On the Complexity of Enumeration and Scheduling for Extensible Embedded Processors

Paolo Bonzini and Laura Pozzi
Faculty of Informatics
University of Lugano (USI)
Switzerland
Email: paolo.bonzini@lu.unisi.ch, laura.pozzi@unisi.ch

University of Lugano
Faculty of Informatics
Technical Report 2008/07
December 2008

Abstract

Compiling for extensible processors includes searching the application's data-flow graphs for code sequences that can be added (as custom instructions) to the core instruction set, as well as finding optimal ways to use these sequences at run-time. Depending on the targeted architecture, different algorithms may be adopted, but toolchains for different architectures often share two common building blocks. The first is a subgraph enumeration algorithm that lists subgraphs that satisfy particular constraints; this paper proves that a well-known branch-and-bound algorithm, previously thought to have worst-case exponential complexity, actually achieves optimal complexity (polynomial in the size of the graph). The second building block is a scheduling algorithm that computes an optimal order for feeding inputs to application-specific functional units, as well as for retrieving outputs; we prove the NP-completeness of this problem by reducing a flowshop scheduling problem to it.

1. Introduction

Customizable processors have recently emerged thanks to their ability to balance the inexpensiveness and the flexibility of general purpose processors, with the speed and power advantages of custom circuits (ASICs). In such processors, a standard machine language can be augmented with *custom instructions* (also known as *instruction set extensions*, or ISEs) that execute on application-specific functional units.

Among the tools that automate the design process for a customizable processor, the compiler has a chief importance, because its role has two complementary facets. First, the compiler performs a deep analysis of the program, and can therefore infer the optimal set of extensions that will benefit most; second, the behavior of the compiler itself—in particular the machine description—is affected by the presence of instruction set extensions. These two aspects, when combined, signify that a compiler for customizable processors can both generate a machine description, or parts of it, *and* compile onto it.

In the past years several algorithms have emerged for customizable processor compilation. However, very few of these have been analyzed theoretically to ascertain the time complexity of the problem or the optimality (also in terms of time complexity) of the algorithms.

In this paper we focus on two problems. The first is *subgraph enumeration*, which is used to generate a set of candidate instruction set extensions. This paper continues previous work on analysis of this problem, in which Chen *et al.* [6] established a polynomial upper bound for the output size, while Bonzini *et al.* [5] provided an algorithm that provably achieved that bound. In this paper we will prove that a well-known branch-and-bound algorithm from [9] also has the same complexity—and, unlike the one in [5], it is simple to understand and does not require complex pruning techniques in order to achieve practical run-times.

The second is *I/O scheduling*, which is used to generate an optimal ordering of the inputs and the outputs, constrained by the number of data that can be fed to (and read from) the external functional units. Literature includes two solutions to this problem—an optimal one with exponential complexity [10], and an approximate one with polynomial complexity [11] which the authors experimentally observed to produce optimal results for several benchmarks. In this paper, we actually prove the NP-completeness of this problem, based on reducing a particular 2-machine flowshop scheduling problem to the I/O scheduling problem.

These results give a better understanding the problem space, and provide a stronger basis for future work in this field. For example, a polynomial time bound for subgraph enumeration is useful when studying the impact on instruction set extensions of compiler transformations, particularly those that can possibly increase basic block size [3, 4].

The rest of the paper is organized as follows. Section 2 surveys previous work on this topic from the customizable processors community. Sections 3 and 4 present our results on the two problems, respectively subgraph enumeration and I/O scheduling. Section 5 concludes the paper and presents possible extensions of this work.

2. Related work

Identifying custom instructions is usually seen as a time-consuming job, under the rationale that the number of possible patterns can grow exponentially with the number of instructions in the basic blocks considered by the algorithm. This is true in general; however, as a consequence of the lack of theoretical analysis of the problem, most literature wrongly considered this to be the case even if microarchitectural constraints such as the number of register file ports are taken into account.

