ABSTRACT
In this paper, we present the first exact algorithm to solve the constrained I/O placement problem for FPGAs that support multiple I/O standards. We derive a compact integer linear programming formulation for the constrained I/O placement problem. The size of the integer linear program derived is independent of the number of I/O objects to be placed and hence is scalable to very large design instances. For example, for a Xilinx Virtex-E FPGA, the number of integer variables required is never more than 32 and is much smaller for practical design instances. Extensive experimental results using a non-commercial integer linear program solver shows that it only takes seconds to solve the resultant integer linear program in practice.

Categories and Subject Descriptors
B.7.2 [Integrated Circuits]: Design Aids; J.6 [Computer-Aided Engineering]: CAD

General Terms
Algorithms

Keywords
Placement, I/O placement, field-programmable gate array, I/O standards

1. INTRODUCTION
To keep pace with increasing clock speeds, higher data rates, and new low-voltage applications, a wide variety of new high-performance, low-voltage I/O standards have been introduced. Since FPGAs are commonly used in applications where they communicate with several devices and buses which may use different I/O standards, the newer generation of FPGAs are designed so that each device can support multiple I/O standards simultaneously.

Examples of FPGAs that support multiple I/O standards simultaneously include Xilinx’s Virtex[6], Virtex-E[7], Virtex-II[8], and Virtex-II Pro[9], and Altera’s APEX 20KE, APEX 20KC, and MAX 7000B[1]. All these FPGAs use a banked I/O organization such that certain restrictions exist as to which I/O standards can be mixed in the same I/O bank. Hence the user I/O objects must be placed carefully subject to the constraints imposed by the banked organization. We call this the constrained I/O placement problem.

The constrained I/O placement problem for FPGA designs with multiple I/O standards was previously considered in [2]. In [2], a heuristic algorithm to automate the I/O placement process was proposed and was reported to be being deployed in Xilinx’s placement tool. Unfortunately, there is no guarantee that the heuristic can find a feasible I/O placement solution even when feasible I/O placement solutions exist. And the user is required to pre-place the I/O objects manually in a feasible manner in the event that the heuristic fails to place them. Moreover, the method proposed in [2] is incapable of distinguishing those design instances that are inherently infeasible. So the user may try in vain to pre-place the I/O objects when the instance is actually infeasible.

In this paper, we present the first exact algorithm for the constrained I/O placement problem based on a compact integer linear programming formulation. It automatically determines if a problem instance is feasible or not. And it always produce a feasible I/O placement if the instance is feasible. We will use Xilinx Virtex-E devices as the example in this paper.

The rest of the paper is organized as follows. First, in Section 2.1, we describe the banked I/O organization and the constraints it imposes on I/O placement. Second, in Section 2.2, we show the shortcomings of a previously proposed heuristic approach for the constrained I/O placement problem and hence the need for an exact algorithm. Then, in Section 3, we present an exact approach based on an elegant integer linear programming formulation which can be efficiently solved. In Section 4, we suggest a new FPGA placement flow for FPGAs with multiple I/O standards. Experimental results are reported in Section 5. Finally, we conclude the paper in Section 6.

2. BACKGROUND

2.1 Banked I/O Organization

Different I/O standards differ in terms of their $V_{ref}$ voltage and $V_{ceo}$ voltage requirements. Firstly, some I/O standards require the use of a differential amplifier input and an external reference voltage, $V_{ref}$, must be provided to the amplifier. Secondly, some I/O standards require a specific supply voltage, $V_{ceo}$, to power either or both input and output blocks. Table 1 lists the requirements of the twenty I/O standards supported by Virtex-E FPGAs. There are two columns for $V_{ceo}$, one labeled “Output $V_{ceo}$” and the other “Input $V_{ceo}$” because some standards such as CTT only require a $V_{ceo}$ voltage for their output blocks but not for their input blocks.
the other hand, the $V_{ref}$ voltage (if any) of a standard is required by the input objects of the standard but not by the output objects of the standard. Naturally, bidirectional I/O objects of a standard have the combined requirements of the input objects and output objects of that standard. For example, since an input object of the CTT standard requires a $V_{ref}$ voltage of 1.50V but not a $V_{cco}$ voltage, and an output object of the same standard requires a $V_{cco}$ voltage of 3.3V but not a $V_{ref}$ voltage, so a bidirectional I/O object of the CTT standard requires a $V_{ref}$ voltage of 1.50V and a $V_{cco}$ voltage of 3.3V.

Table 1: $V_{ref}$ and $V_{cco}$ requirements of the 20 I/O standards supported by Virtex-E FPGAs.

<table>
<thead>
<tr>
<th>I/O standard</th>
<th>Output $V_{cco}$ (V)</th>
<th>Input $V_{cco}$ (V)</th>
<th>Input $V_{ref}$ (V)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LVTTL</td>
<td>3.3</td>
<td>3.3</td>
<td>N/A</td>
</tr>
<tr>
<td>LVC莫斯2</td>
<td>2.5</td>
<td>2.5</td>
<td>N/A</td>
</tr>
<tr>
<td>LVCMOS18</td>
<td>1.8</td>
<td>1.8</td>
<td>N/A</td>
</tr>
<tr>
<td>SSTL3.3K &amp; II</td>
<td>3.3</td>
<td>N/A</td>
<td>1.50</td>
</tr>
<tr>
<td>SSTL2.2K &amp; II</td>
<td>2.5</td>
<td>N/A</td>
<td>1.25</td>
</tr>
<tr>
<td>GTL</td>
<td>N/A</td>
<td>N/A</td>
<td>0.80</td>
</tr>
<tr>
<td>GTL+</td>
<td>N/A</td>
<td>N/A</td>
<td>1.0</td>
</tr>
<tr>
<td>HSTL I</td>
<td>1.5</td>
<td>N/A</td>
<td>0.75</td>
</tr>
<tr>
<td>HSTL III &amp; IV</td>
<td>1.5</td>
<td>N/A</td>
<td>0.90</td>
</tr>
<tr>
<td>CTT</td>
<td>3.3</td>
<td>N/A</td>
<td>1.50</td>
</tr>
<tr>
<td>AGP-2X</td>
<td>3.3</td>
<td>N/A</td>
<td>1.32</td>
</tr>
<tr>
<td>PCI33_3</td>
<td>3.3</td>
<td>3.3</td>
<td>N/A</td>
</tr>
<tr>
<td>PCI66_3</td>
<td>3.3</td>
<td>3.3</td>
<td>N/A</td>
</tr>
<tr>
<td>BLVDS &amp; LVDS</td>
<td>2.5</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>LVPECL</td>
<td>3.3</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

