Fast, Quasi-Optimal, and Pipelined Instruction-Set Extensions

Ajay K. Verma AjayKumar.Verma@epfl.ch
Philip Brisk Philip.Brisk@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

Nowadays many customised embedded processors offer the possibility of speeding up an application by implementing it using Application-Specific Functional units (AFUs). However, the AFUs must satisfy certain constraints in terms of read and write ports between AFU and processor register file. Due to these restrictions the size and complexity of AFUs remain small. However, in recent some work has been done on relaxing the register file port constraints by serialising register file access (i.e., by allowing multi cycle read and write). This makes the problem of selecting best AFU significantly more complex. Most previous approaches use a two staged process to solve this problem, i.e., first selecting AFUs under some higher I/O constraints and then serialise them under the actual register file port constraints. Not only these methods are complex but also lead to suboptimal solutions. In this paper we formulate the AFU selection problem as an Integer Linear Programming and solve it optimally. We show experimentally that our methodology produces significantly better results compared to state of art techniques.

1. INTRODUCTION AND MOTIVATION

The availability of customised embedded processors has made the ISE (Instruction Set Extension) identification and AFU generation one of the most effective ways to improve the performance of the base processor. The problem of ISE identification has attracted the attention of many researchers in the last decade resulting in copious amount of publications. A typical approach for ISE identification is to start with a compiler's intermediate representation of an application (such as a dataflow graph), find the best ISE and create a new AFU to execute the ISE.

An AFU corresponds to a collection of instructions from the original application; however, not any collection of instructions can be a valid AFU. The corresponding AFU must satisfy some constraints such as bounded I/O ports between AFU and the register file. In other words, the problem of finding the best AFU is some kind of optimisation problem under various constraints. Most of the algorithms for ISE identification work by pruning the set of potential AFUs based on I/O and other constraints. Hence, for higher I/O constraints these approaches are ineffective.

Note that removing some of the constraints might improve the performance of selected AFU. Pozzi and Ienne showed in their work [9] a way to relax I/O constraints on the AFU by serialising the register file access. In other words, the actual AFU might have more number of inputs and outputs than available register file ports, however, while executing the AFU in hardware the access to register file is serialised in such a way that in each cycle the number of I/O access between AFU and register file is within limits.

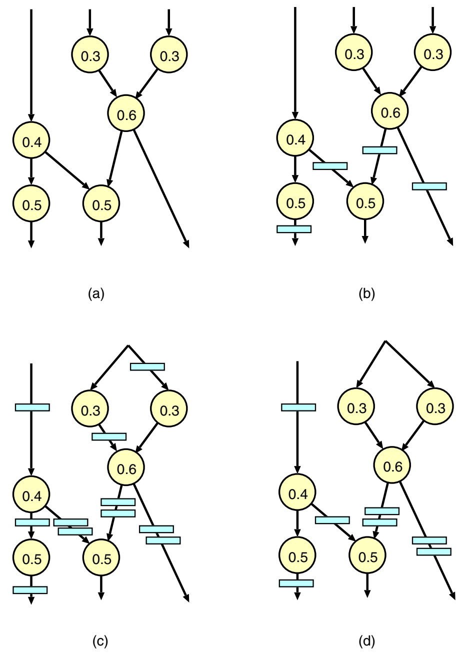

Figure 1: (a) An AFU with I/O constraints $(3,3)$, (b) Pipelining the AFU of part (a) under the I/O constraints $(3,3)$, (c) pipelining an AFU under the constraints $(2,1)$, and (d) the optimal pipelining of the same AFU under the I/O constraints $(2,1)$.

An example of serialising the I/O access is shown in Fig. 1. In Fig. 1(a) an AFU is shown which has 3 inputs and 3 outputs. If the actual number of I/O constraints is also $(3,3)$, then this AFU can be pipelined as shown in Fig. 1(b). All the three inputs are accessed at the beginning of the first cycle, and the four instructions with hardware latencies 0.3, 0.3, 0.4, and 0.6 are executed in the first cycle, and the remaining two instructions are executed in the second cycle. In other words, the whole AFU can be executed in two cycles in hardware.

Now consider another AFU shown in Fig. 1(c). Suppose the I/O constraints are $(2,1)$. In this case a possible pipelining of the AFU is shown in Fig. 1(c). As one can verify, in each cycle at most two inputs are read from register file and at most one output is written. The total number of cycles used in this implementation is 4, which is not the best possible execution time of the AFU. In fact, the optimal pipelining of this AFU under the I/O constraints $(2,1)$

is shown in Fig. 1(d), which makes the AFU execute in 3 cycles.

