Rethinking Custom ISE Identification: A New Processor-Agnostic Method

Ajay K. Verma
ajaykumar.verma@epfl.ch

Philip Brisk
philip.brisk@epfl.ch

Paolo Ienne
paolo.ienne@epfl.ch

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

ABSTRACT

The last decade has witnessed the emergence of the Application Specific Instruction-set Processor (ASIP) as a viable platform for embedded systems. Extensible ASIPs allow the user to augment a base processor with Instruction Set Extensions (ISEs) that execute on Application Specific Functional Units (AFUs) – dedicated hardware that executes the ISEs. Due to the limited number of read and write ports in the register file of the base processor, the size and complexity of AFUs are generally limited. Recent work has focused on overcoming these constraints by serialising access to the register file. Apart from these complications, the primary challenge in the identification and selection of the best AFU is the modelling of AFU performance in the context of different base processors: once the base processor changes, the ISE identification and AFU selection process must be redone from scratch. Exhaustive ISE/AFU enumeration methods are not scalable and generally fail for larger applications. To address this concern, a new approach to ISE/AFU identification is proposed. In particular, we show that the speedup model of ISEs/AFUs is independent of the specific details of the base processor, under fairly reasonable assumptions. The approach presented here significantly prunes the list of best ISE/AFU candidates compared to previous approaches. Experimentally, we observe the new approach produces optimal results on larger applications where prior approaches either fail or produce inferior results.

Categories and Subject Descriptors

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

General Terms

Algorithms, Performance, Design

Keywords

Custom Processors, ISE Identification, Maximal Cluster

CASES'07, September 30–October 3, 2007, Salzburg, Austria.

1. INTRODUCTION AND GOAL

ISE identification and AFU generation are the primary methods by which automated tools accelerate the performance of extensible ASIPs. To date, a wealth of literature [1, 3, 4] has been published on this topic. Typical approaches identify and extract ideal ISE candidates from a compiler′s intermediate representation of an application; the best ISE is then identified using a cost function, and an AFU is generated to execute the ISE. Although prior techniques have proven quite successful, there is still considerable room for improvement. This paper advances the state of the art in ISE identification and AFU generation in several respects.

First and foremost, we formally prove that the optimal ISE can be identified using a speedup model that is independent of the specific details of the execution pipeline of the base processor, under assumptions that are generally reasonable for current extensible ASIPs on the market. Based on these assumptions, a sparse set of ISE candidates is generated, out of which the best one can be chosen.

The first assumption is that the base processor must be a RISC. As a counter-example, the proof does not hold for superscalar processors that perform dynamic optimisations in hardware such as out-of-order speculative execution and register renaming. The second assumption is that the cost of executing an ISE on an AFU is never slower than executing the same operations in software. This assumption is generally true, except for hybrid systems where the base processor and AFUs are synthesised on different technologies. An example of a hybrid system is the Xilinx Virtex-2 Pro FPGA, where the base processor is an IBM PowerPC, but the ISEs are synthesised on the general logic of an FPGA. An individual addition or multiplication operation, for example, is likely to be slower on the general logic of an FPGA than on the highly optimised ALU of the FPGA. This dichotomy remains true even if dedicated multiplier and DSP blocks are utilised. As reported by Kuon and Rose [2], mismatches in terms of bitwidth (e.g., performing 5×5 bit multiplication on a 9×9 bit multiplier) and the cost of routing data to and from the dedicated blocks can severely impede performance. In a pure CMOS system, in contrast, the generated AFUs will always have the correct bitwidth and delays, and hence the cost of executing an AFU will be less than executing the same operations in software.

A second contribution is a reevaluation of the assumptions underlying the processor of ISE generation. Previous work [5] assumes that each AFU receives all operands at once, and produces all results in exactly one clock cycle. Under this model, the number of inputs and outputs of each ISE cannot exceed the number of read and write ports of the register file in the base processor. In 2005, Pozzi and Ienne [6] developed a technique for multi-cycle pipelined

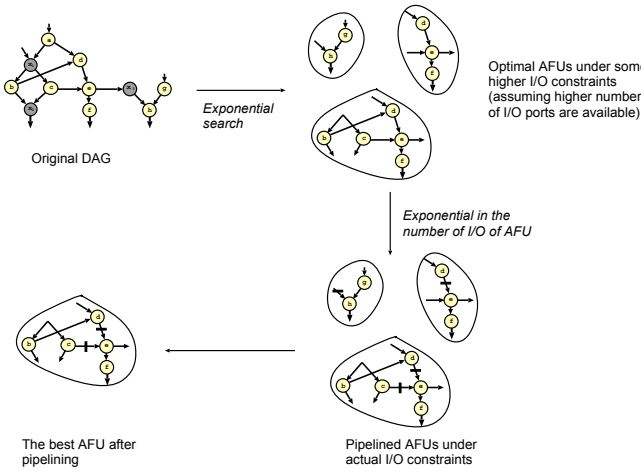


Figure 1: An overall description of the previous approach for ISE identification.

ISE execution; during each cycle, data is read from and written to the base processor′s register file. Under this execution model, there is no need to limit the number of inputs and outputs of each custom instruction. Unfortunately, this eliminates the most effective pruning criteria that allowed previous ISE enumeration methods to converge.

Prior techniques for ISE generation exhaustively enumerate the subgraphs of the compiler′s intermediate representation, implicitly rejecting all of those that do not meet the I/O constraints of the register file on the base processor; however, this pruning criteria was the key to fast convergence of an otherwise exponential worst-case method. It is already well-known that increasing the number of read and write ports on the base processor′s register file significantly increases the runtime of ISE enumeration [5], due to the decreasing effectiveness of pruning. To get a significant speedup for multi-cycle ISEs, Pozzi and Ienne [6] repeatedly ran traditional ISE enumeration methods, increasing the number of read and write ports each time.

The ISE identification method presented here, in contrast, does not require I/O constraints to effectively prune the search space, and must only run once. Since there are no I/O constraints, the enumeration method can generally choose the largest subgraphs possible (in the most general case, the entire graph itself). In fact, the only constraining factors are forbidden nodes, which are operations in the compiler′s intermediate representation that must be executed in software, and the requirement that all enumerated subgraphs be convex. The approach to ISE identification presented in this paper generates a potentially optimal set of ISEs that are convex and contain no forbidden nodes. This set is then pruned using theoretical properties that will be derived later in the paper.

A third contribution is a faster algorithm for pipelining compared to the one proposed by Pozzi and Ienne [6]. This, in turn, enables a fourth contribution. Due to the high runtime cost of pipelining, Pozzi and Ienne used a single-cycle speedup model to identify the best custom instruction, and then pipeline it. There is no guarantee, however, that this custom instruction will still be the best once it has been pipelined. Due to the more efficient algorithm for pipelining, it becomes possible to pipeline each of the potentially optimal ISEs, so the best candidate can be selected after pipelining.