The commercial FPGAs that support multiple I/O standards[1, 6, 7, 8, 9] all use a banked I/O organization (see Fig. 1). The I/O blocks of a bank are served by a single $V_{ref}$ voltage and a single $V_{cco}$ voltage. Hence no two I/O objects that require differing $V_{ref}$ voltages or differing $V_{cco}$ voltages can be put in the same bank. For example, an output object of the LVTTL standard and an output object of the LVCMOS2 standard will create a conflict if they are put in the same bank because they require different voltages. On the other hand, I/O objects of different standards can be put in the same bank if there is no conflict in their $V_{ref}$ voltage and $V_{cco}$ voltage requirements. For example, output objects of the LVTTL standard, output objects of the CTT standard, and output objects of the GTL standard can be mixed in the same bank with the $V_{cco}$ voltage of the bank configured to 3.3V.

![Banked I/O organization](image)

Figure 1: Banked I/O organization.

The Xilinx FPGAs have one interesting architectural characteristic explained below. Besides the regular user I/O pins, each bank also has some $V_{cco}$ pins to receive the $V_{cco}$ voltage and some $V_{ref}$ pins to receive the $V_{ref}$ voltage required by the bank. If a bank does not need to access a $V_{ref}$ voltage (i.e., none of the I/O objects assigned to it requires a $V_{ref}$ voltage), the $V_{ref}$ pins in the bank can be used to accommodate user I/O signals instead. However, the $V_{cco}$ pins in a block are dedicated pins and can never be used to accommodate user I/O signals. We note that our algorithm to be presented in Section 3 is able to take advantage of the conditional usage of $V_{ref}$ pins to find a feasible I/O placement. For design instances where there is a feasible solution only if the conditional usage of $V_{ref}$ is taken into account, our algorithm is powerful enough to compute such a solution.

2.2 Previous Algorithm

Due to the various constraints described in Section 2.1, assigning user I/O objects to the I/O banks to obtain a feasible I/O placement is a non-trivial problem. This problem was first considered in [2] and a heuristic algorithm was proposed. Below we describe that heuristic and discuss its shortcomings.

First, simulated annealing is applied to place the I/O objects along with the logic blocks. The annealing process tries to minimize a cost function that is a weighted sum of the wirelength cost, the timing cost, and the banking violation cost. But the annealing process cannot ensure that the resultant I/O placement will be feasible. A weighted bipartite matching step is used to try to re-assign the I/O objects in the I/O banks to repair minor banking rule violations of the annealing placement solution. The weighted bipartite matching step attempts to perform I/O object re-assignment assuming that the $V_{ref}$ voltage and the $V_{cco}$ voltage of a bank must remain fixed to the the prevalent $V_{ref}$ voltage and the prevalent $V_{cco}$ voltage of the bank obtained at the end of the annealing process. (The prevalent $V_{ref}(V_{cco})$ voltage of a bank is defined as the $V_{ref}(V_{cco})$ voltage value required by the largest number of I/O objects assigned to the bank.) However, it is not guaranteed that this step can successfully re-assign the I/O objects in a feasible manner. Because there may not exist a feasible I/O placement where the $V_{ref}$ and $V_{cco}$ voltages of a bank are equal to the prevalent $V_{ref}$ and $V_{cco}$ voltages of the bank obtained at the end of the annealing process. (The prevalent $V_{ref}(V_{cco})$ voltage of a bank is defined as the $V_{ref}(V_{cco})$ voltage value required by the largest number of I/O objects assigned to the bank.) Hence the prevalent $V_{ref}$ voltage of both banks are $V_{cco}$. It is easy to see that there is no feasible I/O placement unless the prevalent voltages of the banks are changed, for example, as in Fig. 2(b).

(a) $A_1 A_1 A_2$  
(b) $A_1 A_1 A_2$  

Figure 2: (a) An infeasible I/O placement after simulated annealing. (b) A feasible I/O placement with the prevalent voltages changed.