2. NONOPTIMALITY OF PRIOR ART

Pozzi and Ienne showed in their work [9] that pipelining the I/O access is extremely helpful in increasing the speedup of the application and might increase the speedup by 1.5x compared to nonpipelined AFUs. The algorithm used by Pozzi and Ienne first finds a set of best AFUs under some higher I/O constraints and then pipelines each of these AFUs under the actual I/O constraints and selects the one with maximum speedup. Although their method produces AFUs with significant speedup, it has certain drawbacks which make it suboptimal and in some cases impractical. In particular their approach suffers from the following problems:

- Since pruning-based ISE identification algorithms are impractical for higher I/O constraints, the Pozzi algorithm considers AFUs only up to a fixed I/O constraints. However, there is no guarantee that the optimal AFU will also have a bounded number of I/O. Normally the Pozzi algorithm considers the I/O constraints from $(2, 1)$ to $(10, 5)$. However, in the benchmark *aes* the optimal AFU turns out to have 22 inputs and 22 outputs, for instance. This is one of the main sources of suboptimality of their algorithm.

- Their algorithm also solves the problem using a two stage process. In the first stage they find the best AFUs under higher constraints and in the second stage they pipeline these AFUs under actual constraints and choose the best one. However, it might be possible that an AFU which was not optimal under higher I/O constraint, might become optimal after pipelining under the actual I/O constraints. An artificial example illustrating such a case is shown in Fig. 2. In Fig. 2, the software latency of each node is one cycle. The optimal AFU under the I/O constraint $(1, 1)$ by the original method is a path of 6 nodes where nodes in the path have hardware latencies 0.5 and 0.6 alternatively. One can notice that in the nonpipelined version this AFU saves 2 cycles; however, due to timing constraint the only possible way of pipelining this AFU requires 6 stages, which means that the pipelined version of AFU has zero speedup. On the other hand, if we consider the other AFU shown in the Fig. 2 consisting of two nodes, it saves one cycle in both pipelined and nonpipelined version. In other words, the optimal nonpipelined AFU might become suboptimal after pipelining. This is another reason why the Pozzi's algorithm is not optimal.

- The algorithm used in [9] for pipelining an AFU under actual constraint has an exponential complexity in the number of I/O. Since they consider only AFUs with I/O bounded by $(10, 5)$, this is not a problem. However, since the number of I/O of the best AFU is unbounded, this approach might become impractical. Also there is no proof of the optimality of their pipelining algorithm.

In this paper we present an approach based on Integer Linear Programming (ILP) to solve the same problem, and show that in some cases our approach produces better results compared to the algorithm proposed by Pozzi and Ienne. Also our approach terminates only in a few minutes, compared to their approach which takes several hours for bigger applications such as *aes*. The rest of the paper is organised as follows: Section 3 describes the related work on ISE identification problem. The following section formally defines the problem we want to solve. Next in Section 4 we rewrite the whole problem as an instance of ILP problem, which is solved using ILP solver CPLEX [5]. This is followed by experimental results in Section 6 and concluding remarks in Section 7.

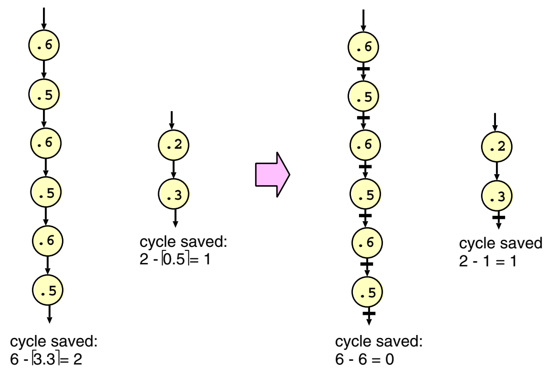

Figure 2: An example showing that the relative merit of AFUs might change after pipelining Assume that software latency of each node is 1.0**.**

3. STATE OF THE ART

The problem of ISE identification is not new and a wealth of literature exists on the topic. Most of the earlier work considers the problem as finding a cluster of instructions satisfying certain I/O constraints. Some of these approaches, such as [3, 6], consider finding clusters which have repeated occurrences in the application. However, these repeating clusters are usually small and do not gain significant speedup.