In fact, the algorithm we analyze in this paper was first proposed by Atasu *et al.* [2] and enumerates all valid patterns by constructing a search tree whose depth is equal to the size of the graph, and where each node has two children, corresponding to including or excluding a node from the graph. While this trivially implies a complexity of $O\left(2^n\right)$, the algorithm avoids exhaustive search by checking whether adding more nodes to the subgraph could "repair" the constraint violations: if this is not the case, one can discard entire branches of the search tree.

The algorithm was further refined in [9] by introducing a new pruning criterion. As we will show, this addition is fundamental to achieve a new bound on the complexity, that is polynomial in n, the size of the graph. However, both [2] and [9] only overviewed how to implement the branch-and-bound criteria in an efficient manner. In this paper we present a possible implementation in detail, which is actually necessary in our complexity proof.

Despite the authors of [9] observe that in practice the run-time grew relatively slowly with n, they present the algorithm as having a worst-case exponential complexity. Because of this, other algorithms were developed that restricted the set of graphs that can be enumerated; for example, Yu and Mitra proposed an alternative algorithm that enumerates only connected patterns [12], but also present its complexity as worst-case exponential. Zhang *et al.*, instead, use the FlowMap algorithm (used in FPGA technology mapping) to find single-output patterns in a graph [7]. Multiple-output patterns can then be derived from the result of these simpler enumerations [13].

Two works present algorithms that are alternative to the one of [9] but solve the same problem. The first, by Chen *et al.*, does not rely on postorder sorting of the nodes, and instead uses constraint violations to guide the search towards nodes that "help" repairing those violations [6]. This paper also is the first to prove a bound for the number of valid sub-

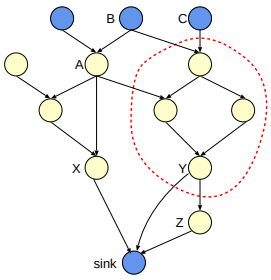

Figure 1. A data-flow graph and a convex cut within it. Shaded nodes are forbidden. The nodes within the dashed area form a convex cut with inputs A, B, C, and a single output X.

graphs (i.e. for the algorithm's output), that is polynomial in the size of the graphs. However, the authors did not perform a substantial complexity analysis of their algorithm, whose complexity is once more declared worst-case exponential.

The only known polynomial algorithm for this problem so far is found in [5]. Relying on the relationship between multiple-vertex dominators [8] and single-output convex cuts, the paper presents an algorithm whose expected complexity is indeed polynomial in the size of the graph (for a fixed maximum number of inputs and outputs in the enumerated subgraphs). However, the algorithm is complicated to implement, and the paper only overviews the techniques that are required for it to compete in speed with Pozzi *et al.*'s. It is hence useful to prove that a simpler algorithm in [9] actually has the same complexity.

All the algorithms we presented so far consider four constraints: the number of inputs and outputs of the enumerated subgraph, the convexity of the cut, and the absence from the cut of vertices included in a set of *forbidden* nodes. If only the last two constraints are kept, the resulting problem becomes equivalent to clique enumeration [11].

In this paper we only consider the complexity of enumeration under I/O constraints, because clique enumeration is a well known *EXPTIME* problem[1]. However, research on removing the I/O constraints led to the study of another combinatorial problem, that is I/O scheduling. This was introduced in [10] as a way to minimize the latency of the ISE and, secondarily, the number of registers in the circuit; a variant that only minimizes the latency was formulated in [11], where the authors also report good results using a heuristic solver. This second, weaker formulation is the second problem we analyze in this Section 4 of this paper, concluding that it is an NP-complete generalization of 2-machine flowshop scheduling.

[1]If only the best candidate is needed instead the problem is NP-complete, and indeed Atasu *et al.* propose in [1] an ILP formulation of the problem.

3. Subgraph enumeration

In this section, we consider the subgraph enumeration problem. In Section 3.1 we introduce a few definitions and formalize the problem (using a simpler formulation than the one used in [9]). In Section 3.2 we detail the algorithm outline presented by Pozzi *et al.*, in order to be able to prove a new time complexity bound in Section 3.3.

3.1. Definitions

As depicted in figre 1, a data flow of each basic block is represented by a graph $G(V, E)$. The definition of cut, and in particular of convex cut, are as follows.

