Area-Efficient Instruction Set Synthesis for Reconfigurable System-on-Chip Designs

Philip Brisk  
Computer Science Department  
University of California, Los Angeles  
Westwood, CA, 90095  
philip@cs.ucla.edu

Adam Kaplan  
Computer Science Department  
University of California, Los Angeles  
Westwood, CA, 90095  
kaplan@cs.ucla.edu

Majid Sarrafzadeh  
Computer Science Department  
University of California, Los Angeles  
Westwood, CA, 90095  
majid@cs.ucla.edu

ABSTRACT
Silicon compilers are often used in conjunction with Field Programmable Gate Arrays (FPGAs) to deliver flexibility, fast prototyping, and accelerated time-to-market. Many of these compilers produce hardware that is larger than necessary, as they do not allow instructions to share hardware resources. This study presents an efficient heuristic which transforms a set of custom instructions into a single hardware datapath on which they can execute. Our approach is based on the classic problems of finding the longest common subsequence and substring of two (or more) sequences. This heuristic produces circuits which are as much as 85.33% smaller than those synthesized by integer linear programming (ILP) approaches which do not explore resource sharing. On average, we obtained 55.41% area reduction for pipelined datapaths, and 66.92% area reduction for VLIW datapaths. Our solution is simple and effective, and can easily be integrated into an existing silicon compiler.

Categories and Subject Descriptors
C.1.3 [Processor Architectures]: Other Architectural Styles

General Terms
Algorithms, Performance, Design

Keywords
Compiler, Field-Programmable Gate Array (FPGA), Integer Linear Programming (ILP), Resource Sharing

1. INTRODUCTION
Recent advances in sub-micron technology have enabled the widespread use and deployment of reconfigurable embedded systems, namely Field Programmable Gate Arrays (FPGAs). Compilers that target FPGAs are empowered with the ability to customize the FPGA configuration to best accelerate the performance of an application or set of applications. Compilers that transform high-level programs into FPGA configuration files must generate custom hardware that can fit into fixed-area device. The problem of generating a set of custom instructions for a given application has been studied extensively in recent years [1] [2] [4] [5] [6] [8] [11] [16] [17]. The instructions are then synthesized and programmed onto an FPGA. Due to area constraints, only a limited subset of instructions can be implemented in hardware.

To pack the greatest number of operations into a fixed-size device, a compiler must aggressively minimize total design area. Many compilers, however, estimate the area associated with individual instructions, but do not consider area savings due to resource sharing. They produce sub-optimal solutions because they fail to incorporate resource sharing into their area estimates, therefore dramatically overestimating the cost of the resulting datapath. This is especially true of compilers that model the problem using integer linear programming (ILP).

This paper presents a compelling argument against ILP-based solutions to the problem of determining which instructions to synthesize. The primary contribution is an efficient and accurate polynomial-time heuristic that uses resource sharing to minimize the area required to synthesize a set of instructions. The heuristic uses a two-phase hierarchical decomposition to select the resources that are shared. This algorithm is applied to the synthesis of both pipelined and VLIW FPGA configurations.

The paper is organized as follows. Section 2 discusses related work; Section 3 introduces background material; Sections 4, 5, and 6 present the contributions of the paper. Experimental results are detailed in Section 7. Section 8 concludes the paper.

2. RELATED WORK
Huang and Malik [9] recently studied resource sharing to reduce reconfiguration overhead using datapath merging. Coarse-grain logic blocks perform computations, and reconfigurable logic is used to provide an interconnection network between blocks and storage. Datapaths are merged one at a time to optimize interconnection sharing. At each step a maximum bipartite matching problem is solved to maximize interconnection sharing between blocks. Moreano et. al. [14] extend Huang and Malik’s work with a technique that relies on solving the Maximum Clique Problem, which is NP-Complete [7]. Consequently, the quality of their results depends on the quality of the clique-finding heuristic.

Our work differentiates itself from [9] and [14] in that we begin with a set of customized instructions, which are modeled as directed acyclic graphs (DAGs), not general graphs. Our goal is to maximize area reduction, not the sharing of interconnections, although we could reformulate the problem to balance both of these goals if we so desired. Finally, the instructions that we merge do not initially contain low-level details such as the location of storage elements (e.g. registers).
A related problem is Regularity Improvement [10]. Algebraic transformations are applied to a flow graph to improve its internal regularity. The techniques in this paper exploit similarities between instructions instead of using rule-based manipulations.

3. PRELIMINARIES

A Dataflow Graph (DFG) $G = (V, E)$ is a directed acyclic graph (DAG), where vertices represent operations and input/output ports, and edges represent data dependencies between operations. Let $G = (V, E)$ be a DFG. $V$ is comprised of three disjoint subsets, $V \text{in}$ (input ports), $V \text{out}$ (output ports), and $V \text{op}$ (operations). Each $v \in V \text{in}$ has in-degree 0; every $v \in V \text{out}$ has out-degree 0. Every $v \in V \text{op}$ has in-degree 1 (unary operations) or in-degree 2 (binary operations), and out-degree > 0. Each vertex $v \in V$ has an integer type, $t(v)$, which represents an operation or I/O port.

Graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are said to be isomorphic if there exist functions $f: V_1 \rightarrow V_2$ and $g: E_1 \rightarrow E_2$ such for every pair of edges $e_1 = (u_1, v_1) \in E_1$ and $e_2 = (u_2, v_2) \in E_2$:

$$g(e_1) = e_2 \Leftrightarrow f(u_1) = u_2 \land f(v_1) = v_2$$  \hspace{1cm} (1)

The problem of determining if two graphs are isomorphic has no known polynomial-time solution, but has been proven NP-Hard. The problem of determining whether one graph is isomorphic to a subgraph of another is NP-Complete [7].

4. PATH-BASED RESOURCE SHARING

A path is a DFG $v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k$, represented by sequence $\langle t(v_1), t(v_2), \ldots, t(v_k) \rangle$. Let $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ be sequences. $Z = \langle z_1, z_2, \ldots, z_k \rangle$ is a subsequence of $X$ if there is a strictly increasing sequence $\langle i_1, i_2, \ldots, i_k \rangle$ of indices of $X$ such that

$$X_{i_j} = Z_j \text{ for } j = 1, 2, \ldots, k$$  \hspace{1cm} (2)

If $Z$ is a subsequence of $X$ and $Y$, then $Z$ is said to be a common subsequence of $X$ and $Y$. The problem of determining the longest common subsequence (LCS) of a set of sequences has an $O(mn \log m)$ solution [13]. For example, if $X = \langle A, B, C, B, D, A \rangle$ and $Y = \langle B, D, C, A, B, A \rangle$, the LCS of $X, Y$ is $\langle B, C, B, A \rangle$.

A substring is defined to be a contiguous subsequence. The problem of determining the longest common substring (LCStr) of a set of strings has an $O(mn + nm)$ solution [18]. The LCStr of $X$ and $Y$ in the preceding example is either $\langle A, B \rangle$ or $\langle B, D \rangle$.

Each path represents a sequence of machine-level operations. Each operation must execute on some functional unit, whose area is assumed known. For each operation type $t(v)$, we associate two quantities, delay($t(v)$) and area($t(v)$), delay and area estimates for all vertices of the same type as $v$. The area and delay of a sequence $X$, denoted $A(X)$ and $D(X)$ respectively, are defined to be the sum of the areas and delays of each operation within $X$.

To maximize area reduction by resource sharing along a set of paths, we desire the Maximum Area Common Subsequence (MACSeq) or Substring (MACStr) of a set of sequences, as opposed to the LCS or LCStr. MACSeq and MACStr favor shorter sequences of high-area components (e.g. multipliers, dividers) rather than longer sequences of low-area components (e.g. logical operators), which could be found by LCSeg or LCStr.

Figure 1 If $z_i$ and $z_{i+1}$ match consecutive characters in $X$ and $Y$, then a mux is necessary on the input to $z_i$ (top). Otherwise, muxes are necessary on the inputs to $z_i$ and $z_{i+1}$ (bottom).

Theorem 1. Let $\Delta A_{\text{Seq}}(X, Y, Z)$ be the area reduction when $X$ and $Y$ share resources on MACSeq $Z$ and $\Delta D_{\text{Seq}}(X, Y, Z)$ be the delay increase (along each path) due to multiplexers. Then:

$$A(Z) \leq \frac{m}{2} \times \text{area(mux)} \leq \Delta A_{\text{Seq}}(X, Y, Z) \leq A(Z)$$  \hspace{1cm} (3)

$$0 \leq \Delta D_{\text{Seq}}(X, Y, Z) \leq \frac{m}{2} \times \text{delay(mux)}$$  \hspace{1cm} (4)

Proof. $\Delta A_{\text{Seq}}$ and $\Delta D_{\text{Seq}}$ must account for the resources shared along $Z$ as well as the multiplexers that are introduced to the datapath. If $X$ and $Y$ are identical, then all resources can be shared without multiplexers, justifying the upper bound in (3) and the lower bound in (4). In the worst case, if two operations $z_i$ and $z_{i+1}$ appear consecutively in $X$ and $Y$, then a multiplexer will not be needed on the input to $z_{i+1}$ (see Figure 1). Since $m \leq n$, $Z$ has at most $m$ characters and at most $m/2$ non-consecutive operations. At most $m/2$ multiplexers will be introduced to the datapath. \hfill \square