The work of other researchers such as [2, 8, 11, 4] show the importance of growing large clusters in order to achieve higher speedup. All these works consider all possible clusters of instructions and reject the ones which have either higher number of I/O's than allowed, contains some forbidden instructions such LOAD, STORE, JUMP etc., or the ones which create a self-loop. These three criteria are used effectively in pruning the search space and the algorithm mentioned in [8] can even handle the applications consisting more than a thousand nodes. Atasu *et al.* have proposed an approach [1] based on Integer Linear Programming to solve the same problem, which not only converges faster but can also be stopped during its execution to provide approximate solutions for larger applications.

Some approaches have also been proposed to relax some of the constraints on AFUs. The work of Pozzi and Ienne [9] suggested a method to relax I/O constraints as discussed in the earlier section. Some other researchers such as Nagaraju *et al.* have also presented some heuristics [7] for ISE generation under relaxed I/O constraints. However, their approach assumes that all the inputs and outputs of optimal AFU must be forbidden instructions, which is not always true. In a recent work Verma *et al.* [10] have presented an efficient heuristic to solve the same problem. However, their algorithm assumes that the underlying speedup model is monotonic, i.e., increasing the size of an AFU will never decrease the speedup. Although the assumption holds for a typical single-issue RISC processor, in general this assumption can be wrong. In this paper we extend the work of Pozzi and Ienne by formulating the whole problem as an instance of Integer Linear Programming and solve it using an efficient ILP solver.

4. PROBLEM FORMULATION

In this section we define the problem more formally. Each basic block can be represented as a directed acyclic graph (DAG) $G =$

(V, E), where the nodes correspond to instructions (such as ADD, MUL, etc.) and the edges correspond to data dependencies between the instructions. Since each basic block has certain inputs and outputs, we can construct a supergraph $G^+ = (V \cup V^+, E \cup E^+)$ of G. The additional nodes V^+ represent the inputs and outputs of the basic blocks and the additional edges E^+ connect V^+ to V and vice versa. Now onwards we will use G in place of G^+

Along with G, we are also given a subset F of V, known as forbidden nodes. The forbidden nodes correspond to instructions which cannot be the part of any AFU. LOAD, STORE, JUMP and other instructions which require the access to main memory may fall in this category. Also note that the cluster of nodes which will be selected as an AFU should not create a self-loop, i.e., the AFU should not have an instruction whose input is available only when the AFU has been executed. In other words, the cluster must be convex and any two nodes inside the cluster must have all paths between them inside the cluster.

For each node $u \in V$, we have two positive real values SW_u and HW_u which denote the latency of the corresponding instruction when implemented in software and hardware. Similarly, for a cluster S of the nodes, $SW(S)$ and $HW(S)$ denote the latencies when the cluster S is implemented in software and hardware respectively.

The task is to find a convex subgraph S of G, which does not contain any forbidden nodes and maximises the gain in terms of cycle count when implemented in hardware, i.e., maximises $SW(S) - HW(S)$. Note that both the functions $SW(S)$ and $HW(S)$ are processor-specific. As an example, for RISC processor $SW(S)$ can be approximated as:

$$SW(S) = \sum_{u \in S} SW_u.$$

On the other hand, for VLIW processor some of the instructions can be scheduled in parallel, and in that case the $SW(S)$ can be approximated as the critical path delay of S in software. The problem becomes even more complicated for superscalar processors that perform dynamic optimisations at runtime.

In contrast, $HW(S)$ depends on the specifics of AFU synthesis, and depends on the number of available I/O ports between AFU and register file. If the number of input ports is m and the number of output ports is n, then $HW(S)$ can be computed by pipelining the AFU S under the I/O constraints (m, n) [9]. In other words one needs to insert registers in the edges of the induced graph S^+ (the graph containing S, its inputs and outputs, a source node v_{src} connected to all input nodes and a sink node v_{sink} connected to all output nodes of S). The total latency of the DAG, once pipelined, is denoted by R. If we denote the number of registers on edge (u, v) by $\rho(u, v)$, then pipelining the DAG can be formulated as the following optimisation problem:

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

- **Pipelining:** *The circuit must operate at some given cycle time (let's say λ), i.e., for any path which has no registers on its edges the total hardware latencies of the nodes in the path must not exceed λ.*
- **Legality:** *All operands of a node must arrive at the same time. In other words, for any node $v \in S^+$ all paths from v_{src} to v must contain the same number of registers. We define $R - 1$ as the number of register on any path between v_{src} and v_{sink}.*
- **I/O Serialisation:** *At any cycle at most m inputs can be read from the register-file, and at most n outputs can be written at register-file. More formally for any $i \geq 0$ number of input nodes whose incoming edges have exactly i registers must not exceed m, and similarly all output nodes whose outgoing edges have i registers must not exceed n.*

The optimal value of R corresponds to $HW(S)$. Apart from finding the optimal subgraph one also needs to pipeline the subgraph optimally.