Definition 1 (Cut): A cut S is a subgraph of a direct acyclic graph G. We call *inputs of* S the set $\mathrm{I}(S)$ of predecessor vertices of those edges which enter the cut S from the rest of the graph G, that is $\mathrm{I}(S) = \bigcup_{v \in S} \mathrm{pred}(v) \setminus S$. Similarly, we call *outputs of* S the set $\mathrm{O}(S)$ of vertices which are part of S, but have at least one successor $v \notin S$.

Definition 2 (Convex cut): A cut S is convex if there is no path from a vertex $u \in S$ to another vertex $v \in S$ which contains a vertex $w \notin S$.

The shaded area in Figure 1 is an example of a convex cut. Node[2] X is an output and nodes A, B, C are inputs.

In order to define the problem, we define three subsets of V that are of interest. First of all, I_{ext} (external inputs) are input variables of the basic block, and are a subset of the source vertices. Dually, O_{ext} (external outputs) is the set of values that are computed in the basic block and are live at the end of the basic block.

Nodes that cannot be included in a cut are called *forbidden* and form a set F. They may still be chosen as inputs to a cut. Some forbidden nodes will be marked as such by the user, and represent operations that are not allowed in a special instruction—for example, loads and stores if the custom functional unit cannot have any memory port. In addition to these nodes, nodes in I_{ext} are implicitly forbidden, because their values are calculated outside the basic block.

F includes a dummy v_{sink} vertex which is forbidden and also a successor of all external outputs. This node is introduced in order to simplify the formulation: since every external output that is part of a cut S will always have a successor outside the cut (namely v_{sink}), it will also be included in $\mathrm{O}(S)$.

In Figure 1, shaded circles represent forbidden nodes: of these, the three nodes on the top line are external inputs, while the sink is the successor of the three external outputs X, Y, Z. The nodes within the dashed area form a convex cut with three inputs and one output.

[2]The terms *vertex* and *node* will be used interchangeably.

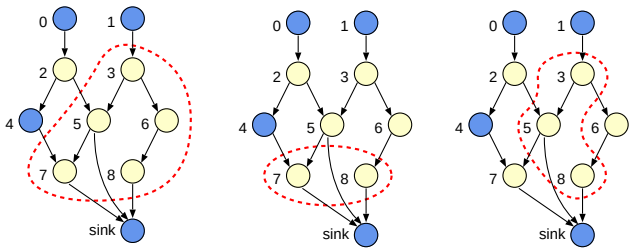

Figure 2. Eliminating invalid cuts. Supposing $N_{\text{in}} = 2$, $N_{\text{out}} = 2$, and that the search has reached node 2 (in reverse topological order, i.e. starting from node 8), these three cuts are all rejected: a) has three outputs (5-7-8), b) has three permanent inputs (4-5-6), c) is not convex. Entire branches of the search tree can be skipped: the cuts are invalid independent of whether or not node 2 is part of the cut.

The target architecture may pose additional constraints on the cuts that can be accepted. We consider constraints on the maximum number of read and write ports in the register file which a custom instruction can use. These are indicated respectively by N_{in} and N_{out}. For example, the cut of Figure 1 will not be part of the solution if $N_{\text{in}} < 3$.

Thus, the posed problem can be formalized as follows:

Problem 1 (Subgraph enumeration) *Given a direct acyclic graph G, a set of forbidden nodes F, and a maximum number of inputs N_{in} and of outputs N_{out}, enumerate all the convex cuts $S \subseteq G$ under the constraints that $|\mathrm{I}(S)| \leq N_{\text{in}}$, $|\mathrm{O}(S)| \leq N_{\text{out}}$, and $S \cap F = \emptyset$.*

3.2. Algorithm

The algorithm we analyze is the one outlined in [9]. The basic idea of the algorithm is to order nodes topologically and process them backwards; this allows several optimizations because the following invariants hold[3]:

1. Adding to a convex cut S a node u that (in the chosen topological order) comes before every node $v \in S$, will not remove any output from S.

2. Adding to a convex cut S a node u that (in the chosen topological order) comes before every node $v \in S$, will not remove from $\mathrm{I}(S)$ the inputs coming after u in the topological order.