The previous approach to ISE generation is shown in Fig. 1. As discussed above, this approach is problematic for several reasons:

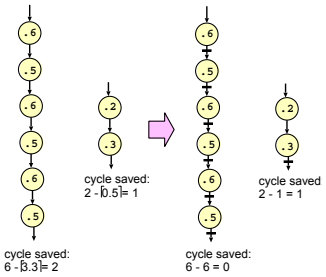


Figure 2: An example showing that the relative merit of AFUs might change after pipelining Assume that software latency of each node is 1.0**.**

- Nonoptimal ISEs should also be considered, because these ISEs may become optimal after pipelining. Fig. 2 shows an example where this occurs. In Fig. 2, the software latency of each node is 1.0 cycles, and the timing constraint requires that each stage in the pipeline have a period of no more than 1.0 cycles. The optimal ISE, using the original method, is a path containing 6 nodes, where nodes have alternating delays of 0.5 and 0.6 cycles. The suboptimal ISE has 2 nodes, of delays 0.2 and 0.3 cycles. Due to the timing constraint, the only possible way to pipeline the optimal ISE is to create an AFU with 6 pipeline stages and a total latency of 6 cycles: the same as software execution. In the suboptimal ISE, both nodes can be collapsed into a single-cycle AFU, compared to a 2-cycle latency in software. Consequently, after taking pipelining into account, the optimal ISE, based on previous methods, yields a suboptimal AFU that offers no speedup. Based on this example, the problem of ISE generation under I/O constraints would appear to reduce to the problem of enumerating every feasible subgraph as a potential ISE, and then pipelining each candidate to measure the latency of its AFU.

- The pipelining algorithm taken by Pozzi and Ienne [6] is to repeat the enumeration process for all I/O constraints from $(2, 1)$ to (N, N), where N is the number of nodes in the DAG. Due to runtime considerations, the maximal size of the search is stopped at $(10, 5)$. The runtime increases exponentially due to the ineffectiveness of pruning as I/O constraints increase. Furthermore, the algorithm used for pipelining is exponential in the number of input and output nodes of the DAG. Clearly, this approach cannot scale as the number of nodes in the DAG increases.

- If the software/hardware latency of some instruction changes, the result of the ISE enumeration and AFU generation algorithm is no longer guaranteed to be optimal. For example, a custom instruction that is optimal for a speedup model based on 0.18 micron CMOS technology may become suboptimal for 0.13 micron CMOS technology. One must either repeat the entire process or accept a potentially suboptimal solution. The approach presented here, in contrast, finds a sparse set of potential ISEs. Irrespective of the specific technology used, the optimal ISE is always included in this set.

Fig. 3 illustrates the new approach for ISE identification. ISE generation proceeds without I/O constraints, based on the knowledge the pipelining will appropriately serialise access to the register file. The algorithm has 6 main steps:

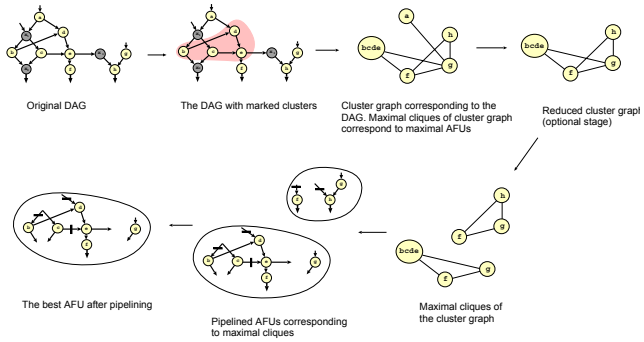


Figure 3: An overall description of our approach.

Step 1, Clustering—In this step, nodes are grouped into a set of equivalent classes based on assumptions regarding the speedup model, which will be discussed in Section 4. If nodes x and y belong to the same equivalence class C, then any ISE that includes x can also include y without losing convexity. Consequently, any ISE that includes x but not y is suboptimal, because y could always be added to it—and our equivalence model assumes that adding y to the ISE cannot negatively affect the speedup obtained.

Step 2, Cluster Graph Construction—In the compiler's intermediate representation, each basic block is a DAG, where vertices represent operations and edges represent data dependencies. Each equivalence class identified in Step 1 is then compressed into a single vertex in the DAG. An edge is placed between every pair of equivalence classes that could be merged into the same ISE without violating convexity constraints. The result graph, which is undirected, is called a *Cluster Graph*. Compared to the original DAG, the cluster graph has significantly fewer vertices and edges, which reduces the overall size of the search space for ISE identification.

Step 3, Pruning—This step is optional. Pruning techniques are used to identify equivalence classes that provably cannot be part of an optimal ISE; these classes are then removed from the cluster graph, further reducing the size of the search space.

Step 4, Clique Enumeration—This step enumerates every maximal clique in the cluster graph, which corresponds to a potentially optimal custom instruction. The clique enumeration process is an exhaustive search, which has an exponential worst-case running time. In practice, however, it is significantly faster than the subgraph enumeration method used by prior techniques [5, 4].

Step 5, Clique Pruning—This step is optional and depends on the specifics of the speedup model. We use certain properties of the speedup model to bound the speedup of an AFU. Based on these bounds we can remove some of the nodes and edges from the cluster graph (when the maximal cluster containing a node or edge is guaranteed to provide minuscule speedup).

Step 6, Pipelining—The final step is to pipeline the ISEs corresponding to each of the remaining cliques. The ISE that offers the maximal speedup is then selected.

To select multiple ISEs from the same DAG, the process can be repeated. The optimal ISEs from previous iterations are marked as forbidden, to prevent overlapping ISEs. The process can terminate as desired by the user. Possible stopping criteria include: a fixed allowable number of ISEs, the aggregate area of the set of selected ISEs cannot exceed some fixed value, or all operations in the DAG are marked as forbidden.

The remainder of the paper is organised as follows. Section 2 discusses related work in the area of ISE generation. Section 3 formally defines the ISE generation and pipelining problems. Section 4 describes our approach to solve the problem, including formal proofs regarding the speedup model, as well as algorithms for ISE enumeration and pipelining. Section 5 presents an experimental evaluation of the proposed approach, and compares it to prior work. Section 6 concludes the paper.

2. RELATED WORK

The vast majority of prior techniques for ISE generation use the number of input and output ports between AFU and register file to constrain the set of subgraphs that can be enumerated [4]; however, the pipelining algorithm of Pozzi and Ienne [6] eliminates the need for these constraints.

To date, one algorithm for ISE generation has been developed by Pothineni *et al.* [7] with pipelining in mind. Their work is similar to this paper with respect to the assumptions regarding the monotonicity of convex subgraphs; however, their algorithm considers only connected subgraphs as potential AFUs. They use an unconstrained MaxMIMO algorithm to enumerate ISEs. Afterwards, they pipeline the MaxMIMOs using the algorithm of Pozzi and Ienne. Rather than searching for the best ISE, they find an overlapping set of ISE candidates and then select a nonoverlapping subset using an algorithm similar in principle to Guo *et al.* [8]. Pothenini *et al.* do not describe any techniques that are comparable to our methods for clustering and clique enumeration.

The cluster graph described in this paper has some similarities to the All Pairs Common Slack Graph (APCSG) described by Brisk *et al.* [9]. The difference is that APCSG edges are placed between operations that can be scheduled in parallel. Their approach is not optimal and focuses primarily on finding VLIW-style parallel instructions.

Other ISE generation techniques have discarded I/O constraints for reasons unrelated to pipelining. Kastner *et al.* [10], for example, use a similar method to find ideal IP blocks to integrate into a reconfigurable fabric. Likewise, Cadambi and Goldstein [11] use similar methods to build a macro-generator library for FPGAs. These techniques do not extend a base processor with constraints on the register file.

