Exploiting Pipelining to Relax Register-File Port Constraints of Instruction-Set Extensions

Laura Pozzi
Laura.Pozzi@epfl.ch

Paolo Ienne
Paolo.Ienne@epfl.ch

Ecole Polytechnique Fédérale de Lausanne (EPFL)
School of Computer and Communication Sciences
CH-1015 Lausanne, Switzerland

ABSTRACT

Customisable embedded processors are becoming available on the market, thus making it possible for designers to speed up execution of applications by using *Application-specific Functional Units* (AFUs), implementing *Instruction-Set Extensions* (ISEs). While these processors have become available, the state of the art on automatic ISE identification is improving; many algorithms are being proposed for choosing, given the application's source code, the best ISEs under various constraints. Read and write ports between the AFUs and the processor register file are an expensive asset, fixed in the microarchitecture—some processors indeed only allow two read and one write ports—and, on the other hand, a large availability of inputs and outputs to and from the AFUs exposes high speedup. This paper proposes a solution to the limitation of actual register file ports by serialising register file access and therefore addressing multi-cycle read and write. It does so in an innovative way for two reasons: (1) it exploits and brings forward the progress in ISE identification under constraint [1, 16, 4, 19], and (2) it combines register file access serialisation with pipelining in order to obtain the best global solution. This paper proposes an algorithm for scheduling graphs—corresponding to ISEs—under input/output constraint; experiments show that by using the proposed method applications can be sped-up tangibly: speedup for low I/O constraints is 32% better on average, and 65% better at best, than that obtained by state of the art techniques.

Categories and Subject Descriptors

C.3 [**Computer Systems Organization**]: Special-Purpose and Application-Based Systems

General Terms

Algorithms, Design, Performance

CASES'05, September 24–27, 2005, San Francisco, California, USA.

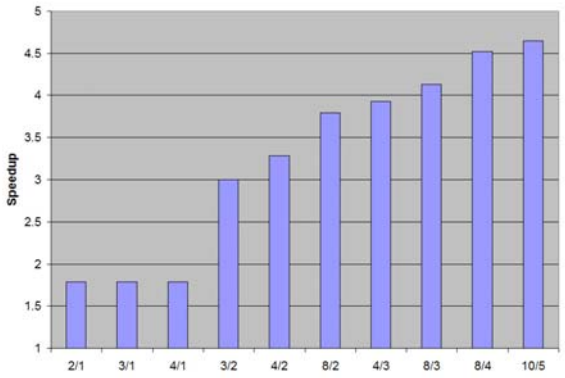

Figure 1: ISE performance on the *des* cryptography algorithm, as a function of the I/O constraint.

Keywords

Automatic Instruction-Set Extension, Input/Output Constrained Scheduling, Multi-cycle Register Access, Embedded Customised Architectures

1. INTRODUCTION AND MOTIVATION

Customisable Processors represent an emerging and effective paradigm for executing embedded application under high performance, short time to market, and low power requirements. Among the possible customisation directions, a particularly interesting one is that of *Instruction-Set Extensions* (ISE): *Application-specific Functional Units* (AFUs) can be added to the processor core in order to speed up a particular application and implement specialised instructions. As these processors become available—e.g., Tensilica Xtensa [11], ARC ARCtangent [8], STMicroelectronics ST200 [7], and MIPS CorExtend [10]—techniques are emerging for automatically selecting the best ISEs for an application, given the application source code and under various constraints.

A particularly expensive asset of the processor core is the number of ports to the register file that the AFUs are allowed to use. While this number is typically kept small in available processors—indeed some only allow two read and one write ports—it is also true that input/output allowance impacts directly on speedup. A typical trend can be seen in Figure 1, where the speedup for various combinations of