Since bipartite matching can fail to obtain a feasible re-assignment, a greedy bin packing method is used in [2] as a last resort to re-assign the I/O objects. The I/O objects are sequentially assigned in decreasing order of packing difficulty. I/O objects that require both...
a $V_{ref}$ voltage and a $V_{cco}$ voltage are the most difficult, I/O objects that require either a $V_{ref}$ voltage or a $V_{cco}$ voltage but not both are less difficult, lastly I/O objects that neither require a $V_{ref}$ voltage nor a $V_{cco}$ voltage are the least difficult.) When an I/O object is assigned, it is put into the bank with the highest affinity to it such that it does not conflict with other I/O objects already assigned to the bank. The affinity of a bank to an I/O object $x$ is defined as the sum of the number of I/O objects with the same $V_{ref}$ voltage as $x$ and the number of I/O objects with the same $V_{cco}$ voltage as $x$ in the infeasible placement produced by simulated annealing. Unfortunately, this last resort can still fail to yield a feasible I/O placement. One example is shown in Fig. 3. Assume that there are only two banks and the capacity of each bank is six. Suppose that I/O objects of class $A_1$ require a $V_{cco}$ voltage equals to $V_{C1}$ but do not require a $V_{ref}$ voltage. And I/O objects of class $B_i$ ($i = 1, 2$) require a $V_{ref}$ voltage equals to $V_{Ri}$ but do not require a $V_{cco}$ voltage. Hence all I/O objects have the same level of difficulty. Assume that I/O objects of class $A_1$ are assigned first, followed by I/O objects of class $B_1$, and then I/O objects of class $B_2$. First, we assign the I/O objects of class $A_1$, since $affinity_{A_1, \text{bank}1} = 4 > affinity_{A_1, \text{bank}2} = 0$, all four I/O objects of class $A_1$ are assigned to bank 1. Next we assign the I/O objects of class $B_1$, since $affinity_{B_1, \text{bank}1} = 2 > affinity_{B_1, \text{bank}2} = 1$, all three I/O objects of class $B_1$ are assigned to bank 2. Finally, we assign the I/O objects of class $B_2$. Though $affinity_{B_2, \text{bank}1} = 3 > affinity_{B_1, \text{bank}2} = 0$, assigning the class $B_2$ I/O objects to bank 1 will conflict with the class $B_1$ I/O objects already in bank 2, so we must assign the class $B_2$ I/O objects to bank 1. However, we are stuck after assigning two class $B_1$ I/O objects to bank 1 since bank 1 has become full (see Fig. 3(b)). It is not difficult to see that a feasible I/O placement (see Fig. 3(c)) does exist for the given instance though the greedy bin packing procedure has failed.

(a) $A_1A_1A_1A_1B_1$ \hspace{2cm} $B_1B_1B_1B_1B_2B_2$

(b) $A_1A_1A_1A_1$ \hspace{2cm} $B_1B_1B_1$ \hspace{2cm} $A_1A_1A_1$ \hspace{2cm} $B_1B_1B_1$ \hspace{2cm} $A_1A_1A_1A_2B_2$

(c) $A_1A_1A_1B_1B_1B_1$ \hspace{2cm} $A_1B_2B_2B_2$

Figure 3: (a) Result after simulated annealing. (b) Reassignment by the greedy bin packing procedure where one class $B_2$ object cannot be assigned anywhere. (c) A feasible I/O placement.

From the above discussion, it is clear that the heuristic approach proposed in [2] can easily fail to produce a feasible I/O placement even after going through two fixing procedures. Moreover, the heuristic method is incapable of distinguishing infeasible problem instances from feasible ones. In the next section, we present an exact method that guarantees to produce a feasible I/O placement if the given instance is feasible. In the case that the given instance is infeasible, it will give a “proof” that it is indeed infeasible.

### 3. A COMPACT ILP FORMULATION

It is well known that integer linear programming (ILP) is a versatile approach for solving combinatorial optimization problems. However, it is viable in practice only when the number of integer variables is small or when the integer linear program has some special structures so that it can be solved efficiently by some specialized method. Very often different ILP formulations can be found for the same problem, some of which are more efficiently solvable than others. Finding the best ILP formulation for a problem is not always straightforward. In this section, we present a compact ILP formulation for the constrained I/O placement problem. We will use the Virtex-E FPGA family as an example. Based on a number of observations, we are able to come up with a formulation that requires no more than 32 integer variables (the number is much smaller for most constrained I/O placement problem instances that occur in practice). An integer linear program of such size can be solved very quickly.

Perhaps the most straightforward ILP formulation for the constrained I/O placement problem is a 0-1 integer linear program defined as follows. First, define a 0-1 integer variable $y_{ik}$ for each I/O object $i$ and bank $k$ where $y_{ik} = 1$ if and only if I/O object $i$ is assigned to bank $k$. Second, formulate the capacity constraint for each bank (number of I/O objects assigned to a bank must not exceed the capacity of the bank), the assignment constraint for each I/O object (an I/O object must be assigned to exactly one bank), and the compatibility constraints of the I/O objects (I/O objects with incompatible $V_{cco}$ or $V_{ref}$ voltages cannot be assigned to the same bank), in terms of $y_{ik}$'s. Then the I/O placement problem is feasible if and only if the corresponding ILP is feasible. (Alternatively, we can modify the assignment constraint such that each I/O object must be assigned to at most one bank, and try to maximize the number of I/O objects that are assigned.) However, since there are hundreds of I/O objects and eight banks, this ILP formulation requires thousands of integer variables which is not computationally feasible to solve. In the following, we will present a completely different ILP formulation which is much more elegant and compact by making a number of important observations.

First, notice that we can classify the I/O objects into four types. (i) I/O objects that require a $V_{cco}$ voltage but not a $V_{ref}$ voltage. (ii) I/O objects that require a $V_{ref}$ voltage but not a $V_{cco}$ voltage. (iii) I/O objects that require both a $V_{cco}$ voltage and a $V_{ref}$ voltage. (iv) I/O objects that require neither a $V_{cco}$ voltage nor a $V_{ref}$ voltage. The classification of the I/O objects supported by the Virtex-E FPGAs is shown in Table 2.