3. PROBLEM STATEMENT

Each basic block can be represented as a DAG $G = (V, E)$ where nodes correspond to primitive operations (e.g. ADD, MUL, LOAD) and edges correspond to data dependencies between operations. We can extend G into a larger DAG, $G^+ = (V \cup V^+, E \cup E^+)$, where V^+ is the set of inputs and outputs of the basic block, and E^+ is the set of edges connecting vertices in V and V^+. To simplify notation, we will henceforth use G in place of G^+.

Along with G, we are given a subset $F \subseteq V$ of forbidden nodes that cannot be included in any ISE. Initially, forbidden nodes correspond to operations such as LOAD, STORE, and JUMP, which require access to main memory. If the ISE generation algorithm is run multiple times to find multiple ISEs in the same DAG, then already-identified ISEs are also marked as forbidden nodes.

For each node $u \in V$, there are two positive real values SW_u and HW_u, which are the latencies of u implemented in software (on the base processor) and hardware. A convex subgraph $S \subseteq V$ is a subset of nodes, such that for every pair of nodes $u, v \in S$, every path from u to v in G consists solely of nodes in S. For a convex subgraph S, $SW(S)$ and $HW(S)$ are the latencies of S when implemented in hardware and software respectively.

The total number of cycles saved by implementing S as an ISE is $M(S) = SW(S) - HW(S)$. The ISE generation problem is to find a convex subgraph S of G that contains no forbidden nodes

(i.e., $S \cap F$ is empty) and maximises $M(S)$. In general, both $SW()$ and $HW()$ are process-specific functions. For example, in a RISC processor, $SW(S)$ can be approximated by adding the software latencies of all nodes in S, e.g.,

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

Throughout this paper, we assume that the base processor is a RISC.

$HW(S)$, in contrast, is dependent on specific issues of AFU synthesis (algorithms, ASIC vs. FPGA, etc.). Clearly, $HW(S)$ depends on the available I/O ports between the AFU that implements S as an ISE and the register file in the base processor. If m and n are the number of input and output ports of the register file, then $HW(S)$ can be computed by pipelining the AFU for S under I/O constraints (m, n) [6].

Pipelining, itself, is also a complicated problem. Let $G_S = (S, E_S)$ be the subgraph of G induced by S. We extend G_S with two additional nodes, v_{src} and v_{sink}, which are connected to all of the inputs and outputs of S respectively by edge sets E_{src} and E_{sink}. Let $S^+ = S \cup v_{src} \cup v_{sink}$ and $E_S^+ = E_S \cup E_{src} \cup E_{sink}$. Pipelining is then applied to the resulting DAG $G_S^+ = (S^+, E_S^+)$. Let R denote the total latency of the G_S^+ after pipelining, which is achieved by inserting registers on the edge set E_S^+. Let $\rho(u, v)$ denote the number of registers inserted onto edge (u, v).

Given these definitions, pipelining can be formulated as an optimisation problem [6]:

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

- **Pipelining:** *The maximum register-to-register latency of the circuit cannot exceed λ, a user-specified value. For any path p through G_S^+ whose edges contain no registers, the sum of the hardware latencies of the operations in p cannot 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. The number of registers on any path between v_{src} and v_{sink} will be $R - 1$ (yielding R pipeline stages).*
- **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 back. Formally, let $S_{in}(i)$ be the set of input nodes whose incoming edges have exactly i registers, meaning that these nodes read their values from the register file at the i^{th} clock cycle. Likewise, let $S_{out}(j)$ be the set of output nodes whose outgoing edges have exactly j registers. Then $\mid S_{in}(i) \mid \leq m$ and $\mid S_{out}(j) \mid \leq n$.*

The optimal value of R corresponds to $HW(S)$. Consequently, one must solve the pipelining problem optimally in order to solve the ISE generation problem optimally as well.

4. ALGORITHMS FOR ISE GENERATION AND PIPELINING

Fig. 3 provided an overview of the new approach to ISE generation. This section describes the approach in greater detail, while focusing on its advantages over previous approaches. It is important to note that if there are no inherent assumptions regarding the performance model of the base processor and AFUs, then it is impossible to compare the merits of two different ISEs, S_1 and S_2, without comparing their speedups, i.e., computing $M(S_1) - M(S_2)$. Without any assumptions, the only feasible approach for ISE generation is to enumerate all convex subgraphs, compute their speedups, and choose the best one.

It is important to recognise, however, that the speedup model is not a random function, and that it is certainly reasonable to make assumptions about it, as long as the properties used to justify the assumptions are relatively safe. The algorithms for ISE generation presented here exploit these properties to reduce the size of the search space significantly.

4.1 Monotonicity of Speedup Model

The most important property of the speedup model is *Monotonicity*, and is defined as follows.

DEFINITION 1. *A speedup model is monotonic, if for any two convex subgraphs S_1 and S_2,*

$$(S_1 \subseteq S_2) \Rightarrow M(S_1) \leq M(S_2).$$

Theorem 1, which follows, shows that the speedup model for RISC processors, under several fairly weak assumptions, is monotonic; moreover, its monotonicity is independent of the hardware and software latencies of operations.

THEOREM 1. *The speedup model for ISE generation for RISC processor is monotonic, under the assumption that for any convex subgraph S, $SW(S) \geq HW(S)$.*

PROOF. Let S_1 and S_2 be convex subgraphs of G, and assume that S_2 is a supergraph of S_1. Since both S_1 and S_2 are convex, S_2 can be obtained from S_1 by a sequence of the following three operations:

- Choose a convex subgraph T, such that $T \cap S_1$ is empty and there are no paths from any node in S_1 to a node in T, and vice versa. Let $S_1' = S_1 \cup T$.
- Choose a node v from outside S_1. Let $P(v, S_1)$ be the subgraph of G induced by the set of nodes on all paths from v to nodes in S_1, and let $S_1' = S_1 \cup P(v, S_1)$.
- Choose a node v from outside S_1. Let $P(S_1, v)$ be the subgraph of G induced by the set of nodes on all paths from nodes in S_1 to v, and let $S_1' = S_1 \cup P(S_1, v)$.

Since S_1 is a convex subgraph, S_1' must also be convex if it is constructed using these three operations. Given S_1 and S_2, as described above, we can construct S_2 from S_1 by repeatedly applying these three operations, yielding the following sequence of subgraphs:

$$S_1 = S^{(0)} \subset S^{(1)} \subset \cdots \subset S^{(k)} = S_2.$$

$S^{(i+1)}$ is generated from $S^{(i)}$ by applying a transformation.

To prove the correctness of the theorem, we must prove that none of these transformations reduces the speedup of resulting subgraph.

Consider the first operation, where $SW(S_1)$ and $SW(T)$ be the software latencies of AFUs corresponding to subgraphs S_1 and T. After pipelining, let R_1 and R_T be the number of registers inserted between v_{src} and v_{sink} for S_1 and T respectively. Now we must pipeline S_1'.