3. Adding to a non-convex cut a node u that (in the chosen topological order) comes before every node $v \in S$, will not restore the convexity of the cut.

[3] Proofs are included in [2] (invariants 1 and 3) and [9].

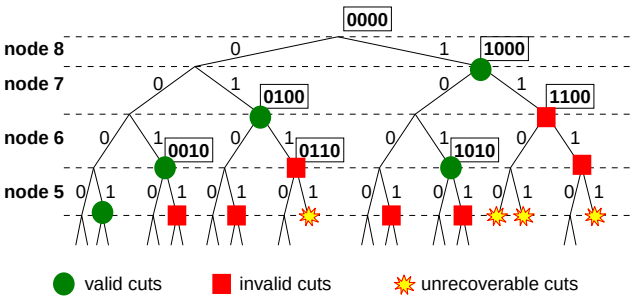

Figure 3. Pruning the search tree. For $N_{\text{in}} = 2$, $N_{\text{out}} = 2$, the first four levels of the search tree have to be visited completely. On the fourth level, however, five "unrecoverable" cuts are found, and the entire branch below those cuts can be discarded.

Of these invariants, the third is only needed to prove the correctness of the algorithm; the first two instead are also central to proving its complexity.

From each of these invariants we can derive a condition that, if broken, allows to discard the entire search tree under S (see Figure 3). These conditions, represented in Figure 2, are respectively that $|O(S)| > N_{\text{out}}$, that $|\{u_i \in I(S) : i \geq index\}| > N_{\text{in}}$, and that S is not convex.

The pseudocode in Figure 4 is an implementation of the algorithm. Function SEARCH tries adding to S all the nodes $\{u_i \in V : i < index\}$, and expects as a precondition that S does not include any of them. It also assumes that the last node in the topological order is forbidden. This is true because the last node in the order will have no successors and, after the artificial sink node v_{sink} is added, it will be the only node without a successor.

Unlike the pseudocode in Pozzi *et al.*'s paper, Figure 4 also includes the details of an $O(1)$ implementation of search tree pruning. Function SEARCH maintains a count of outputs and *permanent inputs*, i.e. nodes that are inputs for all the cuts in the nodes that will be explored recursively; these are those inputs $u_i \in \mathrm{I}(S)$ such that $i \geq index$. These two counts are updated on every recursive call. If the number of outputs or permanent inputs exceeds, respectively, N_{out} or N_{in}, exploration of an entire branch of the search tree can be avoided.

3.3. Complexity proof

Based on this implementation, we will now prove that, in addition to the exponential $O(2^n)$ upper bound for time, this algorithm also admits an alternative bound of $O\left(n^{N_{\text{in}}+N_{\text{out}}}\tau(n)\right)$, where $\tau(n)$ is the complexity of processing a leaf of the search tree and of the convexity test (whichever is more expensive). Our proof is constructive; we transform the pseudocode so that the different upper bound is clearly visible.

The first step is to move to the caller the update of n_{permin} and n_{out} according to how many successors of u_{index} are in the cut. The modified pseudocode of Figure 5 shows that three cases are possible:

- if the node has zero successors in the cut, adding it to the cut will create an output;
- if the node has at least one successor in the cut, and at least one successor *not* in the cut, adding it to the cut will create an output, and excluding it will turn it into a permanent input;
- if the node's successors are all part of the cut, adding it to the cut will not create an output, but excluding it will still create a permanent input.

In order to query how many successors of any node are part of S, a side table is updated every time nodes are added and removed from the cut. When node u is added or removed, the count changes for all its predecessor, giving a cost of $O(d_{in})$, where d_{in} is the maximum in-degree of G, for each recursive call. This cost is smaller than $\tau(n)$, because the convexity test can also be done in $O(d_{in})$ time, and thus can be ignored.

We then proceed to transform one of the two recursive calls into iteration. In Figure 6, each recursive call then increments one of n_{permin} or n_{out}. Since $n_{\text{permin}} < N_{\text{in}}$ and $n_{\text{out}} < N_{\text{out}}$, there can be no more than $N_{\text{in}} + N_{\text{out}}$ recursive calls active at any time, each of which will execute the **while** loop at most n times. This proves the complexity result given at the beginning of this section.

