | 70% (54.9%) |

**Table 1. Cumulative weights and potential cycle savings for some additional specialised opcodes (between 1 and 32) on the *MediaBench* suite.**

"weight". Weight is calculated as the number of nodes contained in the candidate, multiplied by its static recurrence (number of occurrences in the code) and by its dynamic recurrence (number of times the basic block was visited during execution). (4) Finally, instruction selection is run after identification; here, simply the best $n$ candidates are chosen.

A number of assumptions are used to calculate the number of cycles which can be saved through addition of ad-hoc functional units: (1) Two assumptions are made to approximate the behaviour of a real processor from the microarchitecturally neutral SUIF nodes. For the unextended processor, each node is considered to execute in one cycle—i.e., a CPI of 1 is assumed. This corresponds for instance to a single-issue unextended machine with ideal behaviour, or to a double-issue unextended machine with approximately non-ideal (realistic) behaviour. Another assumption along the same lines consists in considering each SUIF node as a machine instruction; in fact, the exact nature of the mapping between SUIF nodes and machine instructions will depend on the particular target architecture, but the assumption tends to hold on average. (2) Candidates are considered to be executed in one cycle when collapsed into special instructions (this assumption is discussed in next section). (3) No code to spill registers to memory is considered. One can note that collapsing subgraphs into single instructions reduces register pressure and therefore spilling, thus increasing the benefits of ad-hoc functional units; such benefits have not been estimated yet. In summary, with the above assumptions, cycle savings enabled by the introduction of the application-specific opcodes were therefore calculated by considering that a candidate with $n$ nodes makes it possible to save $n-1$ cycles every time it is visited (executed).

| | **2** Inp. | **4** Inp. | **6** Inp. | **8** Inp. | $\infty$ Inp. |
|---|---|---|---|---|---|
| **gsmencode** | 35.5% | 35.5% | 35.5% | 35.5% | 39.3% |
| **djpeg** | 13.6% | 13.6% | 13.6% | 14.7% | 14.7% |
| **cjpeg** | 16.4% | 16.4% | 16.4% | 16.4% | 17.7% |
| **rasta** | 10.8% | 10.8% | 11.2% | 11.2% | 11.2% |
| **md5** | 24.2% | 24.2% | 54.0% | 54.0% | 54.0% |

**Table 2. Dependence of potential cumulative cycle saving of the 8 best MaxMISOs on the limit imposed to the MaxMISOs input count.**

## 4. Experimental results

Results obtained on the *MediaBench* suite are shown in Table 1, for a varying number of groups of instructions collpsed in new ad-hoc opcodes (from 1 to 32 additions). In the table, numbers are given for a limit of 6 inputs for each candidate. Although the algorithm presented in Section 2 identifies subgraphs with an undetermined number of inputs, in order to take into account microarchitectural limitations, the $n$ best candidates can be chosen among those with a number of inputs below the maximum allowable microarchitecturally. In particular, a restriction of a maximum of 6 inputs was imposed to calculate the weights and potential cycle reductions of Table 1. With a reasonable number of ad-hoc opcodes assigned (e.g., 8, column 5), one observes dynamic cycle savings between 11% and 54% with typical values around 30%, which indicate typical speedups

| Proc | graphtype | dyn. occ. | inputs | outputs | nodes | stat. occ. | weight | rel. weight | rel. weight | rel. cyc. saved |
|---|---|---|---|---|---|---|---|---|---|---|
| Transform | | | | | | | | | | |
| | **basic block** | 300 | 2 | 8 | 869 | 1 | 260,700 | 45.0% | | |
| | MaxMISO | 300 | 5 | 1 | 11 | 32 | 105,600 | | 18.2% | 16.6% |
| | MaxMISO | 300 | 5 | 1 | 10 | 16 | 48,000 | | 8.3% | 7.5% |
| | MaxMISO | 300 | 5 | 1 | 9 | 16 | 43,200 | | 7.5% | 6.7% |
| | MaxMISO | 300 | 1 | 1 | 2 | 66 | 39,600 | | 6.8% | 3.4% |
| MD5Update | | | | | | | | | | |
| | **basic block** | 19,192 | 3 | 4 | 6 | 1 | 115,152 | 19.9% | | |
| | MaxMISO | 19,192 | 2 | 1 | 2 | 1 | 38,384 | | 6.6% | 3.3% |
| | **basic block** | 4,784 | 3 | 2 | 34 | 1 | 162,656 | 28.1% | | |
| | MaxMISO | 4,784 | 4 | 1 | 10 | 1 | 47,840 | | 8.3% | 7.4% |
| | MaxMISO | 4,784 | 2 | 1 | 4 | 3 | 57,408 | | 9.9% | 7.4% |
| | MaxMISO | 4,784 | 2 | 1 | 3 | 1 | 14,352 | | 2.5% | 1.6% |
| | MaxMISO | 4,784 | 1 | 1 | 2 | 1 | 9,568 | | 1.7% | 0.8% |
| | **basic block** | 19,192 | 1 | 2 | 2 | 1 | 38,384 | 6.6% | | |
| **total** | | | | | | | | 100% | 70.1% | 54.7% |