One possibility would be to add R_T additional registers on each incoming edge of inputs to S_1 and R_1 additional registers on each outgoing edge of outputs of T, as shown in Fig. 4. We argue that no more than m input nodes of S_1' have the same number of registers on their incoming edges. This constraint is already satisfied among the input nodes of S_1 and T individually. Since $HW(T) \leq R_T$,

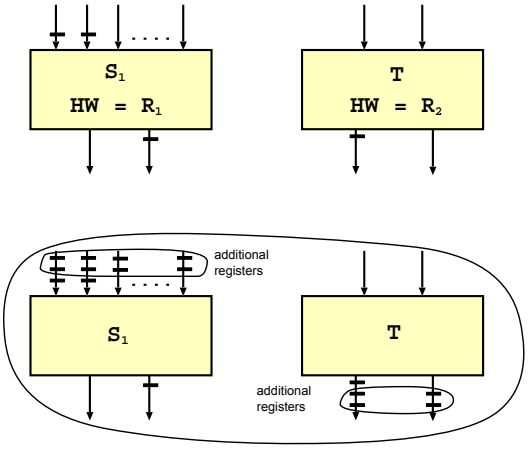


Figure 4: Disjoint union of two convex subgraphs increases the speedup.

each input of T will have fewer than R_T registers on its incoming edges. Each input of S_1 has at least R_T registers on its incoming edges. This means that no input of S_1 can have the same number of registers on its incoming edge as any input of T. This ensures that the input constraint is satisfied for S_1'. Likewise, the analogous argument holds for the outputs of S_1' as well.

For S_1', the number of registers on every path from v_{src} to v_{sink} is $R_1 + R_2$. This ensures that $M(S_1') \geq SW(S_1) + SW(T) - (R_1 + R_2) \geq M(S_1) = SW(S_1) - R_1$.

Now, consider the second operation, and assume that there are t paths from v to S_1. Consider the set $X = \{v_1, v_2, \ldots, v_r\}$ of nodes on these paths that are directly connected to the nodes of S_1. The smallest convex graph H containing all of the nodes in X is the union of the t paths. Let $HW(H)$ be the number of registers inserted after pipelining H. Under the assumption of monotonicity, $HW(H) \leq SW(H)$.

Once again, consider supergraph S_1'. Place $HW(H)$ additional registers on each incoming edge of the inputs of S_1 and $HW(S_1)$ addition registers on the outgoing edges of outputs of H that are not connected to inputs of S_1. By applying the same argument as above, this ensures that no more than m input nodes of S_1' will have the same number of registers on their incoming edges, and no more than n output nodes of S_1' will have the same number of registers on their outgoing edges.

Now, let $N(S_1)$ be a set of nodes in S_1 that are directly connected to nodes in H. This adds extra paths from v_{src} to nodes in $N(S_1)$, which come via H. To satisfy the legality constraint for these nodes, we must insert extra pipeline registers on some edges. Note that all paths from v_{src} to nodes in $N(S_1)$ that pass through S_1 will have at least $HW(H)$ registers; however, all paths from v_{src} to nodes in $N(S_1)$ that pass through H will have fewer than $HW(H)$ registers. It suffices to insert additional pipeline registers on the edges between nodes in H and nodes in S_1 to satisfy the legality constraint. This does not affect the quality of the original pipelining because these edges do not exist in S_1 or H alone. It is important to note that the addition of these pipeline registers do not increase the maximum number of registers on all paths from v_{src} to v_{sink}.

In the entire procedure, we have added only $HW(H)$ extra registers to each path through S_1 and $HW(S_1)$ extra registers on each path through H. Thus, the number of registers from v_{src} to v_{sink}, e.g., $HW(S_1')$ cannot exceed $HW(S_1) + HW(H)$. As discussed

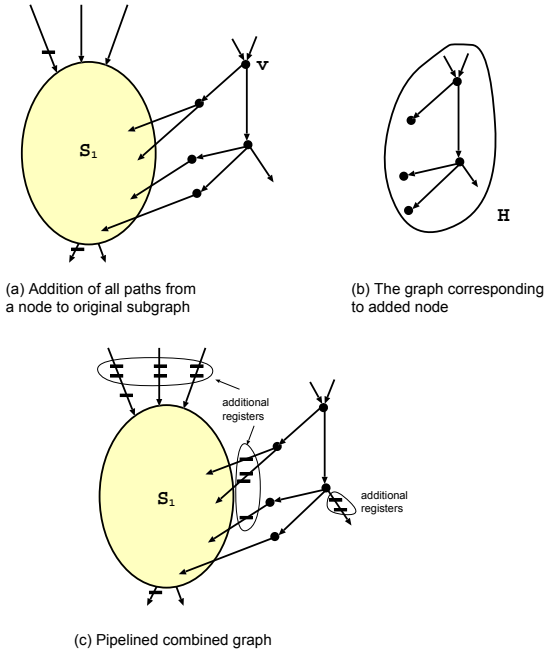


Figure 5: Adding a single node and all paths from this node to the subgraph increases the speedup.

above, the registers added on the paths containing nodes from both H and S_1 do not increase the number of registers between v_{src} and v_{sink}. Thus, $M(S_1') \geq SW(S_1) + SW(H) - (HW(S_1) + HW(H)) \geq M(S_1)$. Fig. 5 illustrates the key points of the proof in this case.

The proof for the third operation is analogous to the proof of the second, and has been omitted to conserve space. □

Theorem 1 indicates that increasing the number of nodes in an ISE can never reduce the speedup. Consequently, the optimal AFU will be maximal.

4.2 Reducing the DAG Size via Clustering

The previous section showed that the subgraph corresponding to an optimal AFU should be maximal; however, we must still show how to generate maximal subgraphs that are *Valid*, meaning they are convex and contain no forbidden nodes. One possibility would be to enumerate all convex subgraphs and then consider only maximal ones as AFU candidates. Although this approach eliminates the need to pipeline every nonmaximal subgraph that is enumerated, it is still not computationally feasible.

In this section, we show how to reduce the size of the original DAG in such a way that the set of maximal valid subgraphs is preserved. The key idea behind this reduction is that there are pairs (or sets) of certain nodes that will always occur together in maximal valid subgraphs. We cluster these nodes together into a single node, reducing the number of nodes and edges in the DAG. Section 4.2.1 defines an equivalence relationship between these clusterable nodes, and Section 4.2.2 introduces an efficient algorithm to determine this relationship.

4.2.1 Equivalence Relationship between Clusterable Nodes

We define a relation $\sim$ between two nodes u and v, such that $u \sim v$ is true if u and v are clusterable, and false otherwise. This relation is formalised as follows:

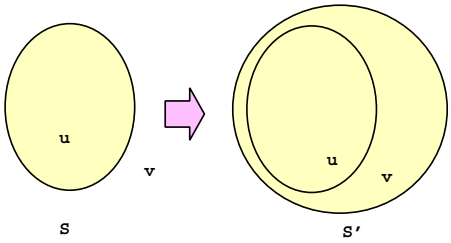


Figure 6: Two nodes are related if any convex subgraph containing one of them can be extended to a convex subgraph containing both of them.

DEFINITION 2. *For two nodes u and v in G, $u \sim v$ if and only if any valid subgraph containing u or v, but not both, can always be extended to a valid subgraph containing both u and v. Fig. 6 illustrates the preceding concept.*

It is trivial to see that the relationship is reflexive ($u \sim u$) and symmetric ($u \sim v \Rightarrow v \sim u$). To prove that $\sim$ is an equivalence relation, we must also prove that it is transitive.