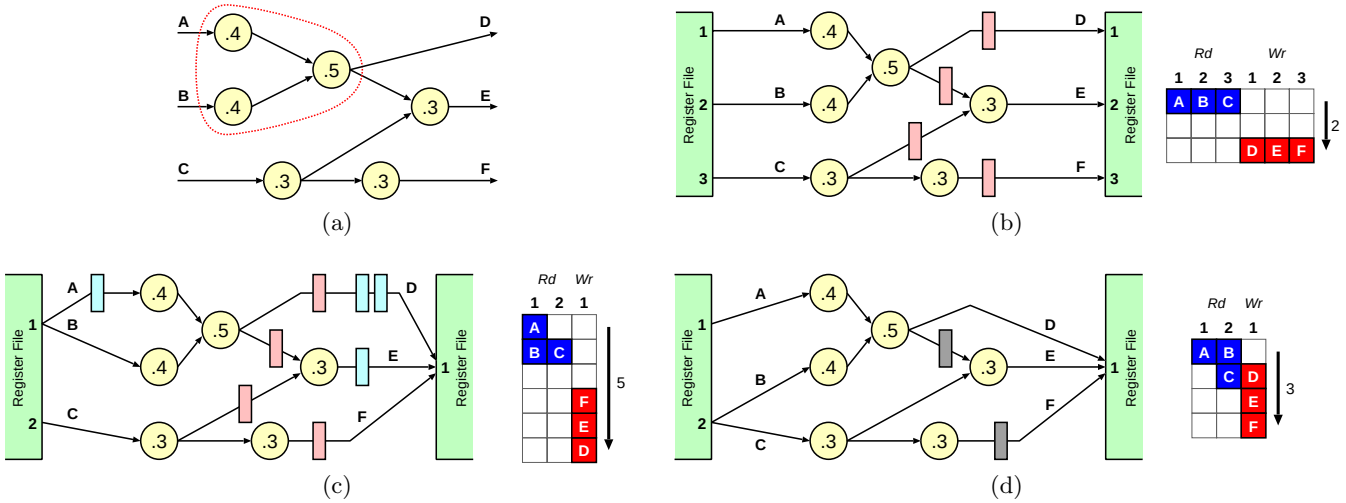

Figure 2: Motivational Example. (a) The DAG of a basic block annotated with the delay in hardware of the various operators. (b) A possible connection of the pipelined datapath to a register file with 3 read ports and 3 write ports (latency = 2). (c) A naive modification of the datapath to read operands and write results back through 2 read ports and 1 write port, resulting in a latency of 5 cycles. (d) An optimal implementation for 2 read ports and 1 write port, resulting in a latency of 3 cycles. Rectangles on the DAG edges represent pipeline registers. All implementations are shown with their I/O schedule on the right.

I/O constraints is shown, for an application implementing the *des* cryptography algorithm. While an explanation of the experimental setup is detailed later, here the aim is to show that, on a typical embedded application, the I/O constraint impacts strongly on the potentiality of ISE: speedup goes from 1.7 for 2 read and 1 write ports, to 4.5 for 10 read and 5 write ports. Intuitively, if the I/O allowance increases, larger portions of the application can be mapped onto an AFU, and therefore a larger part can be accelerated.

As a motivational example, consider Figure 2(a), representing the *Direct Acyclic Graph* (DAG) of a basic block. Assume that each operation occupies the execution stage of the processor pipeline for one cycle when executed in software. In hardware, the delay in cycles (or fraction thereof) of each operator is shown inside each node. Under an I/O constraint of 2/1, the subgraph indicated with a dashed line on Figure 2(a) is the best candidate for ISE. Its latency is one cycle (ceiling of the subgraph's critical path), while the time to execute the subgraph on the unextended processor is roughly 3 cycles (one per operation). Two cycles are therefore saved every time the ISE is used instead of executing the corresponding sequence of instructions. Under an I/O constraint of 3/3, on the other hand, the whole DAG can be chosen as an AFU (its latency in hardware is 2 cycles, its software latency is approximately 6 cycles, and hence 4 cycles are saved at each invocation). Figure 2(b) shows a possible way to pipeline the complete basic block into an AFU, but this is exclusively possible if the register file has 3 read and 3 write ports. If the I/O constraint is 2/1, a common solution is to implement the smaller subgraph instead, and reduce significantly the potential speedup.

We claim that an ISE identification algorithm should search for candidates exceeding the constraint, and then map them on the available I/O by serialising register port access. Figure 2(c) shows a naive way to implement serialisation, which simply (i) maintains the position of pipelines registers as it was in Figure 2(b) and (ii) adds registers at the beginning and at the end to account for serialised access. As indicated in the I/O access table, value A is read from the register file in a first cycle, then values B and C are read and execution starts. Finally, two cycles later, the results are written back in series into the register file, in the predefined (and naive) order of F, E and D. The schedule is legal since only at most 2 read and/or 1 write happen simultaneously. Latency, calculated from the first read to the last write, is now 5 cycle: only 1 cycles is saved. However, a better schedule for the DAG can be constructed by changing the position of the original pipeline registers, in order to allow that register file access and computation can proceed in parallel. Figure 2(d) shows the best legal schedule, resulting in a latency of 3 cycles and hence a gain of 3 cycles: searching for larger AFU candidates and then pipelining them in an efficient way, in order to serialise register file access and to ensure I/O legality, can be beneficial and, as it will be seen in the experimental section, it can boost the performance of ISE identification.

This paper proposes an ISE identification algorithm that recognises the possibility of serialising operand-reading and result-writing of AFUs that exceed the processor I/O constraints. It also presents an algorithm for input/output constrained scheduling that minimises the resulting latency of the chosen AFUs by combining pipelining with multi-cycle register file access. Measurements of the obtained speedup show that the proposed algorithm finds high-performance schedules resulting in tangible improvement when compared to the single-cycle register file access case.

The rest of this paper is organised as follow: Section 2 discusses related work and Section 3 described the ISE strategy that we follow. Section 4 focuses on I/O constrained scheduling, and defines it formally, while Section 5 proposes an algorithm for solving it. A description of the experimental setup is in Section 6 and results validating the proposed

ideas are described in Section 7. Finally, Section 8 concludes and proposes ways forward.

2. RELATED WORK

Discussion of the state of the art is here divided in two parts: the first relates to scheduling and pipelining, while the second details works on automatic Instruction-Set Extension.

A well known *unconstrained* scheduling for minimum latency is ASAP, while many scheduling algorithms under constraint have been presented, such as resource-constrained and time-constrained [6]. Resource-constrained scheduling limits the number of computational resources that can be used in a cycle; it is an intractable problem, and list scheduling is a heuristic used for solving it. Proposed solutions to time-constrained scheduling, where relative timing constraints between operations are specified, include Force Directed Scheduling [15] and integer linear programming. This paper defines and solves another type of constrained scheduling, called here *I/O constrained scheduling*, which finds the minimum latency schedule for a DAG under the constraint that no more than N_{in} inputs and no more than N_{out} outputs can be read and written in any given cycle. It can be seen as a special case of resource-constrained scheduling. Retiming algorithms are also related to this work, where registers are moved in a circuit in order to optimise performance or area. In particular, a reported algorithm for retiming DAGs [3] is similar to a step of the I/O constrained scheduling algorithm presented here.