5. ILP FORMULATION

Integer Linear Programming (ILP) has been shown to be a very effective way to solve combinatorial problems such as the one mentioned above. Although solving an ILP is an NP-hard problem, there are efficient tools such as CPLEX [5] which can solve the sufficient large ILP's in reasonably small time.

Any instance of Integer Linear programming has two elements: constraints and objective function. The constraints are represented using linear inequalities and equalities, and the objective function must be a linear function of input variables. In CPLEX many kind of input variables are allowed such as real, integer, Boolean, piecewise continuous variable etc. Next we show how to describe the above problem in terms of an ILP instance. We give the formulation for a RISC processor model; however, any model can be chosen as long as $SW()$ and $HW()$ can be described in terms of linear equalities and inequalities.

Before describing the constraints and objective function, we describe the input variables and their interpretation which will help in understanding the ILP formulation. The list of the input variables is as follows:

- x_i**:** For each node n_i in the graph, we introduce a Boolean variable x_i, which is true if the corresponding node is in the chosen subgraph, and false otherwise.
- p_i, s_i**:** The two variables p_i and s_i are also Boolean variables corresponding to node n_i. p_i is set true if at least one of the predecessor nodes of n_i is in the chosen subgraph, and false otherwise. Similarly s_i is true if one of the successors of n_i is in the subgraph.
- ρ_{ij}**:** The variables ρ_{ij}'s are used for pipelining the chosen subgraph, and denote the number of registers on the edge connecting node n_i to n_j. Note that if there is no edge from n_i to n_j, then ρ_{ij} must be zero. Additionally, if none of the two nodes n_i and n_j are in the chosen subgraph, then also ρ_{ij} must be zero. This is because we only want to pipeline the subgraph corresponding to an AFU and not the whole DAG.
- A_i, B_i**:** The first of the two variables is an integer variable and denotes the first cycle in which all inputs to node n_i become available. Once again, note that, since we want to pipeline only the subgraph corresponding to the AFU, A_i for any node outside the AFU must be zero. On the other hand B_i is a real value and denotes the smallest time after the A_i-th cycle at which the output of n_i is ready.
- $c_{ik}^{IN}, c_{ik}^{OUT}$**:** c_{ik}^{IN} is also a Boolean variable and is true if the node n_i is an input of the chosen subgraph and is read after k cycles of execution. Similarly c_{ik}^{OUT} is a Boolean variable and is true if node n_i is an output of the optimal subgraph and is written back to the register file after k cycles of execution.

Next we describe the constraints on these variables. Since if-else, max, min, and, or etc. can be written in terms of linear constraints, we will describe our constraints in terms of these functions to improve readability.

No Forbidden Nodes: Since the optimal subgraph should not contain any of the forbidden nodes, the x_i corresponding to each forbidden node n_i must be zero, i.e.,

$$\forall n_i \in F, x_i = 0.$$

Convexity Constraint: Since the optimal subgraph should be convex, the values of x_i's cannot be chosen arbitrarily. In fact, if at least one of the predecessors as well one of the successors of a node are in the subgraph, then that node must be in the subgraph too. In other words,

$$p_i \text{ and } s_i \Rightarrow x_i,$$
$$\text{i.e., } p_i + s_i - x_i \leq 1.$$

Here the p_i (s_i) are the Boolean variables which are true if at least one of the predecessors (successors) of n_i is in the chosen subgraph. The p_i and s_i can be defined as follows:

$$p_i = \begin{cases} 0 & \text{if } n_i \text{ has no children.} \\ \cup(x_j \cup p_j) & \text{where } n_j's \text{ are children of } n_i. \end{cases}$$
$$s_i = \begin{cases} 0 & \text{if } n_i \text{ has no parents.} \\ \cup(x_j \cup s_j) & \text{where } n_j's \text{ are parents of } n_i. \end{cases}$$

In other words, a predecessor of n_i is present in the chosen subgraph if one of the two conditions are true: either one of the children of n_i is in the chosen subgraph, or the predecessor of a child of n_i is in the subgraph.