THEOREM 2. *The relation $\sim$ is a transitive relation.*

PROOF. Let u, v, and w be nodes in G, and assume that $u \sim v$ and $v \sim w$. To establish transitivity, we must prove that $u \sim w$. Assume that there is a valid subgraph S that contains u, but not w. If $v \in S$, then S can be extended to a valid subgraph S' such that $w \in S'$, since $v \sim w$. If $v \notin S$, then S can be extended to a new subgraph S' such that $v \in S'$ since $u \sim v$. Since $v \in S'$, then S' can be extended to a new subgraph S'' such that $w \in S''$ since $v \sim w$. The same argument can be used to show that any subgraph S containing w can be extended to contain both u and w. □

Theorem 2 shows that we can partition the nodes of a DAG into equivalence classes, defined by the relation $\sim$, which is true for every pair of nodes belonging to the same equivalence class, and false for every pair of nodes belonging to distinct classes. If C is an equivalence class, any maximal valid subgraph must contain either all nodes in C, or none. This immediately implies that the nodes of an equivalence class can be clustered into a single node, thereby reducing the number of nodes and edges in the DAG. The clustering transformation is analogous to the edge contraction method described by Kastner *et al.* [10].

4.2.2 Determining the Equivalence Relation

Here, we describe an efficient algorithm to evaluate $u \sim v$ for two nodes u and v. One possibility is to enumerate all maximal valid subgraphs containing u, and then check whether v belongs to each of these subgraphs; however, this is highly inefficient. Fortunately, there is a much more efficient way to accomplish this, described here.

DEFINITION 3. *Given node u, the* Consistent Set *$P(u)$ is the set of all nodes x, such that there exists at least one valid subgraph containing both u and x. If $v \notin P(u)$, then no valid subgraph can contain both u and v.*

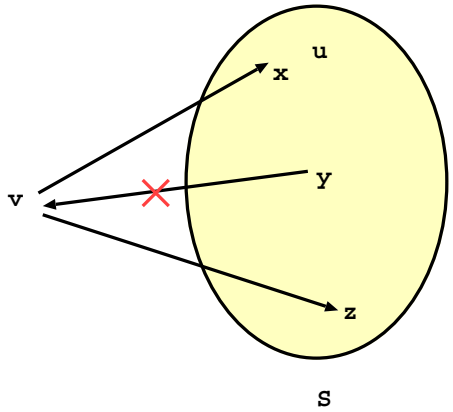
First and foremost, if f is a forbidden node, then $P(f)$ is empty. If u and v are not forbidden, and there is no path from u to v as well as from v to u, then $S = \{u, v\}$ is a valid subgraph, i.e., $v \in P(u)$ and vice versa.

Now, suppose that there is at least one path from u to v. Clearly, if there is a forbidden node along at least one path, then $v \notin P(u)$.

```
checkMembership (node u, node v, set F) {
// The function takes two nodes u,v and the set of
// forbidden nodes F and decides whether v ∈ P(u).
  (Pred_u, Pred_v) = (predecessors(u), predecessors(v));
  (Succ_u, Succ_v) = (successors(u), successors(v));

  if (({u, v} ∩ F ≠ ∅) or (Pred_u ∩ Succ_v ∩ F ≠ ∅) or
  (Succ_u ∩ Pred_v ∩ F ≠ ∅))
    return false;
  return true; }
```

Figure 7: Algorithm to decide the membership of v in $P(u)$.



Since S is convex, all paths between x and y must be inside S

Figure 8: All paths from a node to a convex subgraph must be monodirectional.

On the other hand, if there are no forbidden nodes, then the subgraph corresponding to the union of all of these paths will be a valid subgraph containing both u and v, hence $v \in P(u)$ and vice versa. Based on these observations, an algorithm to determine membership of v in $P(u)$ is given in Fig. 7. Theorem 3, which follows, provides the foundation to determine the relationship $\sim$ efficiently.

THEOREM 3. *For any two nodes u and v, $u \sim v \Leftrightarrow P(u) = P(v)$.*

PROOF. First, assume that $P(u) \neq P(v)$. Then there is a node x such that $x \in P(u)$ and $x \notin P(v)$, or vice versa. If $x \in P(u)$, then there will exist a valid subgraph S containing both u and x. Since $x \notin P(v)$, v cannot be included in S. Therefore, $u \not\sim v$.

Now, let us assume that $P(u) = P(v)$. Let us consider a valid subgraph S, such that $u \in S$ and $v \notin S$. For each node $s \in S$, $s \in P(u)$ by definition. Since $P(u) = P(v)$, $S \subseteq P(v)$. Now, consider all paths between v and the nodes in S. Since S is convex, these paths must be mono-directional, meaning that all of them will be from v to S, or from S to v; this is illustrated in Fig. 8. Moreover, none of these paths can contain a forbidden node, since $S \subseteq P(v)$. If we include all of the nodes of these paths in S, we have a convex graph with no forbidden nodes that contains both u and v. In other words, any valid subgraph containing u or v, but not both, can be extended to a valid subgraph containing both u and v. Therefore, $u \sim v$. □

Theorem 3 shows that one can easily find the clusters in an original DAG. First, the consistent set $P(u)$ is found for every node u. Then all of the nodes having the same consistent set are put into the same cluster. Since all forbidden nodes have null consistent

set, they will be placed in a single cluster, which henceforth will be called a *Forbidden Cluster*.

Any maximal subgraph can be found by taking the union of some subset of clusters. One possibility would be to enumerate all subsets of clusters, check the convexity of the corresponding subgraph, pipeline each convex subgraph, and pick the one that yields the best speedup. To make this approach more efficient, we formulate the problem in a different fashion, as described in the next section.

4.3 Cluster Graph and its Maximal Cliques

This section introduces the *Cluster Graph* of a DAG G, and shows that the problem of enumerating the maximal valid subgraphs of G is equivalent to enumerating the maximal cliques of the cluster graph. Since the number of maximal cliques in an undirected graph is significantly smaller than the subgraphs of a DAG, the clique enumeration approach is preferable. Prior work (e.g., [12]) has established relatively efficient methods for enumerating the maximal cliques of a graph.

DEFINITION 4. *The cluster graph $\mathcal{C}(G)$ of a DAG G is an undirected graph whose nodes correspond to the nonforbidden clusters of G, and an edge is placed between nodes corresponding to clusters C_1 and C_2 if and only if there exist two nodes u and v in G such that $u \in C_1$, $v \in C_2$ and $v \in P(u)$*

Since all nodes in a cluster have the same consistent sets, all nodes in that cluster will be consistent with all of the nodes in a neighbouring clusters. Likewise, if two clusters are not neighbours in the cluster graph, then there cannot be any valid subgraph containing nodes from both clusters. Next, we show that the maximal valid subgraphs of the DAG, G, correspond to maximal cliques of cluster graph $\mathcal{C}(G)$.

THEOREM 4. *Each maximal valid subgraph of G correspond to a maximal clique in the cluster graph and vice versa.*

PROOF. Applying Theorem 3, suppose that a valid subgraph S corresponds to the union of clusters $C_1, \ldots, C_m$. Since all nodes in a valid subgraph are consistent with one another, for any two nodes u and v in the subgraph: $v \in P(u)$ and $u \in P(v)$. If $u \in C_i$ and $v \in C_j$ then there is an edge between C_i and C_j in the cluster graph. Therefore, $C_1, \ldots, C_m$ is a clique in the cluster graph. This shows that each maximal valid subgraph corresponds to a clique in the cluster graph.