<p>| Table 2: Classification of I/O objects supported by Virtex-E FPGAs. |
|------------------|------|------|------------------|</p>
<table>
<thead>
<tr>
<th>I/O object</th>
<th>Input</th>
<th>Output</th>
<th>Bidirectional</th>
</tr>
</thead>
<tbody>
<tr>
<td>LVCMOS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LVCMOS2</td>
<td>i</td>
<td>i</td>
<td>i</td>
</tr>
<tr>
<td>LVCMOS18</td>
<td>i</td>
<td>i</td>
<td>i</td>
</tr>
<tr>
<td>SSTL3 I &amp; II</td>
<td>ii</td>
<td>i</td>
<td>iii</td>
</tr>
<tr>
<td>SSTL2 I &amp; II</td>
<td>ii</td>
<td>i</td>
<td>iii</td>
</tr>
<tr>
<td>GTL</td>
<td>ii</td>
<td>iv</td>
<td>ii</td>
</tr>
<tr>
<td>GTL+</td>
<td>ii</td>
<td>iv</td>
<td>ii</td>
</tr>
<tr>
<td>HSTL I</td>
<td>ii</td>
<td>i</td>
<td>iii</td>
</tr>
<tr>
<td>HSTL II</td>
<td>ii</td>
<td>i</td>
<td>iii</td>
</tr>
<tr>
<td>CTT</td>
<td>ii</td>
<td>i</td>
<td>iii</td>
</tr>
<tr>
<td>AGP-2X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PCI33</td>
<td>i</td>
<td>i</td>
<td></td>
</tr>
<tr>
<td>PCI66</td>
<td>i</td>
<td>i</td>
<td></td>
</tr>
<tr>
<td>BLVDS &amp; LVDS</td>
<td>iv</td>
<td>i</td>
<td></td>
</tr>
<tr>
<td>LVPECL</td>
<td>iv</td>
<td>i</td>
<td></td>
</tr>
</tbody>
</table>

From the above discussion, it is clear that the heuristic approach proposed in [2] can easily fail to produce a feasible I/O placement even after going through two fixing procedures. Moreover, the heuristic method is incapable of distinguishing infeasible problem instances from feasible ones. In the next section, we present an exact method that guarantees to produce a feasible I/O placement if the given instance is feasible. In the case that the given instance is infeasible, it will give a “proof” that it is indeed infeasible.
Furthermore, since there are four possible \(V_{cco}\) voltage levels (see Table 1), we can divide the I/O objects of type (i) into four equivalence classes \(A_1, A_2, A_3,\) and \(A_4.\) We denote the \(V_{cco}\) voltage used by \(A_i\) as \(V_{C_i}(i = 1, \ldots, 4).\) Similarly, since there are seven possible \(V_{ref}\) voltage levels (see Table 1), we can divide the I/O objects of type (ii) into seven equivalence classes \(B_1\) to \(B_7.\) We denote the \(V_{ref}\) voltage used by \(B_j\) as \(V_{R_j}(j = 1, \ldots, 7).\) Hence, we can divide the I/O objects of type (iii) which require both a \(V_{cco}\) voltage and a \(V_{ref}\) voltage, there are 28 (= \(4 \times 7\)) possible combinations since there are four possible \(V_{cco}\) voltage levels and seven possible \(V_{ref}\) voltage levels. Hence, we can divide the I/O objects of type (iii) into 28 equivalence classes as \(\text{cco}_{i j}\) where \(C_{ij}\) uses the same \(V_{cco}\) voltage as \(A_i\) and the same \(V_{ref}\) voltage as \(B_j.\) However, we note that out of these 28 equivalence classes, all except five are always empty since only five combinations of \(V_{cco}\) and \(V_{ref}\) can occur in practice by checking Table 1. We denote the set of the five valid \((i, j)\) index pairs as \(F.\) Finally, we define a single equivalence class \(D\) for all I/O objects of type (iv). So there are a total of 17 possible equivalence classes of I/O objects. We say that a class \(A_i\) is compatible with a class \(A_j\) if I/O objects of class \(A_i\) and I/O objects of class \(A_j\) can be mixed in the same I/O bank (i.e., they do not have any \(V_{cco}\) or \(V_{ref}\) voltage requirement conflict). It is easy to see that the compatibility relation is reflexive and symmetric. But note that it is not transitive. For example, class \(A_1\) is compatible with class \(D,\) and class \(D\) is compatible with class \(A_2,\) but class \(A_1\) is not compatible with class \(A_2.\) Finally, since the compatibility relation is symmetric, it is legitimate to say “two classes are compatible” or “two classes are incompatible”. The following lemma summarizes for all distinct pairs of classes if they are compatible or not.

**Lemma 1. (Class Compatibility)**

(a) Classes \(A_{i_1}\) and \(A_{i_2}\) are incompatible for all \(i_1 \neq i_2.\)

(b) Classes \(B_{j_1}\) and \(B_{j_2}\) are incompatible for all \(j_1 \neq j_2.\)

(c) Classes \(A_i\) and \(B_j\) are compatible for all \(i = 1, \ldots, 4\) and \(j = 1, \ldots, 7.\)

(d) Classes \(A_{i_1}\) and \(C_{i_1 j}\) are compatible if and only if \(i_1 = j = i_2.\)

(e) Classes \(B_{j_1}\) and \(C_{i_1 j}\) are compatible if and only if \(j_1 = j = j_2.\)

(f) Class \(D\) is compatible with any other class.

**Proof:**

(a) Suppose \(i_1 \neq i_2,\) I/O objects of class \(A_{i_1}\) and I/O objects of class \(A_{i_2}\) require two distinct \(V_{cco}\) voltages, hence cannot be put in the same bank.

(b) Suppose \(j_1 \neq j_2,\) I/O objects of class \(B_{j_1}\) and I/O objects of class \(B_{j_2}\) require two distinct \(V_{ref}\) voltages, hence cannot be put in the same bank.