Only the Subgraph Should be Pipelined: We need to ensure that only the chosen subgraph is pipelined and not the whole DAG. In other words, the variables ρ_{ij}, A_i, and B_i corresponding to nodes and edges outside S^+ must be zero. More formally, if we choose M to be a large integer (at least as large as N), then

$$\begin{aligned} \rho_{ij} &\leq M(x_i + x_j), \\ A_i &\leq Mx_i, \\ B_i &\leq Mx_i, \text{ and} \\ B_i &\leq \lambda. \end{aligned}$$

where λ is the cycle time. The last constraint implies that an instruction, executing in a cycle should be finished before the cycle ends.

All Inputs of a Node Must Arrive on the Same Cycle: This constraint is the same as the second constraint of the optimisation problem of Section 4, which says that all inputs of a node in the subgraph must arrive on the same cycle. Since A_i is the first cycle at which all inputs of node n_i are ready, if some inputs arrive before that cycle, one needs to delay them by inserting registers on their outgoing edges to the node n_i. Formally we can write

$$\begin{aligned} &\text{if } (x_i = 1 \text{ and } n_j \text{ is a children of } n_i) \\ &\quad \text{then } A_i = A_j + \rho_{ji}. \end{aligned}$$

Also, according to the second constraint, all paths from v_{src} to v_{sink} must have exactly $R - 1$ registers. Since in our ILP formulation we do not have v_{src} and v_{sink}, we have to define these constraints in terms of A_i of the output node and ρ_{ij} of the output edges.

$$\begin{aligned} &\text{if } (x_i = 1 \text{ and } x_j = 0 \text{ and } n_i \text{ is a child of } n_j) \\ &\quad \text{then } R - 1 = A_i + \rho_{ij}. \end{aligned}$$

The above constraint means that if n_i is an output of the chosen subgraph (which is only true if n_i is inside the subgraph and one of its parents is outside the subgraph), then the output of n_i must be delayed until $(R-1)$-th cycles by inserting edges on its outgoing edge.

Relations between B_i Values of a Node and its Children: We have defined earlier that B_i for a node is the time after A_i-th cycle by which the output of n_i is ready. Suppose n_j is a child of n_i. Now there can be two cases:

- either there will be some register on the edge between n_i and n_j, or
- there will be no registers on the edge between n_i and n_j.

At this point we define a new variable m_{ji} such that m_{ji} is zero if there is at least one register on the edge between n_i and n_j; otherwise we set the value of m_{ji} as B_j. It is easy to see that all operands of instruction corresponding to node n_i will be ready by the time $A_i + \max_j(m_{ji})$. In other words, B_i will be the same as $HW_i + \max_j(m_{ji})$. More formally we can write these constraints as follows:

$$\begin{aligned} &\text{if } (x_i = 1 \text{ and } \rho_{ji} \geq 1 \text{ and } n_j \text{ is a child of } n_i) \\ &\quad \text{then } m_{ji} = 0, \\ &\text{if } (x_i = 1 \text{ and } \rho_{ji} = 0 \text{ and } n_j \text{ is a child of } n_i) \\ &\quad \text{then } m_{ji} = B_j, \text{ and} \\ &B_i = HW_i + \max_j (m_{ji}). \end{aligned}$$

The first two constraints define the value of m_{ji}, while the last constraint defines the value of B_i using m_{ji}'s.

Definition of c_{ik}^{IN} and c_{ik}^{OUT}: Note that c_{ik}^{IN} is set true only for those nodes which are inputs to the subgraph (i.e., the node should not be in the chosen subgraph, but one of its parents must be in the subgraph). In order to find out the cycle at which this input is read, one needs to compute the minimum number of registers on those outgoing edges of this node which enter the chosen subgraph because that is the first time when the value of this node was accessed. In other words, c_{ik}^{IN} is true if x_i is false (i.e., the node is outside the chosen subgraph), at least one of the parents of this node is in the chosen subgraph, and the minimum of all ρ_{ij} (where x_j is true) is k. Formally we can write these constraints like this:

$$\begin{aligned} &\text{if } (x_i = 0, x_{j_1} = \cdots = x_{j_r} = 1 \text{ and} \\ &\quad n_{j_1}, n_{j_2}, \ldots, n_{j_r} \text{ are parents of } n_i) \\ &\quad \text{then } I_i = \min(\rho_{ij_1}, \rho_{ij_2}, \ldots, \rho_{ij_r}). \end{aligned}$$