To establish the converse, assume that $C_1, \ldots, C_m$ is a maximal clique in the cluster graph, and that the corresponding subgraph in G is nonconvex. Then there must be two nodes $u \in C_1$ and $v \in C_2$ in the subgraph such that there is a path from u to v that includes nodes lying outside of the subgraph. Let $w \in C_{m+1}$ be a node on this path that lies in a cluster outside of the maximal clique.

Since C_1 and C_2 are connected by an edge in the cluster graph, u and v must be consistent, i.e., all paths between u and v contain no forbidden nodes; thus, all paths from u to w and from w to v contain no forbidden nodes as well. In other words, $u \in P(w)$ and $v \in P(w)$, indicating the presence of edges (C_1, C_{m+1}) and (C_2, C_{m+1}) in the cluster graph. Now, suppose that there is a cluster C_i in the clique that is not adjacent to C_{m+1}. This can only be possible if $w \notin P(u')$ for any node $u' \in C_i$. In other words, there must be a path from w to u (or u to w) that contains a forbidden node. This means that a forbidden node exists on one of the two paths $u \rightarrow w \rightarrow u'$ or $u' \rightarrow w \rightarrow v$. This leads to a contradiction, because $u \in P(u')$ and $v \in P(u')$. Therefore C_{m+1} is connected to all clusters in the clique $C_1, \ldots, C_m$, and, thus, $C_1, \ldots, C_{m+1}$ is a clique, contradicting the assumption that the former clique is maximal. This proves that any maximal clique in the cluster graph corresponds to a valid subgraph of G. □

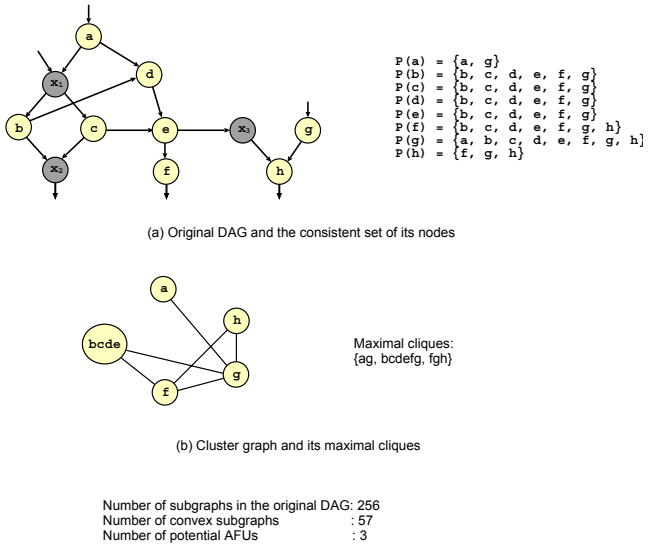


Figure 9: An example of forming cluster graph from the original graph and enumerating its maximal cliques. The black nodes in the graph indicate forbidden nodes.

Fig. 9 shows an example illustrating the execution of this method. This approach generates only the maximal valid subgraphs of the input DAG. Without any additional information about the speedup model, we cannot prune this initial set of potentially optimal ISEs any further; however, with more information, such as the hardware and software latencies of individual nodes, we can reduce the size of the cluster graph, and hence that of the search space, even further. For the sake of brevity, we will not discuss all possible approaches to reduce the size of the cluster graph; however, we will discuss two very simple methods. In the examples that follow, suppose that we know some lower bound α on the number of cycles saved by the AFU that implements the optimal ISE. A lower bound on the optimal number of cycles saved by the optimal AFU can be found by computing the number of cycles saved by any valid subgraph corresponding to any maximal clique in the cluster graph.

For the first example, let $N[C_1]$ be a set constaining cluster C_1 and all its neighbouring clusters. Let T_1 be the subgraph of the original DAG induced by all of the nodes in subgraphs corresponding to clusters in $N[C_1]$. Then, if $SW(T_1) \leq \alpha$, then we can trivially remove C_1 from the cluster graph. The reason is that any clique that contains C_1 corresponds to an AFU whose cycle savings is provably less than α, and the speedup attributable to this AFU is clearly suboptimal.

For the second example, suppose that there exist adjacent cluster nodes C_1 and C_2 in the cluster graphs, let $N[C_1 \cup C_2] = N[C_1] \cap N[C_2]$, and let T_{12} be the subgraph of the original DAG induced by all of the nodes in subgraphs corresponding to clusters in $N[C_1 \cup C_2]$. If $SW(T_{12}) \leq \alpha$, then the edge (C_1, C_2) can be removed from the cluster graph using similar reasoning as above.

4.4 Pruning the Set of ISE Candidates without Pipelining

By applying the procedures described in the previous sections, we can find a potential set of optimal ISEs by enumerating the maximal cliques in the cluster graph. Nonetheless, we must still

determine the best ISE in terms of speedup achievable by its AFU. In order to compute this speedup, we must pipeline it, which involves solving another optimisation problem. Solving this problem for a large set of AFUs, however, is still quite expensive. To reduce the number of AFUs to pipeline, we first calculate some upper and lower bounds on the speedup of each AFU without pipelining. We use these results to eliminate some candidate AFUs from consideration prior to pipelining. Specifically, if the upper bound on the speedup of some AFU is less than the lower bound of another, then there is no need to consider the first and it can be discarded from the set of potentially optimal AFUs. We exploit the RISC speedup model to compute the upper and lower bounds, as described in the next two theorems.

THEOREM 5. *The sppedup of an AFU after pipelining can be bounded as follows:*

- $M(S) \leq \sum_{u \in S} SW_u$ - $\mathsf{max}\left(\frac{IN(S)}{m}, \frac{OUT(S)}{n}\right)$.
- $M(S) \geq \sum_{u \in S} SW_u$ - $\left\lceil \frac{IN(S)}{m} \right\rceil$ - $\left\lceil \frac{OUT(S)}{n} \right\rceil$ - $2\left\lceil \frac{HW_{crit}(P)}{\lambda} \right\rceil$.

We omit the proofs of the two theorems due to lack of space.

4.5 Formulation of AFU Pipelining as a Matrix Problem

Finally, we are left with a set of potentially optimal AFUs that must be pipelined by solving the optimisation problem introduced in Section 3. One possibility could be to consider all possible assignments of registers on incoming edges of input nodes and then allocate register on the edges according to the *ASAP* scheduling heuristic, as described by Pozzi and Ienne [6]; afterwards, the best assignment among all of those considered is chosen. The problem with this algorithm is that it has an exponential worst-case time complexity, which makes it impractical for AFU candidates with a large number of input nodes.

Our approach, in contrast, is to solve the problem with a heuristic that runs significantly faster than Pozzi and Ienne′s algorithm. In our experimental evaluation, our heuristic solved all problem instances optimally; however, we do not have a formal proof of optimality. To describe our method, we begin by introducing some terminology.