(c) Since I/O objects of class \(A_i\) only require a \(V_{cco}\) voltage but not a \(V_{ref}\) voltage, while I/O objects of class \(B_j\) only require a \(V_{ref}\) voltage but not a \(V_{cco}\) voltage, they do not have any conflicting voltage requirement and can be put into the same bank.

(d) I/O objects of class \(A_i\) use a \(V_{cco}\) voltage equals to \(V_{C_i}\) and no \(V_{ref}\) voltage. I/O objects of class \(C_{ij}\) use a \(V_{cco}\) voltage equals to \(V_{C_i}\). So they cannot have any \(V_{ref}\) voltage requirement conflict. Finally, they do not have any \(V_{cco}\) voltage requirement conflict if and only if \(V_{C_i} = V_{C_j}, i.e., i_1 = j_2.\) Hence the result.

(e) I/O objects of class \(B_{j_1}\) use a \(V_{ref}\) voltage equals to \(V_{R_j}\) and no \(V_{cco}\) voltage. I/O objects of class \(C_{ij}\) use a \(V_{ref}\) voltage equals to \(V_{R_j}\). So they cannot have any \(V_{cco}\) voltage requirement conflict. Finally, they do not have any \(V_{ref}\) voltage requirement conflict if and only if \(V_{R_j} = V_{R_j}, i.e., j_1 = j_2.\) Hence the result.

(f) Since I/O objects of class \(D\) neither require a \(V_{cco}\) voltage nor a \(V_{ref}\) voltage, they will not cause any conflict in voltage requirement with I/O objects of any other class.

\(\square\)

Now we are ready to present a compact linear programming formulation for the constrained I/O placement problem. One way to look at the problem is as follows. An I/O bank can be configured with at most one \(V_{cco}\) voltage and at most one \(V_{ref}\) voltage. Depending on the \(V_{cco}\) and \(V_{ref}\) voltage assignments, some classes of I/O objects can be put in the bank while other classes cannot. We say that a bank is of type \(T_{ij}\) if it is configured with a \(V_{cco}\) voltage equals to \(V_{C_i}\) and a \(V_{ref}\) voltage equals to \(V_{R_j}.\) It is clear that only I/O objects of classes \(A_i, B_j, C_{ij},\) and \(D\) can be put into a bank of type \(T_{ij}\) while I/O objects of any other class cannot. The capacity of a type \(T_{ij}\) bank is \(U\) where \(U\) is the number of regular user I/O pins in a bank. We need to distinguish the banks that are not configured with a \(V_{ref}\) voltage, since the \(V_{ref}\) pins of these banks can be used to accommodate I/O objects as noted in Section 2.1. We say that a bank is of type \(T_{ij}\) if it is configured with a \(V_{cco}\) voltage equals to \(V_{C_i}\) but with no \(V_{ref}\) voltage. The capacity of such a bank is \(U + R\) where \(U\) is the number of regular user I/O pins in the bank and \(R\) is the number of \(V_{ref}\) pins in the bank. It is clear that only I/O objects of classes \(A_i, B_j, C_{ij},\) and \(D\) can be put into a bank of type \(T_{ij}\). On the other hand, we do not have to specifically distinguish the banks that are not configured with a \(V_{cco}\) voltage. Since a bank \(\alpha\) configured with a \(V_{ref}\) voltage equal to \(V_{R_j}\) but with no \(V_{cco}\) voltage can always be replaced by a bank \(\beta\) configured with a \(V_{cco}\) voltage equal to \(V_{C_i}\) and some arbitrary \(V_{cco}\) voltage (bank \(\beta\) has the same capacity as bank \(\alpha\) and every I/O object allowable in bank \(\alpha\) is also allowable in bank \(\beta\)). Hence, we have to determine how many banks of each type \(T_{ij}\) \((i = 1, \ldots, 4; j = 1, \ldots, 7)\) is needed in order to minimize the total number of banks required to accommodate all the I/O objects of a design instance. A design instance is feasible if and only if the minimum number of banks required is no more than 8 which is the number of I/O banks available on a device.

The parameters of the problem are defined as follows:

\[
\begin{align*}
U &= \text{number of regular user I/O pins per bank.} \\
R &= \text{number of } V_{ref} \text{ pins per bank.} \\
|A_i| &= \text{no. of class } A_i \text{ I/O objects } (i = 1, \ldots, 4). \\
|B_j| &= \text{no. of class } B_j \text{ I/O objects } (j = 1, \ldots, 7). \\
|C_{ij}| &= \text{no. of class } C_{ij} \text{ I/O objects } ((i, j) \in F). \\
|D| &= \text{number of class } D \text{ I/O objects.}
\end{align*}
\]

Given a constrained I/O pin placement problem instance, the values of \(U, R, |A_i|'s, |B_j|'s, |C_{ij}|'s, \text{ and } |D|\) are all known.
The variables whose values need to be determined are:

\[ a_i = \text{no. of class } A_i \text{ I/O objects assigned to a bank of type } T_i (i = 1, \ldots, 4), \]
\[ a_{ij} = \text{no. of class } A_i \text{ I/O objects assigned to a bank of type } T_{ij} (i = 1, \ldots, 4; j = 1, \ldots, 7), \]
\[ b_{ij} = \text{no. of class } B_j \text{ I/O objects assigned to a bank of type } T_{ij} (i = 1, \ldots, 4; j = 1, \ldots, 7), \]
\[ d_i = \text{no. of class } D \text{ I/O objects assigned to a bank of type } T_i (i = 1, \ldots, 4), \]
\[ d_{ij} = \text{no. of class } D \text{ I/O objects assigned to a bank of type } T_{ij} (i = 1, \ldots, 4; j = 1, \ldots, 7), \]
\[ x_i = \text{no. of type } T_i \text{ banks (} i = 1, \ldots, 4), \]
\[ x_{ij} = \text{no. of type } T_{ij} \text{ banks (} i = 1, \ldots, 4; j = 1, \ldots, 7). \]