Here I_i is an integer variable denoting the cycle at which n_i was accessed for the first time. Now if this cycle is equal to k, then c_{ik}^{IN} will be set to 1, and 0 otherwise. In other words,

$$\text{if } (I_i = k) \text{ then } c_{ik}^{IN} = 1.$$

On the other hand, for c_{ik}^{OUT} we only need to check whether n_i is an output node of the subgraph; if it is, then the variable will be true if its outgoing edge has exactly k registers. In other words,

$$\begin{aligned} &\text{if } (x_i = 1 \text{ and } x_j = 0 \text{ and } \rho_{ij} = k \text{ and } n_i \text{ is a child of } n_j) \\ &\quad \text{then } c_{ik}^{OUT} = 1. \end{aligned}$$

Register-Port Constraints: Due to register-port constraints we need to ensure that in each cycle at most m variable are read from register file, and at most n values are written on register-file. This means that for any k the sum of all c_{ik}^{IN} must not exceed m, and the sum of all c_{ik}^{OUT} must not exceed n:

$$\sum_i c_{ik}^{IN} \leq m \text{ and } \sum_i c_{ik}^{OUT} \leq n.$$

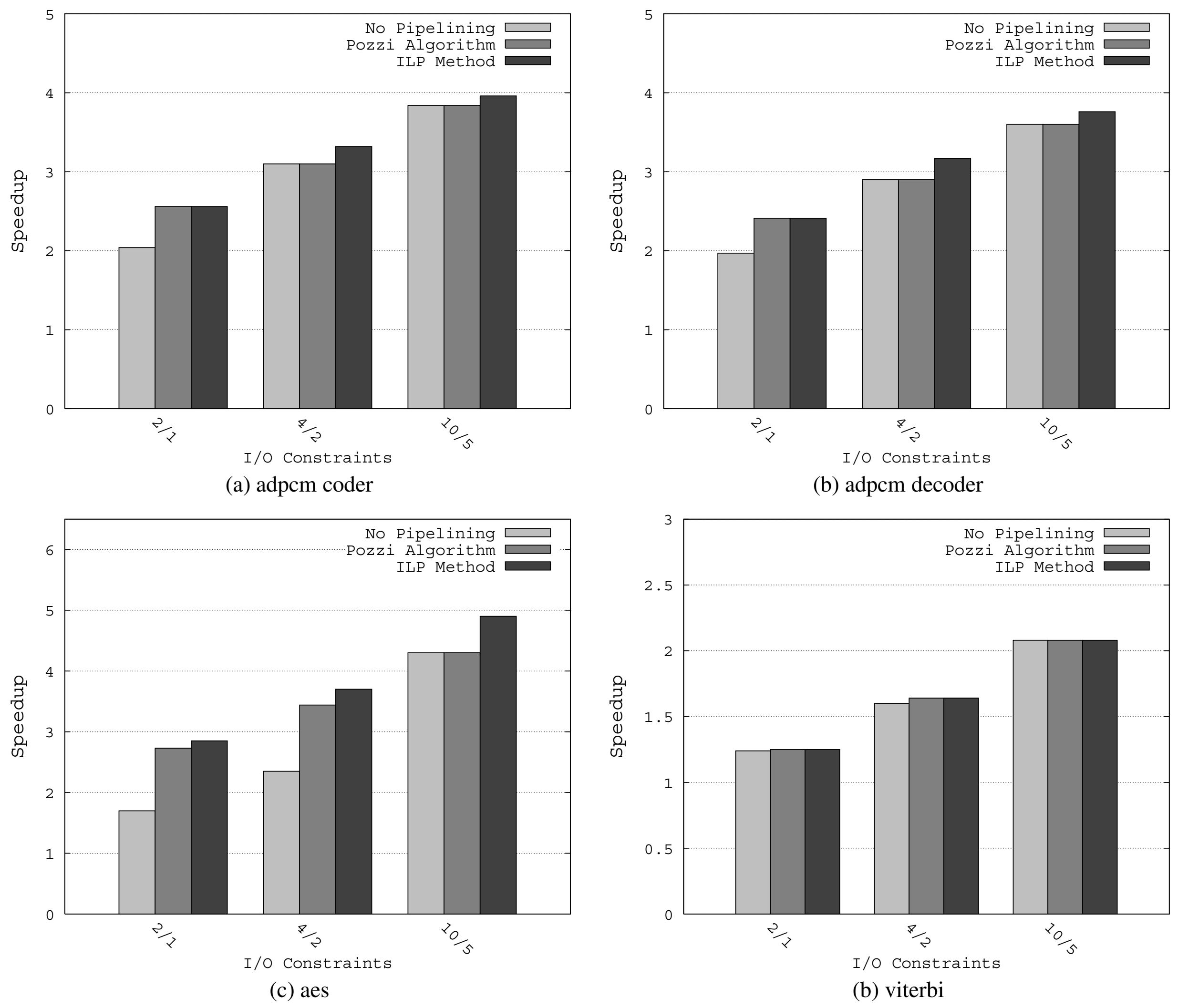

Figure 3: Comparison of the speedup values of AFUs generated by our algorithm with the state of art techniques.

Since the number of inputs and outputs of the optimal AFU can be arbitrarily large, the number of registers on incoming edges to the AFU as well as the number of registers on the outgoing edges of the AFU can be extremely high. In other words c_{ik}^{IN} and c_{ik}^{OUT} can be true for significantly large values of k. At this point one might think that we should define all variables c_{ik}^{IN} and c_{ik}^{OUT} for all values of k from 0 to $\mid V \mid -1$. However, we can trivially bound k for c_{ik}^{IN} to $\frac{|V|}{m}$ and $\frac{|V|}{m}$ for c_{ik}^{OUT} by assuming that even if all nodes of the DAG are inputs to the chosen subgraph, then also maximum number of registers on incoming edges of the subgraph can be $\frac{|V|}{m}$ (and similarly for outgoing edges). In practice, the maximum number of registers on incoming and outgoing edges of optimal AFU turns out to be much smaller than these bounds; by choosing a smaller value of k the convergence of ILP can be made significantly faster.