[9] actually includes a more general condition for declaring an input permanent. In addition to all inputs coming after u_{index} in the topological order, *all forbidden inputs* (including external inputs I_{ext}) *are permanent*. Since they cannot be included in the cut, adding nodes to S will not remove forbidden inputs from $\mathrm{I}(S)$. This allows the algorithm to achieve even better complexity in practice.

Adding this more efficient condition to our implementation is easy. For the pseudocode in Figure 5, for example, it suffices to add the following line at the very beginning of the function:

$$n_{\text{permin}} = n_{\text{permin}} + |(\mathrm{I}(S) \setminus \mathrm{I}(S \setminus \{u_{index}\})) \cap F|$$

4. I/O scheduling

The algorithm from Section 3 can be used to enumerate subgraphs whose number of inputs and outputs is smaller than the number of register file ports. Unfortunately, ports are an expensive asset of the processor.

```
SEARCH(S, index, n_permin, n_out)
  ▷ u_0 to u_{|G|-1} represent nodes of G, ordered topologically
  ▷ u_{|G|} is the artificial sink node
  ▷ The search is started with SEARCH(∅, |G|, 0, 0).
  if u_index ∉ S then
      if ∃v ∈ succ(u_index) : v ∈ S then
          n_permin = n_permin + 1
      if n_permin > N_in then return
  else
      if u_index ∈ F then return
      if ¬∀v ∈ succ(u_index) : v ∈ S then
          n_out = n_out + 1
      if n_out > N_out then return
      if S is not convex then return
      if |I(S)| ≤ N_in then S is a valid cut

  if index > 0 then
      SEARCH(S ∪ {u_{index-1}}, index − 1, n_permin, n_out)
      SEARCH(S, index − 1, n_permin, n_out)
```

Figure 4. Subgraph enumeration algorithm from [9].

```
SEARCH-2(S, index, n_permin, n_out)
  if n_permin > N_in ∨ n_out > N_out then return
  if u_index ∈ S then
      if u_index ∈ F then return
      if S is not convex then return
      if |I(S)| ≤ N_in then S is a valid cut

  if index > 0 then
      if ¬∃v ∈ succ(u_{index-1}) : v ∈ S then
          ▷ No successors are in the cut
          SEARCH-2(S ∪ {u_{index-1}}, index − 1, n_permin, n_out + 1)
          SEARCH-2(S, index − 1, n_permin, n_out)
      elseif ¬∀v ∈ succ(u_{index-1}) : v ∈ S then
          ▷ Some (but not all) successors are in the cut
          SEARCH-2(S ∪ {u_{index-1}}, index − 1, n_permin, n_out + 1)
          SEARCH-2(S, index − 1, n_permin + 1, n_out)
      else
          ▷ All successors are in the cut
          SEARCH-2(S ∪ {u_{index-1}}, index − 1, n_permin, n_out)
          SEARCH-2(S, index − 1, n_permin + 1, n_out)
```

Figure 5. Moving checks to the caller.

```
SEARCH-3(S, index, n_permin, n_out)
  start = index
  while index ≥ 0 ∧ n_permin ≤ N_in ∧ n_out ≤ N_out ∧
          ∧ S ∩ F = ∅ ∧ S is convex do
      if u_index ∈ S ∧ |I(S)| ≤ N_in then S is a valid cut

      index = index − 1
      if index > 0 then
          if ¬∃v ∈ succ(u_index) : v ∈ S then
              SEARCH-3(S ∪ {u_index}, index, n_permin, n_out + 1)
          elseif ¬∀v ∈ succ(u_index) : v ∈ S then
              SEARCH-3(S ∪ {u_index}, index, n_permin, n_out + 1)
              n_permin = n_permin + 1
          else
              SEARCH-3(S, index, n_permin + 1, n_out)
              S = S ∪ {u_index}

  S = S \ {u_i : i < start}
```

Figure 6. Eliminating one recursive call.