Notice that all class \( C_{ij} \) I/O objects must be assigned to a bank of type \( T_{ij} \), hence the number of class \( C_{ij} \) I/O objects assigned to a bank of type \( T_{ij} \) is always equal to \( |C_{ij}| (i = 1, \ldots, 4; j = 1, \ldots, 7) \). The complete integer linear program is presented below.

\[ \text{Minimize} \quad \sum_{i=1}^{4} x_i + \sum_{i=1}^{4} \sum_{j=1}^{7} x_{ij} \]

subject to:

\[ a_i + \sum_{j=1}^{4} a_{ij} = |A_i| \quad \forall i = 1, \ldots, 4, \]
\[ \sum_{i=1}^{4} b_{ij} = |B_j| \quad \forall j = 1, \ldots, 7, \]
\[ \sum_{i=1}^{4} d_i + \sum_{i=1}^{4} \sum_{j=1}^{7} d_{ij} = |D|, \]
\[ a_i + d_i \leq (U + R)x_i \quad \forall i = 1, \ldots, 4, \]
\[ a_{ij} + b_{ij} + |C_{ij}| + d_{ij} \leq UX_{ij} \quad \forall i = 1, \ldots, 4; j = 1, \ldots, 7, \]
\[ a_{ij}, b_{ij}, d_i, d_{ij}, x_i, x_{ij} \geq 0 \]
\[ x_i, x_{ij} \text{ are integers.} \]

The value of the objective function is equal to the total number of banks used. The first constraint expresses the fact that class \( A_i \) I/O objects can be assigned to banks of types \( T_1, T_2, \ldots, T_7 \) only. The second constraint expresses the fact that class \( B_j \) I/O objects can be assigned to banks of types \( T_1, \ldots, T_7 \). The third constraint expresses the fact that class \( D \) I/O objects can be assigned to any type. The fourth constraint expresses the fact that the number of I/O objects assigned to banks of type \( T_l \) must be less than or equal to the number of type \( T_l \) banks times the number of usable I/O pins in a type \( T_l \) bank. The fifth constraint expresses the fact that the number of I/O objects assigned to banks of type \( T_{ij} \) must be less than or equal to the number of type \( T_{ij} \) banks times the number of usable I/O pins in a type \( T_{ij} \) bank. The given I/O placement problem instance is feasible if and only if ILP \( M \) is feasible and has an optimal objective function value less than or equal to 8.

We note that we have deliberately left out the integral constraints for \( a_i, a_{ij}, b_{ij}, d_i, d_{ij} \) in our integer program formulation. In the following lemma, we prove that we can still obtain an optimal solution for which all \( a_i, a_{ij}, b_{ij}, d_i, d_{ij}, x_i, x_{ij} \) are integers by imposing integer constraints on \( x_i, x_{ij} \) only. This observation is an important one since it greatly reduces the number of integer variables in our ILP formulation.

**Lemma 2.** The ILP \( M \) always has an all integral optimal solution (i.e., all \( a_i, a_{ij}, b_{ij}, d_i, d_{ij}, x_i, x_{ij} \) are integral).

**Proof:** First, we show that the ILP is always feasible. It is easy to see that we can always assign all I/O objects into banks without any \( V_{CCP} \) or \( V_{CCP} \) voltage conflict by using a sufficiently large number of banks. For example, we can assign all I/O objects of class \( A_1 \) to banks of type \( T_1 \) (i.e., \( a_1 = |A_1| \)) and \( a_{ij} = 0 \) for \( j = 1, \ldots, 7 \), assign all I/O objects of class \( B_j \) to banks of type \( T_1 \) (i.e., \( b_{ij} = |B_j| \)), and \( b_{ij} = 0 \) for \( i \neq 1 \), assign all I/O objects of class \( D \) to banks of type \( T_1 \) (i.e., \( d_i = |D| \)), and \( d_{ij} = 0 \) for \( i \neq 1 \), and \( d_{ij} = 0 \) for \( i = 1, \ldots, 4; j = 1, \ldots, 7 \). Then we need \( \left( |A_1| + |D| \right) \) banks of type \( T_1 \) (i.e., \( x_1 = \left( |A_1| + |D| \right) \)), \( b_{ij} = \left( B_j \right) \) banks of type \( T_1 \) for \( i \neq 1 \), \( b_{ij} = \left( B_j \right) \) banks of type \( T_1 \) for \( i \neq 1 \), and no bank of type \( T_1 \) for \( i = 1 \) (i.e., \( x_i = 0 \) for \( i = 1 \)). Hence the ILP \( M \) is always feasible.