The second element of each ILP is the objective function. In our case we want to maximise the savings in terms of cycles. Hence, the objective function will be to maximise $SW(S) - HW(S)$. In other words,

$$\text{maximise } \sum_i SW_{n_i} x_i - R.$$

Note that the above objective function assumes that the underlying processor is a RISC processor (that is why $SW(S)$ is written as $\sum_i SW_{n_i} x_i$). This can be generalised to any processor provided that $SW(S)$ can be approximated by solving a system of linear constraints.

6. EXPERIMENTAL RESULTS

We have implemented our algorithm in C++; it takes the description of an application in the form of a graph and outputs the ILP instance whose solution corresponds to the best AFU of the application. The instance of the ILP is solved by the ILP solver CPLEX. In order to measure the relative performance of AFUs we use the RISC speedup model. The software latency of an instruction is estimated as the latency of corresponding instruction in the execution stage. However, the hardware latency of an instruction is estimated by synthesising the corresponding operator on a common standard cell library for UMC $0.18\mu m$ CMOS technology and normalise to the delay of 32-bit MAC (multiply-accumulate) operation.

We ran our algorithm on four applications namely *adpcmcoder*, *adpcmdecoder*, *viterbi*, and *aes*. For each benchmark we consider three sets of I/O constraints: $(2, 1)$, $(4, 2)$ and $(10, 5)$. For each

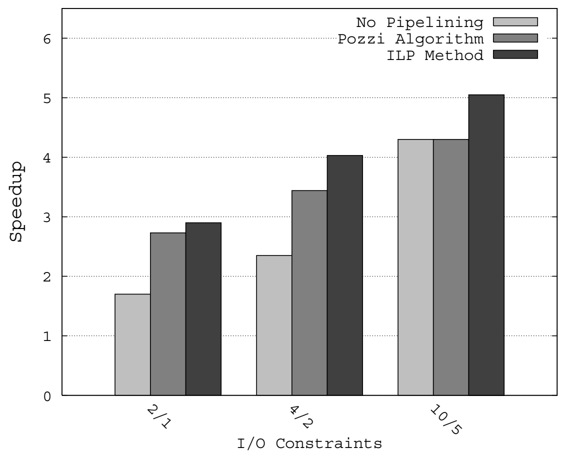

Figure 4: The speedups obtained for *aes* benchmark, when the ILP was run for an hour.

application we consider a maximum of 16 AFUs. For selecting multiple AFUs we use an iterative method similar to the one presented in [8]. In this method the AFUs are selected using a greedy approach one by one, and after selecting each AFU, the selected nodes are set as forbidden nodes to find the next AFU. We compare our results with Atasu *et al.* [8] where no pipelining is done, as well as with the work of Pozzi and Ienne [9] which considers pipelining.