DEFINITION 5. *The* Integral Path Delay *of a path is the minimum number of registers that need to be inserted in the path such that the sum of hardware latencies of all nodes between two consecutive registers does not exceed λ.*

DEFINITION 6. *The* Integral Critical Path Delay *from node u to v in a DAG, $ICD(u, v)$, is defined to be the maximal integral path delay along all paths from u to v. If there is no path from u to v, then $ICD(u, v) = -\infty$.*

DEFINITION 7. *The* Residual Hardware Latency *from u to v, $RHL(u, v)$, is the maximal sum of hardware latencies of all nodes after the $ICD(u, v)$-th register among all paths between u and v.*

Both $ICD(u, v)$ and $RHL(u, v)$ can be computed efficiently by traversing the nodes of the DAG in topological order.

There is an advantage to computing the ICDs and RHLs in advance. Once we have a configuration of the number of registers on incoming edges of the inputs of the AFU, there is no need to schedule the DAG in order to compute the optimal number of registers between v_{src} and v_{sink}. Theorem 6, which follows from Lemma 1, shows that $HW(S)$ can be computed using ICD values alone, without actually scheduling it.

LEMMA 1. *For any node v, there exists a path p (defined as* Dominant Path *for v) from v_{src} to v, such that if we remove all nodes from the DAG except those in p, then the number of registers from v_{src} to v, as well as the RHL, will remain the same.*

PROOF. The proof itself uses induction on the height of node v in the DAG, meaning the maximum path length (in terms of nodes) from v_{src} to v. The base case corresponds to the case when v is an input node, which is trivial, since there is one dominant path p of length 0.

For the induction step, assume that the lemma holds for all nodes of height less than r. Now, consider node v at height r: Since all predecessors of v have height less than r, there exist dominant paths from v_{src} to each predecessor. Now, for node v, let us remove all nodes from the DAG except for v and these paths. This will not affect v, since there will be no difference in the number of registers placed on the dominant paths from v_{src} to each predecessor of v; likewise the RHL remains the same as well. Let u_j be the input node that has the maximal number of registers from v_{src}; if there is more than one such node u_j, then ties can be broken using the RHL. Thus, u_j will determine the number of registers prior to v, along with $RHL(u_j, v)$. Hence, the dominant path for the corresponding predecessor of v will also be the dominant path for v. This completes the induction step. □

THEOREM 6. *Let S be an AFU with input nodes $u_1, \ldots, u_k$ and $x_1, \ldots, x_k$ registers on each incoming edge. Let $R(v)$ denote the maximum number of registers placed on a path from v_{src} to v. Then,*

$$R(v) = \mathsf{max}_{1 \leq i \leq k}(x_i + ICD(u_i, v)).$$

PROOF. Follows directly from the proof of Lemma 1. Without loss of generality, if the dominant path for v passes through input u_1, then $R(v) = x_1 + ICD(u_1, v)$. □

Note that for each input node u_i, $ICD(u_i, v)$ can be computed without knowing the specific value of x_i; moreover, the ICDs can be used to determine the proper value of x_i. Now, if the AFU has l outputs, $v_1, \ldots, v_l$ with $y_1, \ldots, y_l$ registers on their outgoing edges, then the total number of registers from v_{src} to v_{sink} will be given by:

$$R(v_{sink}) = \mathsf{max}_{(1,1) \leq (i,j) \leq (k,l)}(x_i + ICD(u_i, v_j) + y_j).$$

We know that exactly m input nodes will have zero registers on their incoming edges, m inputs will have one register on their incoming edges, etc. The same holds for the number of registers on the outgoing edges of output nodes. Consequently, the only task that we have to do is to find the best mapping between the number of registers and inputs of the AFU, as well as the number of registers and outputs on the AFU. In other words, the pipelining problem translates into the following matrix problem.

PROBLEM 2. *Given an $m \times n$ integer matrix A, an m dimensional integer array R, and an n dimensional integer array C, find permutations $\pi : \{1, 2, \ldots, m\} \rightarrow \{1, 2, \ldots, m\}$ and $\sigma : \{1, 2, \ldots, n\} \rightarrow \{1, 2, \ldots, n\}$, such that the following expression is minimised:*

$$F(A, R, C, \pi, \sigma) = \mathsf{max}_{i,j}(R[\pi(i)] + a_{ij} + C[\sigma(j)]).$$

Column vectors R and C correspond to the number of registers placed on each input and output node respectively, and A is the set of ICD values for every pair of nodes in the graph.

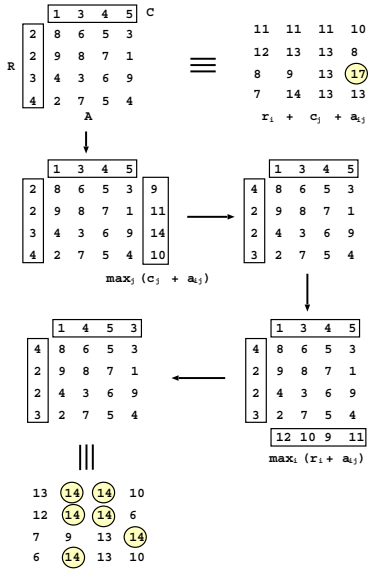


Figure 10: Execution of ping-pong algorithm on a simple example. Note that in this case ping-pong finds the optimal solution only in two iterations.

4.6 Heuristic to Solve the Matrix Problem

First, note that if all entries of vector C were zero, then the permutation σ would be meaningless. The resulting problem would be to find a permutation $\pi : \{1, 2, \ldots, m\} \rightarrow \{1, 2, \ldots, m\}$ such that the following simpler equation was minimised:

$$F(A, R, \pi) = \mathsf{max}_{1 \leq i \leq m}(R[\pi(i)] + a_{ij}).$$

This problem can be solved very easily by assigning the smallest value in $R[.]$ to the largest value of $\mathsf{max}_j(a_{ij})$, the second smallest value of $R[.]$ to the second largest value of $\mathsf{max}_j(a_{ij})$, etc. In other words, the permutation π corresponds to a reverse sorting order of $\mathsf{max}_j(a_{ij})$ values.

Similarly, if permutation σ is fixed, we can find the optimal permutation π by sorting $\mathsf{max}_j(a_{ij}+C[\sigma(j)])$; an analogous situation occurs when π is fixed and we want to find an optimal permutation σ. The heuristic that we use to compute π and σ is based on these observations. We start with randomly generated permutations for π and σ. First, we find the optimal π, given σ. Next, we find the optimal σ, given π. Then, once again, the optimal π, given σ. The process continues as long as at least one of the permutations changes. This heuristic will henceforth be referred to as the *Ping-Pong Algorithm*. Fig. 10 shows an example illustrating the execution of the Ping-Pong Algorithm. It can be proved that if all entries in the input matrix A and arrays R, C are bounded, then the ping-pong algorithm is guaranteed to converge.

5. EXPERIMENTS

The algorithms described in Section 4 were implemented in C++. The input to the program is a DAG, representing a basic block of the application. The program identifies the optimal ISE and outputs the AFU generated for the ISE after pipelining. The RISC speedup model is used to measure the relative performance of AFUs. The software latency of an instruction is estimated to be the latency of the execution stage of the RISC pipeline. The hardware latency of an instruction is estimated by synthesising the corresponding operator on a UMC 0.18μm CMOS technology standard cell library; this latency is then normalized to the delay of a 32-bit multiply accumulate (MAC) operation.