Second, we show that there always exists an all integral optimal solution. Suppose the ILP has an optimal solution for which \( x_i = X_i \) and \( x_{ij} = X_{ij} \), where \( X_i \) and \( X_{ij} \) denote some constants. It is clear that \( X_i \) and \( X_{ij} \) are some integer constants because of the presence of the integer constraints on \( x_i, x_{ij} \) in ILP \( M \). Hence we have

\[ a_i + \sum_{j=1}^{4} a_{ij} = |A_i| \quad \forall i = 1, \ldots, 4, \]
\[ b_{ij} = |B_j| \quad \forall j = 1, \ldots, 7, \]
\[ d_i + \sum_{i=1}^{4} \sum_{j=1}^{7} d_{ij} = |D|, \]
\[ a_i + d_i \leq (U + R)X_i \quad \forall i = 1, \ldots, 4, \]
\[ a_{ij} + b_{ij} + d_{ij} \leq UX_{ij} - |C_{ij}| \quad \forall i = 1, \ldots, 4; j = 1, \ldots, 7. \]

The above can be viewed as a transportation problem[5]. In particular, the transportation problem has source nodes \( T_i \) (i = 1, \ldots, 4) and \( T_{ij} \) (i = 1, \ldots, 4; j = 1, \ldots, 7), and sink nodes \( A_i \) (i = 1, \ldots, 4), \( B_j \) (j = 1, \ldots, 7), and \( D \). The supply at source node \( T_i \) is \((U + R)X_i \) (i = 1, \ldots, 4) and the supply at source node \( T_{ij} \) is \( UX_{ij} - |C_{ij}| \) (i = 1, \ldots, 4; j = 1, \ldots, 7). The demand at sink node \( A_i \) is \( |A_i| \) (i = 1, \ldots, 4), the demand at sink node \( B_j \) is \( |B_j| \) (j = 1, \ldots, 7), and the demand at sink node \( D \) is \( |D| \). Shipments are possible only from \( T_i \) to \( A_i \) or \( D \) (i = 1, \ldots, 4), and from \( T_{ij} \) to \( A_i \) or \( B_j \) or \( D \) (i = 1, \ldots, 4; j = 1, \ldots, 7). The transportation problem is to determine a feasible way of shipment to satisfy the demands of the sink nodes by the supplies of the source nodes.1 The variable \( a_i \) denotes the amount of shipment from \( T_i \) to \( A_i \) (i = 1, \ldots, 4). \( a_{ij} \) denotes the amount of shipment from \( T_{ij} \) to \( A_i \) (i = 1, \ldots, 4; j = 1, \ldots, 7). \( b_{ij} \) denotes the amount of shipment from \( T_{ij} \) to \( B_j \) (i = 1, \ldots, 4; j = 1, \ldots, 7). \( d_i \) denotes the amount of shipment from \( T_i \) to \( D \) (i = 1, \ldots, 4). \( d_{ij} \) denotes the amount of shipment from \( T_{ij} \) to \( D \) (i = 1, \ldots, 4; j = 1, \ldots, 7). It is well known that for any transportation problem instance with integral supplies at the source nodes and integral demands at the sink nodes, there is always an integer feasible solution (i.e., all \( a_i, a_{ij}, b_{ij}, d_i, d_{ij} \) are integers) if the instance is feasible. Hence the desired result follows.

---

1The general transportation problem is to determine a feasible way of shipment incurring the smallest transportation cost when shipment between different pairs of source and sink have different unit transportation costs. Here we simply assume that the unit transportation cost between a pair of source and sink is 1 if shipment between them is allowed, or is \( \infty \) otherwise.
We have a few remarks about our ILP formulation.

1. The size of ILP $\mathcal{M}$ is independent of the number of I/O objects to be placed or the number of banks in the FPGA.

2. The maximum number of integer variables required by our formulation is $32$ (4 $x_i$’s and 28 $x_{ij}$’s) which occurs only when a design instance uses all four distinct $V_{cco}$ voltages and seven distinct $V_{ref}$ voltages. For a design instance that uses $I(\leq 4)$ distinct $V_{cco}$ voltages and $J(\leq 7)$ distinct $V_{ref}$ voltages, the actual number of integer variables required is $I + I \cdot J(\leq 32)$ and the number of constraints (excluding the non-negativity constraints and integer constraints) is $2I + J + I \cdot J(\leq 44)$.

3. We are able to model the conditional usage of the $V_{ref}$ pins in our ILP approach succinctly. However, handling the conditional usage of the $V_{ref}$ pins is likely to be a challenge for other approaches.

4. The proof of Lemma 2 suggests that solving the ILP $\mathcal{M}$ by any ILP solver based on the simplex algorithm [4] will always yield an integer optimum solution since it has the structure of a transportation problem.

5. The proof of Lemma 2 also suggests that instead of using a general ILP solver to solve the ILP $\mathcal{M}$, we may develop a specialized branch and bound algorithm that employs a transportation problem procedure to solve it.

If the optimal objective function value of $\mathcal{M}$ is greater than 8, then there is no feasible solution to the given constrained I/O placement problem instance. Otherwise, let $a_i^*, a_{ij}, b_{ij}, d_i, d_{ij}, x_i^*, x_{ij}$ be an optimal solution of $\mathcal{M}$. A feasible I/O placement can be obtained by arbitrarily selecting $x_i^*$ banks on the FPGA to be configured to type $T_i$ and $x_{ij}$ banks on the FPGA to be configured to type $T_{ij}$, and then arbitrarily select $a_i^*$ of the class $A_i$ objects to assign to the I/O banks configured to type $T_i$ and select $a_{ij}$ of the class $A_{ij}$ objects to assign to the I/O banks configured to type $T_{ij}$, etc.

4. PLACEMENT OF CORE LOGIC AND I/O OBJECTS

[2] suggested the following placement flow for FPGAs with multiple I/O standards. First, the core logic and the I/O objects are placed simultaneously by applying simulated annealing. Then a weighted bipartite matching procedure or a greedy bin packing procedure is applied to re-place the I/O objects to fix the I/O banking rule violations if possible. When both procedures fail, the user needs to manually re-place the I/O objects. The shortcoming of this flow is that it spends a large amount of effort in the simulated annealing process to optimize the timing/wirelength only to yield an infeasible solution most of the time. The timing/wirelength will unavoidably be increased when the I/O objects are re-placed.