We can see in Fig. 3 that for all benchmarks pipelined version has higher speedup than the nonpipelined version. Also note that the difference between the speedup for nonpipelined and pipelined version reduces as the I/O constraints grow. This is because in nonpipelined version for smaller I/O constraints many potential AFUs which although give better speedup cannot be selected due to I/O constraints, while in pipelined version they are selected irrespective of their actual number of inputs and outputs. However, when we increase the I/O constraints the nonpipelined version can select these AFUs with a higher number of I/O's and reduces the gap between speedups obtained by pipelined and nonpipelined version. Also, we can see that in all cases the ILP approach produces better results compared to the Pozzi and Ienne algorithm. The reason for this gain is already explained in Section 2.

Apart from the speedup numbers it is also important to note the execution time of the algorithm. In all our benchmarks we run the ILP solver for 3 minutes. If the solver does not converge in that much time, we take the AFU found in that much time. Except the *aes* in all other benchmarks the ILP solver converges within 180 seconds. However, in case of *aes* it produces the result shown in Fig. 3 in the alloted time. If we allow more time we get only slightly better results in case of *aes*: Fig. 4 shows the speedups obtained for *aes* when the ILP solver was run for an hour. On the other hand, the execution time of Pozzi and Ienne's algorithm varies from minutes to hours. For *aes* it takes several hours and produces inferior results than the ILP method. On other benchmarks it takes several minutes.

7. CONCLUSIONS

In this paper we have shown the suboptimality of previous approaches for ISE identifications. Apart from optimality issues, the state of art techniques also suffer from high runtime complexity, which makes them impractical for larger applications. We have presented a new method for ISE identification based on ILP. Our approach not only improves the speedups compared to the state of art techniques, but also reduces the execution time. The additional advantage of using an ILP solver is that for larger applications the execution can be terminated at any time, and the solver produces the best results found so far, often quite close to the optimal.

Our method is also generalisable to any processor model as long as the software and hardware latencies of an AFU can be modelled in terms of linear constraints, which is true for most processors.

8. REFERENCES

[1] K. Atasu, G. Dundar, and C. C. Ozturan. An integer linear programming approach for identifying instruction-set-extensions. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis*, pages 172–77, 2005.

[2] K. Atasu, L. Pozzi, and P. 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.

[3] P. Brisk, A. Kaplan, R. Kastner, and M. 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, Oct. 2002.

[4] N. Clark, H. Zhong, and S. Mahlke. Processor acceleration through automated instruction set customisation. In *Proceedings of the 36th Annual International Symposium on Microarchitecture*, pages 129–40, San Diego, Calif., Dec. 2003.

[5] ILOG. *CPLEX Optimization Engine*, 2006. Version 10.0.

[6] R. Kastner, A. Kaplan, S. Ogrenci Memik, and E. Bozorgzadeh. Instruction generation for hybrid reconfigurable systems. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 7(4):605–27, Oct. 2002.

[7] N. Pothineni, A. Kumar, and K. Paul. Application apecific datapath extension with distributed I/O functional units. In *Proceedings of the 20th International Conference on VLSI Design*, 2007.

[8] L. Pozzi, K. Atasu, and P. Ienne. Exact and approximate algorithms for the extension of embedded processor instruction sets. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, CAD-25(7):1209–29, July 2006.

[9] L. Pozzi and P. Ienne. Exploiting pipelining to relax register-file port constraints of instruction-set extensions. In *Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*, pages 2–10, San Francisco, Calif., Sept. 2005.

[10] A. K. Verma, P. Brisk, and P. Ienne. Rethinking custom ISE identification: A new processor-agnostic method. In *Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*, pages 125–34, Salzburg, Austria, Sept. 2007.

[11] P. Yu and T. Mitra. Scalable custom instructions identification for instruction set extensible processors. In *Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*, pages 69–78, Washington, D.C., Sept. 2004.