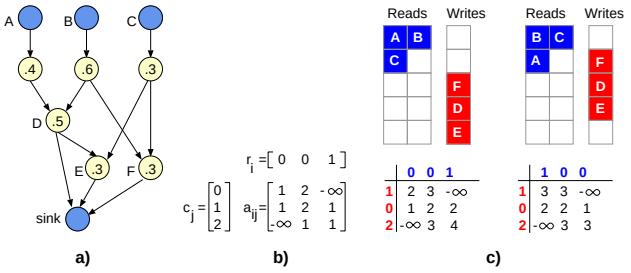

Figure 7. The I/O scheduling problem: a) a data-flow graph whose I/O constraints exceeds the bandwidth of the register file; 2) its integral critical path delay matrix A, and the row/column vectors corresponding to a register file with 2 read ports and 1 write port; b) a permutation of R and C giving a non-optimal solution; the permutation of R gives the order of inputs, while the permutation of C gives outputs in reverse order; c) a permutation of R and C corresponding to an optimal solution.

Section 2 however mentioned another strategy for ISE discovery, which is able to bypass the bandwidth limitations of the register file. In particular, the enumeration algorithm can look for candidates exceeding the processor's I/O constraint, and map them on the available ports by distributing register file accesses over more than one cycle. The new problem that arises is then to find a valid serialization for I/O between the processor and the custom functional unit, according to a given constraint on the number of register file accesses per cycle. Figure 7ac shows a dataflow graph with 3 inputs and 3 outputs, as well as two possible schedules for a register file with 2 read ports and 1 write port.

In Section 4.1 we will introduce I/O scheduling formally and analyze the complexity of existing solutions to this problem. We then prove the NP-completeness of the problem in Section 4.2.

4.1. Problem formalization

I/O scheduling was presented first in [10] and solved there using brute force. The solver enumerated exhaustively all possible schedules of the inputs, looking for the one which exhibited the smallest latency. This allows to minimize not only the latency of the ISE (i.e. the makespan of the schedule), but also the number of registers used.

On the other hand, the complexity of this approach is prohibitive. If N_{in} in the number of inputs in the ISE, and N_{read} is the number of register file read ports, the number of cases to be enumerated is

$$\begin{aligned}
&\binom{N_{\text{in}}}{N_{\text{read}}}\binom{N_{\text{in}}-N_{\text{read}}}{N_{\text{read}}}\cdots\binom{N_{\text{read}}}{N_{\text{read}}} \\
&= \frac{N_{\text{in}}!}{N_{\text{read}}!^{N_{\text{in}}/N_{\text{read}}}} \\
&= O\left(\frac{N_{\text{in}}!}{N_{\text{read}}{}^{N_{\text{in}}}}\right) = O\left(\left(\frac{N_{\text{in}}}{N_{\text{read}}}\right)^{N_{\text{in}}}\right)
\end{aligned} \tag{1}$$

Evaluating each of these cases is relatively cheap (linear in the number of nodes in the ISE), but exhaustive search clearly does not scale; for $N_{\text{in}} = 14$ and $N_{\text{read}} = 2$, the possible schedules are already half a billion, and 81 billion for $N_{\text{in}} = 16$.

In fact, this is the reason why Verma *et al.* propose a heuristic algorithm of polynomial complexity for this problem [11]. They formulate I/O scheduling as a matrix problem. The delays between the inputs and outputs of the circuit are embodied by a matrix A of *integral critical path delays* between inputs and outputs, and the number of registers is defined by two vectors R and C[4]:

$$r_i = \left\lfloor \frac{i}{N_{\text{read}}} \right\rfloor \qquad c_j = \left\lfloor \frac{j + N_{\text{write}} - 1}{N_{\text{write}}} \right\rfloor \tag{2}$$

Figure 7ab shows a dataflow graph for an ISE together with the corresponding matrix formulation of I/O scheduling. As in the picture, elements of A will be set to $-\infty$ in case there is no path between an input and an output.