The problem of identifying instruction-set extensions consists in detecting clusters of operations which, when implemented as a single complex instruction, maximise some metric—typically performance. Such clusters must invariably satisfy some constraint; for instance, they must produce a single result or use no more than four input values. The problem solved by the algorithms presented in this paper is formalised in Section 3, but this generic formulation is used here to discuss related work.

Some methods have been proposed where authors essentially concentrate on targeting maximal reuse of complex instructions [13, 2, 17]. In this case, sequences or simple clusters of operations often appear as the best candidates. The importance of growing larger clusters for high speedup is acknowledged in some recent works [1, 16, 4, 19]. Another recent formulation, experimented on the Nios II processor, uses an exponential enumeration algorithm to find all patterns with a single output [5]; the algorithm is usable in practice in the given microarchitectural context by limiting the number of inputs.

Work on *Application Specific Instruction-set Processors (ASIPs)* generation is also related to ISE identification, but it differs from the latter because it involves generation of complete instruction sets for specific applications [12, 18].

The present work combines any ISE identification algorithm that works under constraint [1, 16, 4, 19] with AFU pipelining and I/O constrained scheduling. It recognises the possibility of serialising access to the register file and identifies AFUs with larger I/O constraint than the allowed microarchitectural one; then, it automatically maps them to the actual read/write port availability. To the best of our knowledge, this is the first work that proposes a solution to exploit this possibility in an automatic way.

3. ISE SELECTION

The global problem that we want to solve is essentially the *single-cut identification* problem addressed in prior work [1]: we want to find a convex subgraph S of the *Data Flow Graph* (DFG) of a basic block. The subgraph S, which we call *cut*, represents the functionality to be implemented in a specialised functional unit. The cut S therefore maximises some merit function $\mathrm{M}(S)$, which represents the speedup achieved when the cut is implemented as a custom instruction, while input and output nodes of S are such as to allow implementation with a limited number of register-file ports—that is, $\mathrm{IN}(S) \leq N_{\text{in}}$ and $\mathrm{OUT}(S) \leq N_{\text{out}}$, where the constants N_{in} and N_{out} depend from the microarchitecture. Finally, S must be a convex graph to guarantee schedulability in typical compilers.

In the present work, our goal differs from the basic formulation above (formalised in [1]) because of two related reasons: (a) we allow the cut S to have more inputs than the read ports of the register file and/or more outputs than the write ports; if this happens, (b) we account for successive transfers of operands and results to and from the specialised functional unit in the latency of the special instruction. We take care of point (b) while we also introduce pipeline registers, if needed, in the datapath of the unit.

The way we solve the new single-cut identification problem consists of three steps: (1) We generate the best cuts for an application using any ISE identification algorithm (e.g., the single-cut identification of [1]) for all possible combinations of input and output counts equal and above N_{in} and N_{out}, and below a reasonable upper bound, e.g., 10/5. (2) We add to the DFG of S both the registers required to pipeline the functional unit under a fixed timing constraint (the cycle time of the host processor) and the registers to store temporarily excess operands and results. In other words, we fit the actual number of inputs and outputs of S to the microarchitectural constraints. (3) We select the best ones among all cuts. Step (2) is the actual problem that we formalise and solve in the present paper.

4. PROBLEM STATEMENT

We call $S(V, E)$ the DAG representing the dataflow of a potential special instruction to be implemented in hardware; the nodes V represent primitive operations and the edges E represent data dependencies. Each graph S is associated to a graph

$$S^{+}\left(V \cup I \cup O \cup \{v_{\text{in}}, v_{\text{out}}\}, E \cup E^{+}\right)$$

which contains additional nodes I, O, v_{in}, and v_{out}, and edges E^{+}. The additional nodes I and O represent, respectively, input and output variables of the cut. The node v_{in} is called *source* and has edges to all nodes in I. Similarly, the node v_{out} is the *sink* and all nodes in O have an edge to it. The additional edges E^{+} connect the source to the nodes I, the nodes I to V, V to O, and O to the sink. Figure 3 shows an example of cut.

Each node $u \in V$ has associated a positive real weight, $\lambda(u)$; it represents the latency of the component implementing the corresponding operator. Nodes v_{in}, v_{out}, I, and O have a null weight. Each edge $(u, v) \in E$ has an associated positive integer weight, $\rho(u, v)$; it represents the number of registers in series present between the adjacent operators. A null weight on an edge indicates a direct connection (i.e., a

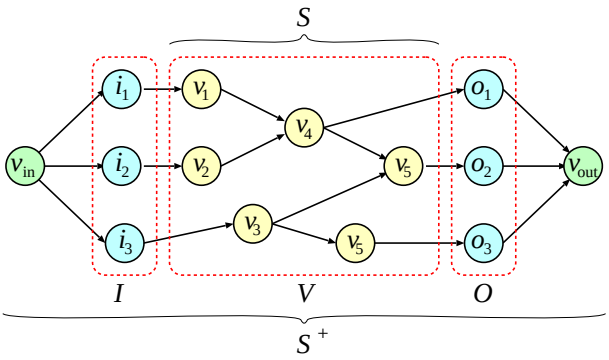

Figure 3: A sample augmented cut S^+.

wire). Initially all edge weights are null (that is, the cut S is a purely combinatorial circuit).