Here we propose an alternative flow. In our flow, a feasible placement of I/O objects is computed first to ensure feasibility, and then it works towards optimality in timing/wirelength. Note that our constrained I/O placement algorithm determines how many banks of each type $T_i/T_{ij}$ are needed, but there is freedom in choosing which banks on the FPGA are configured to type $T_i/T_{ij}$. There is also freedom in permuting the placement of I/O objects within a bank. So we may first obtain an initial feasible I/O placement as described at the end of Section 3. Then we can apply simulated annealing to place the core logic and re-distribute the I/O objects at the same time for wirelength and/or timing optimization. We only need to ensure that the I/O objects are re-distributed in such a way that no I/O banking rule violation will be introduced in the process. For example, the following types of move involving the I/O objects can be used. (1) Swapping any two banks of I/O objects. (2) Exchanging the positions of any two I/O objects within the same bank. (3) Moving an I/O object to a vacant position within the same bank. (4) Exchanging the positions of any two I/O objects of the same class. Compared with [2], our search space is much smaller and only consists of feasible placement solutions, so no time is wasted in searching through infeasible placement solutions. Finally, a weighted bipartite matching procedure similar to that in [2] can be applied after simulated annealing to further optimize the wirelength and/or timing without changing the prevalent $V_{cco}$ and $V_{ref}$ voltages of each bank.

5. EXPERIMENTAL RESULTS

In order to verify the effectiveness of our integer linear program approach to the constrained I/O placement problem, we tested it extensively on a large number of large test cases. We used a non-commercial ILP solver[3]2 to solve the integer linear programs in our experiments.

We note that in general the difficulty of a constrained I/O placement instance increases with the number of I/O objects to be placed, the ratio of the number of I/O objects to the I/O capacity of the FPGA, and the numbers of distinct $V_{cco}$ voltages and distinct $V_{ref}$ voltages used. In order to thoroughly test our algorithm under high stress conditions, we generated test cases with between 640 and 800 I/O objects and assumed that they are placed on a FPGA with 100 regular user I/O pins and 16 $V_{ref}$ pins per bank. Moreover, to make the test cases as difficult as possible we assumed that all I/O objects must require a $V_{cco}$ voltage or a $V_{ref}$ voltage or both, since I/O objects that require neither a $V_{cco}$ voltage nor a $V_{ref}$ voltage are easy to place.

We generated a hundred instances that use two distinct $V_{cco}$ voltages and two distinct $V_{ref}$ voltages each. Then we generated another hundred more difficult instances that use three distinct $V_{cco}$ voltages and four distinct $V_{ref}$ voltages each. Lastly, for theoretical interest we also generated a hundred extremely difficult instances that use four distinct $V_{cco}$ voltages and seven distinct $V_{ref}$ voltages each. The experiments were conducted on a 1.8GHz Pentium 4. We report both the average running time and the maximum running time for the three batches in Table 3. The numbers of feasible and infeasible instances in each batch are reported in the last column of the table. We can see that for practical design instances (batches 1 and 2), the banked I/O organization usually does not preclude a feasible I/O placement solution. For practical design instances, our approach takes no more than a few seconds to find a feasible I/O placement or determine that no feasible I/O placement exists. As expected, instances that use more distinct $V_{ref}$ voltages and $V_{cco}$ voltages are more likely to be infeasible. The majority of the instances in batch 3 are infeasible.

<table>
<thead>
<tr>
<th>Batch</th>
<th>#I/O</th>
<th>Avg. time (s)</th>
<th>Max. time (s)</th>
<th>#feasible</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>22</td>
<td>0.01</td>
<td>0.01</td>
<td>98.2</td>
</tr>
<tr>
<td>2</td>
<td>34</td>
<td>0.22</td>
<td>2.46</td>
<td>76.24</td>
</tr>
<tr>
<td>3</td>
<td>47</td>
<td>130.79</td>
<td>1929.66</td>
<td>9.91</td>
</tr>
</tbody>
</table>

One advantage of our ILP formulation for the constrained I/O placement problem is that it can be solved using a commercial ILP solver. However, it may not scale well for very large instances.

---

2[3] is based on the simplex algorithm.
placement problem is that its size is independent of the number of I/O objects to be placed. Hence it can be applied even for very large designs. We did a second experiment in which we generated test cases with between 1280 and 1600 I/O objects and assumed that they are placed on a FPGA with 200 regular user I/O pins and 32 $V_{ref}$ pins per bank. Again we generated three batches of test cases as in the first experiment. The results are reported in Table 4. We can see that though the problem size has been doubled compared to the first experiment, the running time is virtually unaffected.

| Batch 4 | 2/2 | 0.01 | 0.02 | 1000:0 |
| Batch 5 | 3/4 | 0.20 | 2.01 | 74:26  |
| Batch 6 | 4/7 | 106.33 | 1509.33 | 12:88  |

We note that we were unable to compare with the experimental results in [2] because we did not have access to the five circuits used in [2] which were from Xilinx’s customers.

6. CONCLUSIONS

We presented an elegant integer linear programming formulation for the constrained I/O placement problem for FPGAs that support multiple I/O standards. Unlike the method in [2], ours can automatically determine if a design instance is feasible or not. And it always computes a feasible I/O placement when an instance is feasible. Moreover, the numbers of variables and constraints of our integer linear programming formulation are independent of the number of I/O objects to be placed, hence it can comfortably handle very large designs. Extensive experimental results confirmed its efficiency in practice.

Acknowledgement

We would like to thank Jason Anderson for answering our inquiry about the experimental results in [2].

7. REFERENCES