The problem is then the following:

Problem 2 (I/O scheduling) *Given a maximum number of inputs and outputs that can be scheduled in any cycle (respectively N_{read} and N_{write}), let R and C be defined as in equation* (2)*. Then, given an $N_{\text{in}} \times N_{\text{out}}$ matrix A, find permutations π and σ respectively of $\{0, 1, \ldots, N_{\text{in}} - 1\}$ and $\{0, 1, \ldots, N_{\text{out}} - 1\}$, such that the following expression is minimized:*

$$\lambda = \max_{i,j}(r_{\pi_i} + a_{ij} + c_{\sigma_j}) \tag{3}$$

The outcome λ of the minimization is the latency of the resulting ISE; inputs will be scheduled at cycle r_{π_i} and outputs at cycle $\lambda - c_{\sigma_j}$. Figure 7c shows two schedules for the input data of Figure 7b, both numerically and graphically.

Verma reports that their polynomial solution to this problem always found the optimal latency for the cases in which brute-force search would terminate; however, they did not have a proof of optimality. In fact, in the remainder of this section we will prove the NP-completeness of problem 2.

[4] We assume 0-based indices in the rest of the paper.

4.2. NP-completeness proof

First of all, we prove that I/O scheduling is in NP. We then introduce the decision version of the problem:

Problem 3 (Decision version of I/O scheduling) *Given a maximum number of inputs and outputs that can be scheduled in any cycle (respectively N_{read} and N_{write}), let R and C be defined as in equation* (2)*. Then, given an $N_{\text{in}} \times N_{\text{out}}$ matrix A, and a latency λ, find whether or not there exist permutations π and σ respectively of $\{0, 1, \ldots, N_{\text{in}} - 1\}$ and $\{0, 1, \ldots, N_{\text{out}} - 1\}$, such that the following expression is true:*

$$\max_{i,j}(r_{\pi_i} + a_{ij} + c_{\sigma_j}) < \lambda \tag{4}$$

The two permutations π and σ are a certificate for problem 3. Furthermore, their size is $O(N_{\text{in}} + N_{\text{out}})$, while the size of the problem input is $O(N_{\text{in}} N_{\text{out}})$. Therefore, the problem admits a polynomial certificate and is in NP.

In order to prove the other direction, we reduce a particular flowshop scheduling problem to I/O scheduling. The scheduling problem we use is 2-machine flowshop with delays and unit job lengths (denoted shortly as *F2UD*), and has been proved to be strongly NP-complete by Yu [14].

Problem 4 (F2UD) *Given two machines M_1 and M_2, n jobs j ($j = 0, 1, \ldots, n-1$) whose execution takes 1 unit of time on M_1 and 1 unit of time on M_2, and a delay vector l_j, we define:*

- *t_{1j} as the time at which the first half of job j is scheduled. For any two jobs j and k, $t_{1j} \neq t_{1k}$.*
- *t_{2j} as the time at which the second half of job j is scheduled. For any two jobs j and k, $t_{2j} \neq t_{2k}$. Furthermore, for any job j, $t_{2j} \geq t_{1j} + l_j + 1$.*
- *$T_j = t_{2j} + 1$ as the completion time of job j.*

The problem is then to find a schedule for the jobs that minimizes the makespan

$$T = \max_j T_j \tag{5}$$

We reduce F2UD to I/O scheduling with $N_{\text{read}} = N_{\text{write}} = 1$. Thus, we prove strong NP-completeness of I/O scheduling *even for* $N_{\text{read}} = N_{\text{write}} = 1$. In this case, equation (3) simplifies to the following:

$$\lambda = \max_{i,j}(\pi_i + a_{ij} + \sigma_j) \tag{6}$$

because $r_i = i$ and $c_j = j$. Furthermore, we set $N_{\text{in}} = N_{\text{out}} = j$, $a_{jj} = l_j + 1$, and $a_{ij} = -\infty$ everywhere except on the main diagonal. This further reduces equation (6) to

$$\lambda = \max_i(\pi_i + l_i + 1 + \sigma_i) \tag{7}$$

Given π and σ that correspond to an optimal solution (i.e., to a minimal value of λ), we can derive the scheduling times at M_1 and M_2 from π and σ respectively, by setting i.e. $t_{1j} = \pi_j$ and $t_{2j} = \lambda - \sigma_j$. This is a valid solution for 2-machine flowshop scheduling, because

$$\begin{aligned} t_{2j} &= \lambda - \sigma_j \\ &= \max_i(\pi_i + l_i + 1 + \sigma_i) - \sigma_j \\ &\geq \pi_j + l_j + 1 + \sigma_j - \sigma_j \\ &= \pi_j + l_j + 1 = t_{1j} + l_j + 1 \end{aligned} \tag{8}$$