Our goal is to modify the weights of the edges of S^+ in such a way as to have (1) the critical path (maximal latency between inputs and registers, registers and registers, and registers and outputs) below or equal to some desired value Λ, (2) the number of inputs (outputs) to be provided (received) at each cycle below or equal to N_{in} (N_{out}), (3) a minimal number of pipeline stages, R. To express this formally, we introduce the sets W_i^{IN} which contain all edges (v_{in}, u) whose weight $\rho(v_{\text{in}}, u)$ is equal to i. Similarly the sets W_i^{OUT} contain all edges (u, v_{out}) whose weight $\rho(u, v_{\text{out}})$ is equal to i. We write $\left|W_i^{\text{IN}}\right|$ to indicate the number of elements in the set W_i^{IN}. The problem we want to solve is the particular case of scheduling described below.

PROBLEM 1. *Minimise R under the following constraints:*

1. ***Pipelining.*** *For all combinatorial paths between $u \in S^+$ and $v \in S^+$—that is, for all those paths such that $\sum_{\text{all edges } (s,t) \text{ on the path}} \rho(s,t) = 0$,*

$$\sum_{\text{all nodes } k \text{ on the path}} \lambda(k) \leq \Lambda. \tag{1}$$

2. ***Legality.*** *For all paths between v_{in} and v_{out},*

$$\sum_{\text{all edges } (u,v) \text{ on the path}} \rho(u,v) = R - 1. \tag{2}$$

3. ***I/O schedulability.*** *$\forall i \geq 0$,*

$$\left|W_i^{\text{IN}}\right| \leq N_{\text{in}} \text{ and } \left|W_i^{\text{OUT}}\right| \leq N_{\text{out}}. \tag{3}$$

The first bullet ensures that the circuit can operate at the given cycle time Λ. The second ensures a legal schedule, that is, a schedule which guarantees that the operands of any given instruction arrive together. The third bullet defines a schedule of communication to and from the functional unit that never exceeds the available register ports: for each edge (v_{in}, u), registers $\rho(v_{\text{in}}, u)$ do not represent physical registers, but the schedule used by the processor decoder to access the register file. Similarly, for each (u, v_{out}), $\rho(u, v_{\text{out}})$ indicates when results are to be written back. For this reason, registers on input edges $(v_{\text{in}}, u)$and on output edges (u, v_{out})will be called *pseudoregisters* from now on; in all figures, they are shown with a lighter colour than physical registers. As an

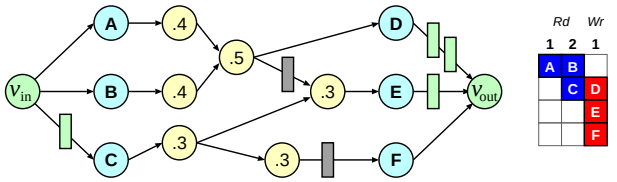

Figure 4: The graph S^+ of the optimised implementation shown in Figure 2(d). All constraints of Problem 1 are verified and the number of pipeline stages R is minimal.

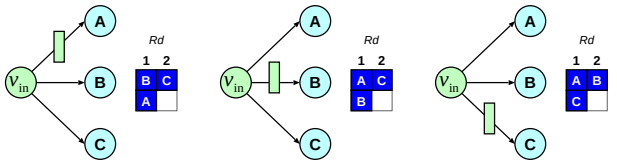

Figure 5: All possible input configurations for the motivational example, obtained by repeatedly applying an *n choose r* pass to the input nodes.

example, Figure 4 shows the graph S^+ of the optimised implementation shown in Figure 2(d) with the pseudoregisters which express the register file access schedule for reading and writing. Note that the graph satisfies the legality check expressed above: exactly two registers are present on any given path between v_{in} and v_{out}.

5. ALGORITHM

The algorithm proposed for solving Problem 1 first generates all possible pseudoregisters configurations at the inputs, meaning that pseudoregisters are added on input edges (v_{in}, u) in all ways that satisfy the input schedulability constraint, i.e., $\left|W_i^{\text{IN}}\right| \leq N_{\text{in}}$. This is obtained by repeatedly applying the *n choose r* problem—or *r combinations of an n set*—with $r = N_{\text{in}}$ and $n = |I|$, to the set of input nodes I of S^+, until all input variables have been assigned a read-slot—i.e., until all input edges (v_{in}, u) have been assigned a weight $\rho(v_{\text{in}}, u)$. Considering only the r combinations ensures that no more than N_{in} input values are read at the same time. The number of n choose r combinations is $\binom{n}{r} = \frac{n!}{r!(n-r)!}$. By repeatedly applying *n choose r* until all inputs have been assigned, the number of total configurations becomes $\frac{n!}{(r!)^x (n-xr)!}$, with $x = \lceil \frac{n}{r} \rceil - 1$. Note that the complexity of this step is exponential *in the number of inputs of the graph*, which is a very limited quantity in practical cases (e.g., in the order of tens). Figure 5 shows the possible configurations for the simple example of Figure 2: $I = \mathsf{A}, \mathsf{B}, \mathsf{C}$ and the configurations, as defined above, are $\mathsf{AB} \rightarrow \mathsf{C}$, $\mathsf{AC} \rightarrow \mathsf{B}$ and $\mathsf{BC} \rightarrow \mathsf{A}$. Note that the above definition does not include, for example, $\mathsf{A} \rightarrow \mathsf{BC}$. In fact, since we are scheduling for minimum latency, as many inputs as possible are read every time.