**Table 3. Results for the MD5 application**

of $1.4\times$ up to above $2\times$.

Of course, different opcodes need not be executed by different functional units: a designer can implement several complex operations on the same functional unit to keep the total number of read/write ports reasonable. This might also result in an area saving in case of similar logic functions being required for different opcodes.

One should note that the complete toolchain is automatic and that the presented approach allows instruction-set extension identification without any human intervention.

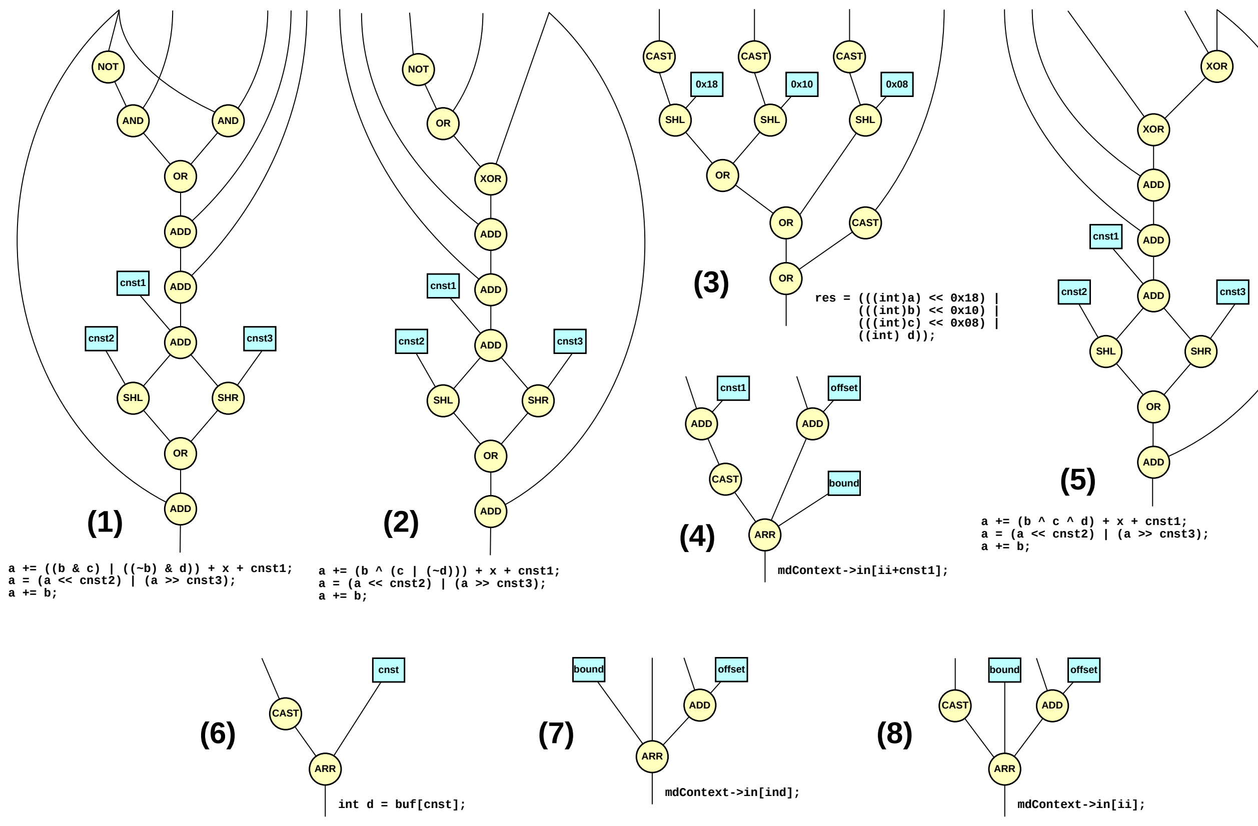
To analyse the importance of opcodes with a large number of inputs, Table 2 shows the evolution of cycle savings for some benchmarks; in each case, the 8 best opcodes are considered. The benchmarks shown are those which actually show a (mostly slight) difference of performance with the input count limitation; all remaining benchmarks show no dependence on input variation (constant number of cycle-savings).