This solution has makespan $T = \lambda + 1$, and is also an optimal solution. Suppose there existed a solution of problem 4 with a makespan $T' < T$. We can assume without loss of generality that the t'_{2j} vector in the solution is a permutation of $\{T' - n, \ldots, T' - 2, T' - 1\}$—if this was not the case, it would be possible to shift the execution of the second half of one or more jobs in order to satisfy this condition. Likewise, we can assume that the t'_{1j} vector in the solution is a permutation of $\{0, 1, \ldots, n-1\}$, by anticipating the execution of some jobs on M_1 if this was not the case.

Then, by setting $\pi'_j = t'_{1j}$, and $\sigma'_j = T' - 1 - t'_{2j}$, we have a solution of I/O scheduling with latency:

$$\begin{aligned} \lambda' &= \max_j(\pi'_j + l_j + 1 + \sigma'_j) \\ &= \max_j(t'_{1j} + l_j + T' - t'_{2j}) \\ &= T' + \max_j(t'_{1j} + l_j - t'_{2j}) \\ &\leq T' - 1 < T - 1 = \lambda \end{aligned} \tag{9}$$

This implies $\lambda' < \lambda$, which contradicts the optimality of the I/O schedule given by π and σ.

5. Conclusion

In this paper, we analyzed the complexity of algorithms for compiling to customizable processors. In particular, we proved that a well-known algorithm for subgraph enumeration achieves time complexity that is polynomial in the size of the input graph (the lower bound for this problem); we also proved strong NP-completeness for the I/O scheduling problem, whose solution is important in order to use large instruction set extensions effectively.

In order to prove the latter result, we presented a flowshop formulation of I/O scheduling. Future work may include extending this formulation to more than two machines; this arises naturally when the application-specific functional unit includes other limited resources than communication bandwidth. Other possible research includes evaluating the quality of approximating algorithms for I/O scheduling, and devising branch-and-bound strategies to solve it exactly.

References

[1] K. Atasu, R. G. Dimond, O. Mencer, W. Luk, C. C. Özturan, and G. Dündar. Optimizing instruction-set extensible processors under data bandwidth constraints. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pages 588–593, Nice, France, Feb. 2007.

[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] R. Bennett, A. Murray, B. Franke, and N. Topham. Combining source-to-source transformations and processor instruction set extensions for the automated design-space exploration of embedded systems. pages 83–92. ACM Press New York, NY, USA, 2007.

[4] P. Bonzini and L. Pozzi. Code transformation strategies for extensible embedded processors. In *Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*, pages 242–52, Seoul, South Korea, Oct. 2006.

[5] P. Bonzini and L. Pozzi. Polynomial-time subgraph enumeration for automated instruction set extension. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pages 1331–36, Nice, France, Apr. 2007.

[6] X. Chen, D. L. Maskell, and Y. Sun. Fast identification of custom instructions for extensible processors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(2):359–68, Feb. 2007.

[7] J. Cong, Y. Fan, G. Han, and Z. 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., Feb. 2004.

[8] R. Gupta. Generalized dominators and post-dominators. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Language*, pages 246–257, New York, NY, USA, Nov. 1992. ACM Press.

[9] 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.

[10] 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.

[11] 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 125–134, Salzburg, Austria, Oct. 2007.

[12] 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.

[13] P. Yu and T. Mitra. Disjoint pattern enumeration for custom instructions identification. In *Proceedings of the 17th International Conference on Field-Programmable Logic and Applications*, pages 273–78, Amsterdam, Netherlands, Aug. 2007.

[14] W. Yu, H. Hoogeveen, and J. Lenstra. Minimizing Makespan in a Two-Machine Flow Shop with Delays and Unit-Time Operations is NP-Hard. *Journal of Scheduling*, 7(5):333–348, Oct. 2004.