Then, for every input configuration, the algorithm proceeds in 3 steps:

(1) A scheduling pass, described in Figure 6, is applied to the graph, visiting nodes in topological order. The algorithm essentially computes an ASAP schedule, but it differs from a general ASAP version because it considers an initial pseudoregister configuration. It is an adaptation of a retiming

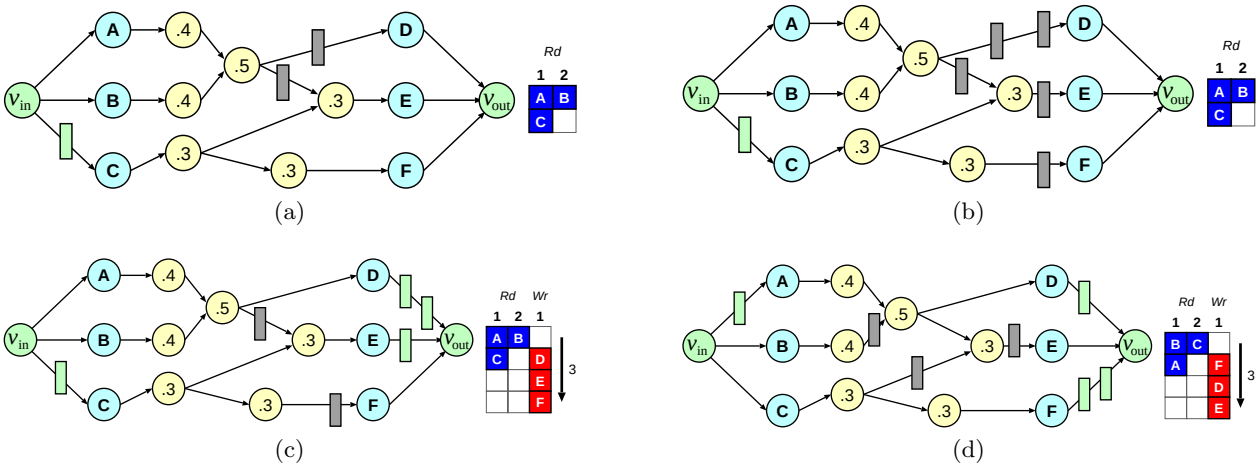

Figure 7: Proposed algorithm. (a) The scheduling pass of Figure 6 is applied to the graph, for the third initial configuration of Figure 5. The schedule is legal at the inputs but not at the outputs. (b) One line of registers is added at the outputs. (c) Three registers at the outputs are transformed into pseudoregisters, in order to satisfy the output constraint. (d) The final schedule for another input configuration. Its latency is also equal to three, but three registers are needed; this configuration is therefore discarded.

```
// path_weight for edges (v_in, u) set to input configuration
// path_weight for other edges initialised to 0
// path_delay initialised to 0
forall_nodes(u ∈ V ∪ I ∪ O ∪ {v_out}) {
  max_pw = max (path_weight of all in_edges of u);
  max_CP_delay = max (CP_delay of all in_edges with max_pw);
  if((max_CP_delay + delay(u) > Λ) {
    additional_reg = 1;
    CP_delay(u) = delay(u);
  } else {
    additional_reg = 0;
    CP_delay(u) = max_CP_delay + delay(u);
  }
  tot_pw = max_pw + additional_reg;
  forall_in_edges(in_e, u)
    weight(in_e) = tot_pw - path_weight(in_e);
  forall_out_edges(out_e, u)
    path_weight(out_e) = tot_pw);
}
```

Figure 6: Pseudocode of the ASAP algorithm. For every node u, `path_delay`(u) indicates the maximum delay among paths to the node that have no registers, and `delay`(u) indicates its individual delay, λ. For every edge e, `path_weight`(e) indicates the maximum number of registers from the source node v_{in} to the edge, and `weight`(e) indicates the number of registers on the edge itself, ρ.

algorithm for DAGs [3] and its complexity is $O(|V| + |E|)$. Figure 7(a) shows the result of applying the scheduling algorithm to one of the configurations.

(2) The schedule is now legal at the inputs but not necessarily at the outputs, and some registers might have to be added. The schedule is legal at the output only if at most N_{out} edges to output nodes have 0 registers (i.e., a weight equal to zero), at most N_{out} edges to output nodes have a weight equal to 1, and so on. If this is not the case, a line of registers on all output edges is added until the previously mentioned condition is satisfied. Figure 7(b) shows the result of this simple step.

(3) Registers at the outputs are transformed into pseudoregisters (i.e., they are moved to the right of output nodes, on edges (u, v_{out})), as shown in Figure 7(c). The schedule is now legal at both inputs and outputs.

All schedules of minimum latency are the ones that solve Problem 1. Among them, a schedule requiring a minimum number of registers is then chosen. Figure 7(d) shows the final schedule for another input configuration which has the same latency but a larger number of registers (3 vs. 2) than the one of Figure 7(c).

6. EXPERIMENTAL SETUP

In order to measure the speedup achieved by the algorithms described in this paper, a particular function $\mathrm{M}(\cdot)$ is assumed to express the merit of a specific S. $\mathrm{M}(S)$ represents an estimation of the speedup achievable by executing S as a single instruction in a specialised datapath.

In software, we estimate the latency in the execution stage of each instruction; in hardware, we evaluate the latency of each operation by synthesising arithmetic and logic operators on a common $0.18\mu m$ CMOS process and normalise to the delay of a 32-bit multiply-accumulate.

The accumulated software values of a cut estimate its execution time in a single-issue processor. The latency of a cut as a single instruction is calculated using the algorithm detailed in Section 5, with and without I/O constraint. The difference between the software and hardware latency is used to estimate the speedup. Although quite rough, this model gives a reasonable estimation of the potentials of ISE.