This indicates that, apart from a few significant cases, even though the algorithm used for identification detects graphs with an unlimited number of inputs, most of the graphs chosen in practice have a very limited input requirement; hence they are very moderately demanding microarchitecturally. In cases like MD5, subgraphs identified do have larger than average sizes and a larger demand in terms of inputs; one of the merits of the presented approach is indeed to be able to identify a good instruction set extension for both kinds of applications.

The MD5 application is now analysed in more detail in order to better explain the estimation methodology used and to validate the one-cycle execution assumption. Table 3 shows the distribution of computational load across the basic blocks of the MD5 application, (only basic blocks with weights greater than 1% are pictured) and lists all MaxMISOs found. In column two, the graph type is given (basic block or MaxMISO), while column 3 contains the dynamic occurrency of the basic block during execution. Then, number of inputs, outputs (always one for a MISO), number of SUIF nodes, and number of static occurrencies within the code are shown, for basic blocks and for all the identified candidates. Finally, the weight column is calculated as the number of nodes multiplied by static and dynamic occurrences of a basic block or candidate. The next columns give respectively the relative weight of basic blocks, the relative weight of candidates, and the relative cycle saving of candidates. Note that the total relative weight for basic blocks does not add up to exactly 100% because rows with weights smaller than 1% were omitted.

Figure 2 depicts the best 8 candidates identified by the described algorithm. This example contains some among the largest candidates found and support the assumption that most identified subgraphs can be implemented in a single instruction and executed in a single cycle. Apart from most subgraphs being composed of only a few nodes (typically below 5–6), compressing them in a single cycle appears to be possible in many cases either because of their intrinsic parallelism (e.g., number 3 in the figure) or because of special arithmetic opportunities (e.g., numbers 1 and 2). To understand the latter, one can think how a multiply-accumulate operator has a minor timing difference from a pure multiplier, or how several additions in chain can be very efficiently performed in carry-save mode and the carry propagated only in the last addition, or finally how the very common shifts by constants are simple wires in hardware.

The current work is being extended by relaxing the single output constraint on identified subgraphs, and the first results in algorithm complexity and applicability are being generated; the purpose is to verify if the fact of relaxing the typical single-output microarchitectural restriction brings sufficient advantage. On the other hand, the current



```
a += ((b & c) | ((~b) & d)) + x + cnst1;
a = (a << cnst2) | (a >> cnst3);
a += b;
```

```
a += (b ^ (c | (~d))) + x + cnst1;
a = (a << cnst2) | (a >> cnst3);
a += b;
```

```
res = (((int)a) << 0x18) |
      (((int)b) << 0x10) |
      (((int)c) << 0x08) |
      ((int) d));
```

```
mdContext->in[ii+cnst1];
```

```
a += (b ^ c ^ d) + x + cnst1;
a = (a << cnst2) | (a >> cnst3);
a += b;
```

```
int d = buf[cnst];
```

```
mdContext->in[ind];
```

```
mdContext->in[ii];
```

**Figure 2. MD5's best 8 MISOs**

selection process is very rough when a limit on the number of inputs is defined: a large subgraph with many inputs is simply dropped whereas heuristics could be developed to prune it. On the practical side, attempts to port the approach on a real architecture are planned. In view of real hardware implementation of ad-hoc functional units, techniques to annotate the data graphs with the effective bit-widths can be used to model hardware with minimal complexity for the task.