The algorithm was run on four benchmarks: *Adpcm coder*, *Adpcm decoder*, *Viterbi*, and *AES*. For each benchmark, we compute the speedup under I/O constraints $(2, 1), (4, 2), (10, 5)$, and (∞, ∞), the last of which indicates no constraints on the I/O. The results are compared with the subgraph enumeration method of Atasu *et al.* [4] both without pipelining, and with pipelining as described by Pozzi and Ienne [6]. The results are shown in Fig. 11.

Fig. 11 shows that pipelining increases the speedup for all benchmarks because of its ability to identify larger subgraphs as AFUs. In all four benchmarks, our algorithm provides speedups equal to or exceeding that of the previous pipelining method, depending on the I/O constraints. In some cases, the previous method, although a heuristic, managed to find optimal solutions which the new method could not improve upon.

In the case of AES for an I/O constraint of $(10, 5)$, the new algorithm selects an AFU that results in a speedup of 5.05, compared to a speedup of 4.3 generated by the algorithm of Pozzi and Ienne. The reason is that the proposed algorithm found a subgraph having 22 inputs and 22 outputs. Pozzi and Ienne's algorithm, in contrast, only considers subgraphs having I/O constraints up to a fixed value, which was limited to $(10, 5)$. If the fixed value was relaxed to $(22, 22)$, their approach could have found the same solution; however, they do not advise relaxing the I/O constraints beyond $(10, 5)$ due to runtime considerations. For other benchmarks, the algorithm presented yields marginal improvements in AFU quality. In these cases, the optimal AFUs have relatively low I/O constraints, and thus they can be found by existing algorithms.

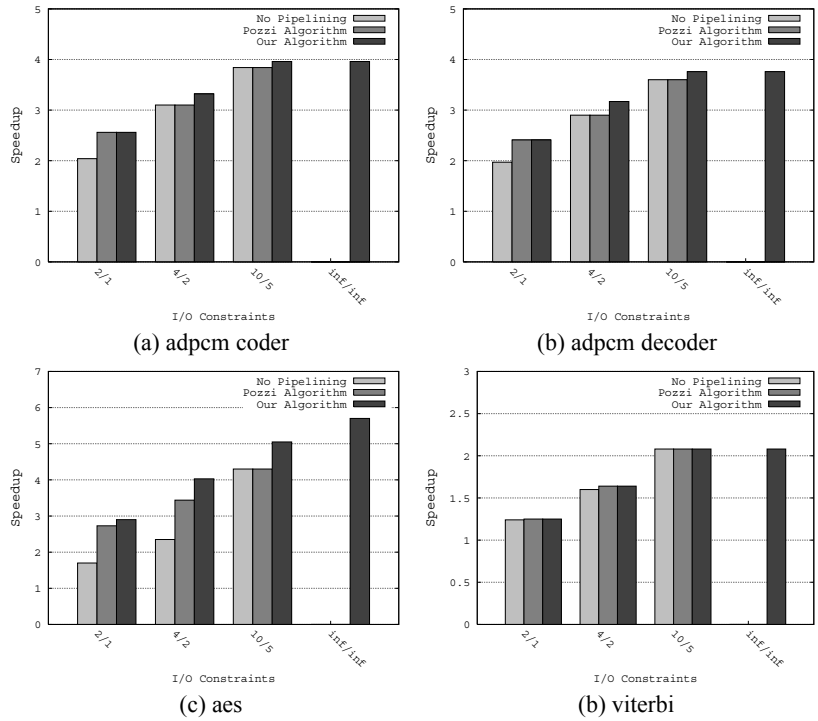
The runtime of the algorithms in this paper are significantly faster than the runtimes reported by Atasu *et al.* [4, 5] and by Pozzi and Ienne [6]. For a constraint of $(10, 5)$ on AES, the previous methods required several hours to perform subgraph enumeration and pipelining; the algorithms described in this paper, in contrast, converged in approximately 30 seconds.

6. CONCLUSIONS

A new approach for ISE generation for extensible processors has been presented. The new approach exploits the fact that the use of pipelined AFUs to implement ISEs eliminates the I/O constraints imposed by previous formulations of the ISE generation problem. Without I/O constraints, prior techniques for ISE generation suffer from runtimes that are exponential in the number of nodes in their graphs. The approach presented in this paper, in contrast, exploits the monotonicity of the speedup function to reduce the number of nodes in the graph via clustering. A cluster graph has been introduced to model the compatibility between clusters and permit the construction of large ISEs by combining clusters. Clique enumeration in the cluster graph has been empirically observed to be much faster than subgraph enumeration in the original DAG. We have also introduced a faster algorithm for pipelining than the current state-of-the-art algorithm developed by Pozzi and Ienne [6].

Most importantly, we have shown that for a base processor that is a RISC, the speedup model used to evaluate the benefit of a custom instruction is independent of the specific details of the RISC pipeline. The benefit from this result is significant because it ensures that engineers do not need to study the architectural details of the pipeline in order to identify good ISEs as future generations of extensible RISC processors are developed.

Altogether, our approach to ISE generation runs significantly faster than the prior approach of Atasu *et al.* [5]. Pozzi and Ienne [6] use an I/O constraint of $(10, 5)$ to limit the size of the ISEs identified; we have found that for AES, the optimal ISE has an I/O



(a) adpcm coder (b) adpcm decoder

(c) aes (b) viterbi

Figure 11: Comparison of the speedup values of AFUs generated by our algorithm with the state of art techniques.

constraint of $(22, 22)$; standard subgraph enumeration methods require several hours to identify a suboptimal $(10, 5)$ ISE, while the new approach found the optimal ISE in approximately 30 seconds, testifying to both the efficacy and efficiency of the proposed technique.

7. REFERENCES

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

[2] I. Kuon and J. Rose, "Measuring the gap between FPGAs and ASICs," in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. CAD-26, no. 2, pp. 203–15.

[3] 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*, Washington, D.C., Sept. 2004, pp. 69–78.

[4] K. Atasu, L. Pozzi, and P. Ienne, "Automatic application-specific instruction-set extensions under microarchitectural constraints," in *Proceedings of the 40th Design Automation Conference*, Anaheim, Calif., June 2003, pp. 256–61.

[5] 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*, vol. CAD-25, no. 7, pp. 1209–29, July 2006.

[6] 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*, San Francisco, Calif., Sept. 2005, pp. 2–10.

[7] N. Pothineni, A. Kumar, and K. Paul, "Application specific datapath extension with distributed I/O functional units," in *Proceedings of the 20th International Conference on VLSI Design*, Banglore, 2007, pp. 551–58.

[8] Y. Guo, G. Smit, H. Broersma, and P. Heysters, "A graph covering algorithm for a coarse grain reconfigurable system," in *Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, 2003, pp. 199–208.

[9] 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*, Grenoble, France, Oct. 2002, pp. 262–69.

[10] 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)*, vol. 7, no. 4, pp. 605–27, Oct. 2002.

[11] S. Cadambi and S. C. Goldstein, "CPR: A configuration profiling tool," in *Proceedings of the 7th IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa Valley, Calif., Apr. 1999, pp. 104–13.

[12] D. Johnson, M. Yannakakis, and C. Papadimitriou, "On generating all maximal independent sets," *Information Processing Letters*, no. 27, pp. 119–123, 1988.