7. RESULTS

The proposed method was tested on some MediaBench [14], EEMBC [9], and cryptography benchmarks. Application C-code was compiled to MachSUIF intermediate representation, preprocessed with an if-conversion pass, and then analysed for ISE identification.

As described in Section 3, we proceed in two steps: (1) We run the ISE algorithm [16] with any combination of constraints greater or equal than the constraints allowed by the microarchitecture, and up to 10/5. Throughout this section,

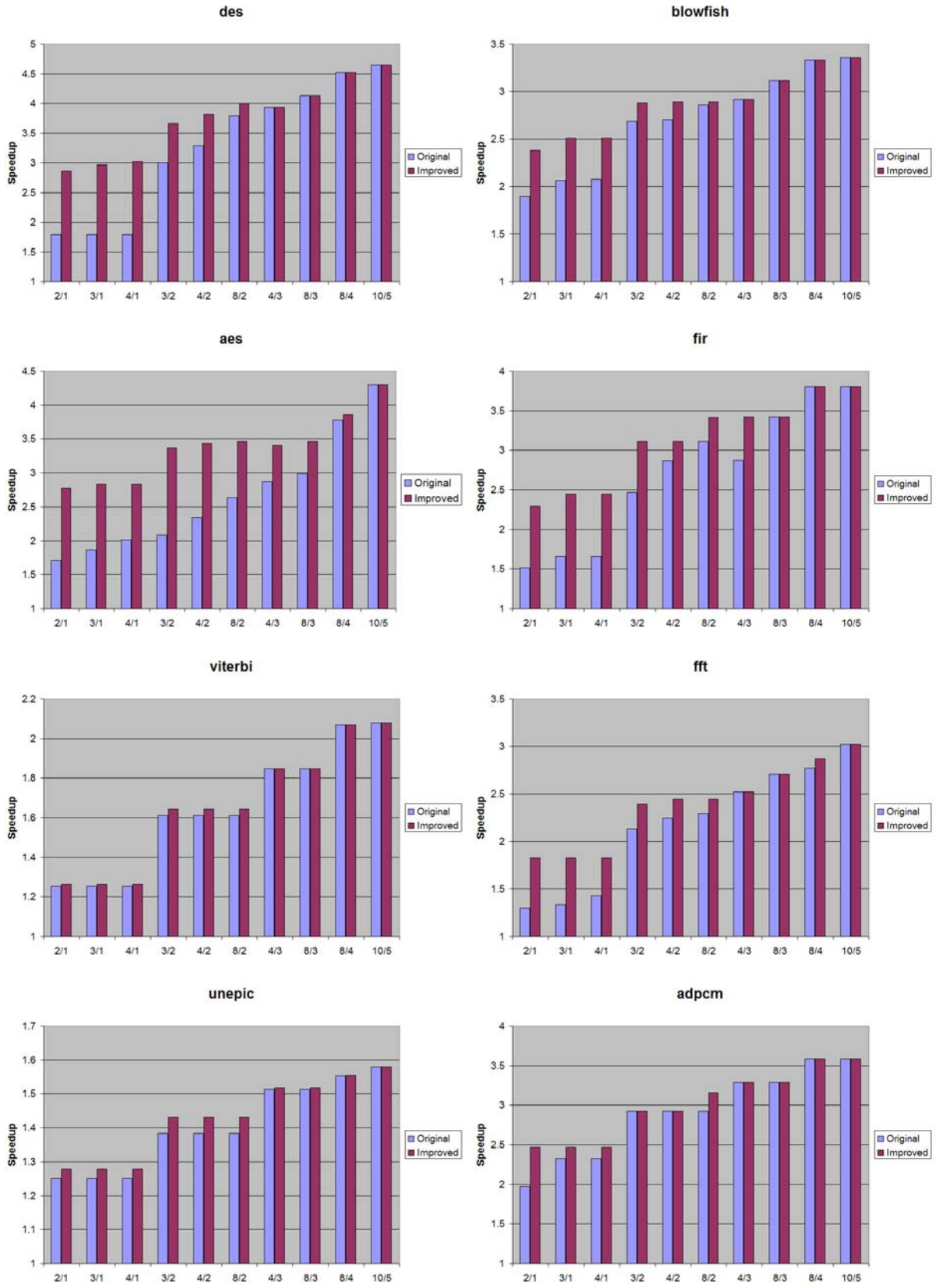

Figure 8: Analysis of the performance of the proposed method.