## 5. Conclusions

This paper describes a formal setting and an efficient algorithm to partition automatically the data flow graph of an application: the arithmetic and logic operations are grouped in disjoint subgraphs which are maximal under the constraint of having a single output. A few identified subgraphs, most frequently executed, are good candidates to be collapsed into new operations and assigned to ad-hoc functional units. Almost any processor can easily support such a microarchitectural extension.

The results on a common multimedia benchmark suite indicate that even without sophisticated processing to expose additional parallelism such an automatic approach to the design of instruction-set is well feasible and brings a tangible advantage in almost all applications analysed. Even under the most conservative microarchitectural restrictions, interesting cycle reductions have been shown. Significantly, the requirements on the standard toolset of the original processor are very modest: the possibility of inlining the new opcodes in standard C code and of modelling them in the compiler and simulator would be enough to complete a fully automatic processor specialisation flow from C code to hardware and software.

## References

[1] S. Aditya, B. R. Rau, and V. Kathail. Automatic architectural synthesis of VLIW and EPIC processors. In *Proceedings of the 12th International Symposium on System Synthesis*, San Jose, Calif., Nov. 1999.

[2] M. Arnold and H. Corporaal. Designing domain specific processors. In *Proceedings of the 9th International Workshop on Hardware/Software Codesign*, pages 61–66, Copenhagen, Apr. 2001.

[3] H. Corporaal. *Microprocessor Architectures—From VLIW to TTA*. Wiley, New York, 1997.

[4] J. Eyre and J. Bier. Infineon targets 3G with Carmel2000. *Microprocessor Report*, 17 July 2000.

[5] P. Faraboschi, G. Brown, J. A. Fisher, G. Desoli, and F. Homewood. Lx: A technology platform for customizable VLIW embedded processing. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 203–13, Vancouver, June 2000.

[6] T. R. Halfhill. ARC Cores encourages “plug-ins”. *Microprocessor Report*, 19 June 2000.

[7] M. F. Jacome, G. de Veciana, and V. Lapinskii. Exploring performance tradeoffs for clustered VLIW ASIPs. In *Proceedings of the International Conference on Computer Aided Design*, pages 504–10, San Jose, Calif., Nov. 2000.

[8] B. Kastrup, A. Bink, and J. Hoogerbrugge. ConCISe: A compiler-driven CPLD-based instruction set accelerator. In *Proceedings of the 5th IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa Valley, Calif., Apr. 1999.

[9] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 330–35, Research Triangle Park, N.C., Dec. 1997.

[10] A. Nene, S. Talla, B. Goldberg, H. Kim, and R. M. Rabbah. *Trimaran—An Infrastructure for Compiler Research in Instruction Level Parallelism*. New York University, 1998. `http://www.trimaran.org/`.

[11] P. Paulin. Trends in embedded systems technology: An industrial perspective. In G. De Micheli and M. G. Sami, editors, *Hardware-Software Codesign*, volume 310 of *NATO Science Series: E Applied Sciences*, pages 311–37. Kluwer Academic, Boston, Mass., 1996.

[12] L. Pozzi. *Methodologies for the Design of Application-Specific Reconfigurable VLIW Processors*. Ph.D. thesis, Politecnico di Milano, Dipartimento di Elettronica e Informazione, Milano, Jan. 2000.

[13] R. Razdan and M. D. Smith. A high-performance microarchitecture with hardware-programmable functional units. In *Proceedings of the 27th International Symposium on Microarchitecture*, pages 172–80, San Jose, Calif., Nov. 1994.

[14] J. Turley. Tensilica CPU bends to designers’ will. *Microprocessor Report*, 8 Mar. 1999.

[15] R. Wilson, C. French, C. Wilson, J. Amarasinghe, J. Anderson, S. Tjiang, S.-W. Liao, M. Tseng, Hall, M. Lam, and J. Hennessy. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *SIGPLAN Notices*, 29:31–37, Dec. 1994.

[16] Z. A. Ye, A. Moshovos, S. Hauck, and P. Banerjee. CHIMAERA: A high-performance architecture with a tightly-coupled reconfigurable functional unit. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 225–35, Vancouver, June 2000.