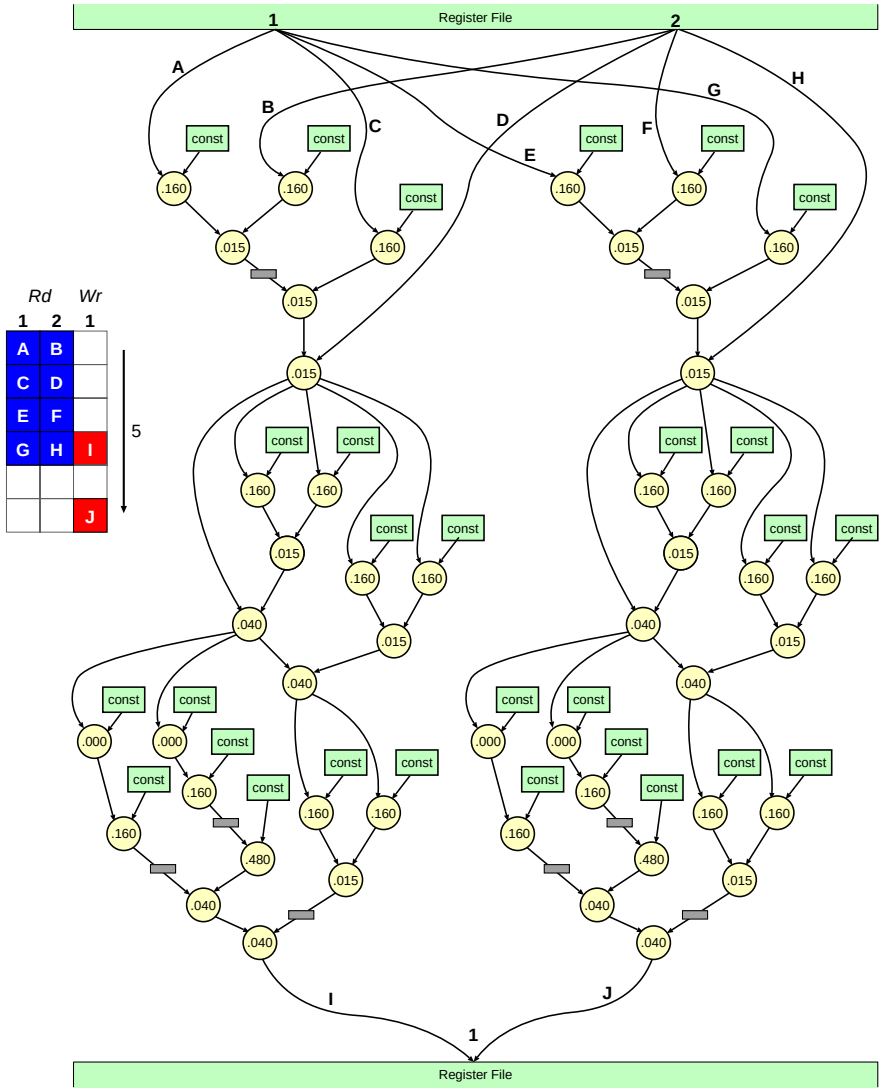

Figure 9: Sample pipelining for a 8/2 cut from the *aes* cryptography algorithm with an actual constraint of 2/1. Compared to a naive solution, this circuit saves eleven registers and shortens the latency by a few cycles.

we call the former constraints *temporary*, and the latter *actual*, or *microarchitectural*. (2) We schedule the identified AFUs using the algorithm described in Section 5, in order to generate an implementation that satisfies the actual constraints and to calculate the new latency.

In these experiments, we tried 10 different combinations of temporary I/O constraints, from 2/1 to 10/5, and the best 16 AFUs for each combination were identified. Then, all of the AFUs were mapped from the temporary constraint combinations onto actual, and smaller, microarchitectural constraints. Finally, the 16 I/O-constrained AFUs from the temporary combination that gave the best speedup were selected as final ones. Experiments were repeated for 10 different combinations of microarchitectural I/O constraint, as well.

Figure 8 shows the speedup obtained for each of the benchmarks, for various I/O constraints. Two columns are present for each I/O combination: The first shows the speedup of the original algorithm [16], where only AFUs with I/O requirement no greater than the microarchitectural allowance are considered. The second shows the speedup obtained by using the selection strategy and the I/O constrained scheduling algorithm presented in this paper, where all AFUs with I/O requirements greater or equal than the actual microarchitectural are considered, and then are rescheduled accordingly.

It can be observed that the speedup is raised tangibly in most cases. The improvement for the 2/1 constraint is on average 32%, and in some cases is as much as 65%, showing that the method presented in this paper represents yet another advance on the state of the art on automatic identification and selection of Instruction-Set Extensions. Some further points that can be noted follow intuition: (1) the difference in speedup between first and second columns decreases as the I/O increases—there is less margin for improvement—and therefore the proposed technique is particularly bene-

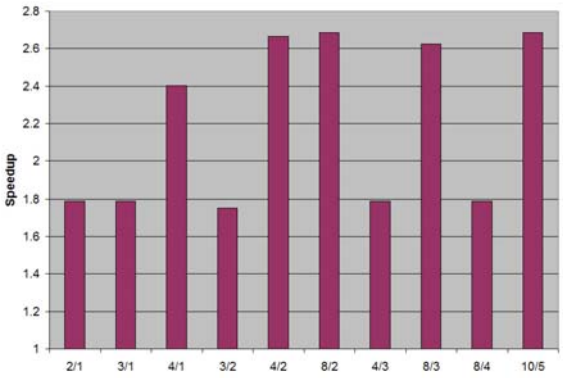

Figure 10: ISE performance on the *des* cryptography algorithm as a function of the temporary constraint for an actual constraint of 2/1.

ficial for low microarchitectural constraints; (2) as I/O increases, the speedup values obtained by the strategy presented in this paper (second columns) tend towards the value of the 10/5 first column, without actually reaching it since of course register-file access serialisation increases latency in general.

Figure 9 shows an example of 8/2 cut which has been pipelined and whose inputs and outputs have been appropriately sequentialised to match an actual 2/1 constraint. The example has an overall latency of five cycles and contains only eight registers (and six of them are essential for correct pipelining). With the naive solution illustrated in Figure 2(c), twelve registers (one each for C and D, two each for E and F, etc.) would have been necessary to resynchronise sequentialised inputs (functionally replaced here by the two registers close to the top of the cut) and one additional register would have been needed to delay one of the two outputs: our algorithm makes good use of the data independence of the two parts of the cut and reduces both hardware cost and latency. This example also suggests some ideas for further optimizations: if the symmetry of the cut had been identified, the right and left datapath could have been merged and the single datapath could have been used successively for the two halves of the cut. This would have produced the exact same schedule at an approximately half hardware cost, but the issues involved in finding this solution go beyond the scope of this paper.

Finally, Figure 10 shows the speedup relative to all temporary I/O constraints, for an actual constraint of 2/1, for the *des* application. The best value, seen in columns 8/2 and 10/5, is the one reported in Figure 8 for *des* (as second column of constraint 2/1). As it can be seen, the speedup does not always grow with the temporary constraints (contrast, for instance, 4/2 with 4/3). This is due to the fact that sometimes selection under certain high temporary constraint might include new input nodes of moderate benefit that later cannot be serialised without a large loss in speedup (too large increase in latency). Note that the value for constraint 2/1 in this graph is the same as the value of the first column for 2/1 on Figure 8 for *des*. Mapping AFUs, generated with a temporary constraint, onto an identical actual constraint is the same as using the original, non improved, ISE algorithm.

As a final remark, note that the presented I/O constrained algorithm ran in fractions of seconds in all cases. In fact, the complexity of the I/O constrained scheduling algorithm proposed is exponential in the number of inputs of the input graph, which is a very limited number in all practical cases. The cycle time was set, in these experiments, to the latency of a 32×32 multiply-accumulate. Of course, any other user-given cycle-time can be used.

8. CONCLUSIONS

This paper presents a novel strategy for automatic Instruction-Set Extension, which combines register file access serialisation and AFU pipelining in order to relax the constraint derived from limited register file ports availability. It also describes an algorithm for achieving I/O constraint scheduling, a way to map AFU circuits exceeding microarchitectural read/write port constraint onto ones that respect it. The paper improves the state of the art in automatic Instruction-Set Extension by being the first to accelerate automatically portions of application initially violating I/O constraints, disregarded by previous work. Results show that speedup is tangibly improved: 32% better than the state of the art on average, and 65% at best, for low microarchitectural constraints. Future work will address more advanced selection methods, and different possibilities of software scheduling of the proposed ISEs.

9. REFERENCES

[1] Kubilay Atasu, Laura Pozzi, and Paolo Ienne. Automatic application-specific instruction-set extensions under microarchitectural constraints. In *Proceedings of the 40th Design Automation Conference*, pages 256–61, Anaheim, Calif., June 2003.

[2] Philip Brisk, Adam Kaplan, Ryan Kastner, and Majid Sarrafzadeh. Instruction generation and regularity extraction for reconfigurable processors. In *Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*, pages 262–69, Grenoble, France, October 2002.

[3] P. Y. Calland, A. Mignotte, O. Peyran, Y. Robert, and F. Vivien. Retiming DAG's. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(12):1319–25, December 1998.

[4] Nathan Clark, Hongtao Zhong, and Scott Mahlke. Processor acceleration through automated instruction set customisation. In *Proceedings of the 36th Annual International Symposium on Microarchitecture*, San Diego, Calif., December 2003.

[5] Jason Cong, Yiping Fan, Guoling Han, and Zhiru Zhang. Application-specific instruction generation for configurable processor architectures. In *Proceedings of the 2004 ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays*, pages 183–89, Monterey, Calif., February 2004.

[6] Giovanni De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, New York, 1994.

[7] Paolo Faraboschi, Geoffrey Brown, Joseph A. Fisher, Giuseppe Desoli, and Fred 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.

[8] Tom R. Halfhill. ARC Cores encourages "plug-ins". *Microprocessor Report*, 19 June 2000.

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

[10] Tom R. Halfhill. MIPS embraces configurable technology. *Microprocessor Report*, 3 March 2003.

[11] Tom R. Halfhill. Tensilica's software makes hardware. *Microprocessor Report*, 23 June 2003.

[12] Bruce Kester Holmer. *Automatic Design of Computer Instruction Sets*. Ph.D. thesis, University of California, Berkeley, Calif., 1993.

[13] Ryan Kastner, Adam Kaplan, Seda Ogrenci Memik, and Elaheh Bozorgzadeh. Instruction generation for hybrid reconfigurable systems. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 7(4):605–27, October 2002.

[14] Chunho Lee, Miodrag Potkonjak, and William 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., December 1997.

[15] Pierre G. Paulin and John P. Knight. Force-directed scheduling for the behavioral synthesis of ASIC's. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 8(6):661–79, June 1989.

[16] Laura Pozzi, Kubilay Atasu, and Paolo Ienne. Optimal and approximate algorithms for the extension of embedded processor instruction sets. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2005. To appear.

[17] Fei Sun, Srivaths Ravi, Anand Raghunathan, and Niraj K. Jha. Custom-instruction synthesis for extensible-processor platforms. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, CAD-23(2):216–28, February 2004.

[18] Johan Van Praet, Gert Goossens, Dirk Lanneer, and Hugo De Man. Instruction set definition and instruction selection for ASIPs. In *Proceedings of the 7th International Symposium on High-Level Synthesis*, pages 11–16, Niagara-on-the-Lake, Ont., April 1994.

[19] Pan Yu and Tulika Mitra. Scalable custom instructions identification for instructionset extensible processors. In *Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*, pages 69–78, Washington, D.C., September 2004.