Theorem 2. Let $\Delta A_{\text{Str}}(X, Y, Z)$ be the area reduction when $X$ and $Y$ share resources on MACStr $Z$ and $\Delta D_{\text{Str}}(X, Y, Z)$ be the delay increase (along each path) due to multiplexers. Then:

$$A(Z) \leq 2 \times \text{area(mux)} \leq \Delta A_{\text{Str}}(X, Y, Z) \leq A(Z)$$  \hspace{1cm} (5)

$$0 \leq \Delta D_{\text{Str}}(X, Y, Z) \leq 2 \times \text{delay(mux)}$$  \hspace{1cm} (6)

Proof. The upper bound in (5) and the lower bound in (6) are justified by the proof of Theorem 1. Now, observe that the MACStr of paths $X$ and $Y$ adds at most 2 multiplexers to the datapath, one on the first operation that is matched, and another to select between the outputs. (see Figure 2). $\Delta A_{\text{Str}}(X, Y, Z)$ and $\Delta D_{\text{Str}}(X, Z)$ must only account for two multiplexers, which justifies the lower bound in (5) and the upper bound in (6). \hfill \square

5. Resource Sharing for DFGs

Figure 3 shows a polynomial-time heuristic that combines a set of DFGs into a supergraph, called a Consolidation Graph (CG). The ideal CG will minimize total system area when synthesized; unfortunately, the problem of constructing a minimal cost weighted supergraph of a set of graphs is NP-Complete [3].
Algorithm: Construct_Consolidation_Graph( G )
Input:  
G = (G1, G2, …, Gk) : a set of DFGs
Output:  
Gc = (Vc, Ec) : CG
Local Variables:  
P, Pc : Set of sets of paths
G ∈ P, P(i) ∈ P, P : Set of paths
Gc : Set of DFGs
1. P ← Φ
2. For i ← 1 to k
   a. Decompose Gi into set of paths Pi.
   b. P ← P ∪ Pi.
3. (Gp, Gp*) ← Construct_GP*( P )
//***** Global Phase ******
4. While |G| > 1 and Gp ≠ Φ
   a. Merge the DFGs in Gp into a CG Gc along the paths in Gp*.
   b. For every DFG Gi ∈ Gp*
      i. G ← G − Gi;
      j. P ← P − {Pj}.
//***** Local Phase ******
5. While there are at least two disjoint paths in Gc that have a non-empty MACSeq/MACStr
   a. Decompose Gc into set of sets of paths Pc, excluding shared vertices.
   b. If area(S) > Amax, Smax = S; Amax = area(S).
   c. Discard Gi; Not needed ******
   d. Merge all of the paths in Gp*, and update Gc accordingly.
6. Decompose Gi into a set of paths Pi.
   a. Merge Gi into set of paths Pj.
   b. P ← P ∪ Pi.
6. Return Gc

Subroutine: Construct_GP*( P )
Input:  
P : Set of sets of paths
Pc, Pj : Set of paths
Output:  
(P*, Gp*) : (Set of paths, Set of DFGs)
Local Variables:  
Δmax : Integer
S, Smax : Subsequence/Substring
//***** Determine which paths to merge ******
1. Δmax ← 0; Smax ← Φ; P* ← Φ; Gp* ← Φ
2. For every pair of paths paths pi ∈ Pc, pj ∈ Pj, i ≠ j
   a. s = MACSeq/MACStr of pi and pj
   b. If area(S) > Δmax
      i. Smax = S; Δmax = area(S)
3. For each set of paths Pj ∈ P
   a. If Smax is is subsequence/substring of some path pj ∈ Pj
      i. P* ← P* ∪ {pj}
   b. Gp* ← Gp* ∪ {Gi}
4. Return (P*, Gp*)

5.1 Complexity Analysis
Several assumptions are made to simplify notation. The first assumption is that all k input DFGs have exactly the same number of vertices, |V|; the second is that all DFGs are composed of the same number of paths; and finally, that all paths have equal length, denoted L. Let n be the total number of paths in all DFGs.

**Theorem 3.** The time complexities of the CG Construction Algorithm implemented with MACSeq and MACStr are:

\[
O\left(\frac{k|V|n^2L^2}{\log L}\right) \quad \text{and} \quad O\left(k|V|n^2L^2\right)
\]

**Proof.** There can be at most O(k) global iterations of the algorithm since at least one DFG must be added to Gp* per iteration. There can be no more than O(\sqrt{n}) local iterations, since at least one vertex of the selected DFG must be merged with a vertex in Gp* each iteration. The complexity of Compute_Gp*(i) is dominated by the for-loop in line 2, which performs an O(n^2) pair-wise comparison between paths from different DFGs. The respective complexities of computing the MACSeq and MACStr are O(L^2/\log L) [13] and O(L) [18].

5.2 Example
As an example of CG construction, consider the two DFGs G1 and G2 shown in Figure 4 (a). They are decomposed into sets of paths shown in Figure 4 (b). Assuming that area(+) < area(x) < area(%), Smax is determined to be <%, +>; the corresponding vertices are shaded in all of the paths in which they occur. The CG is shown in Figure 4 (c). Each remaining un-shared vertex is marked with a 1 or 2 as to whether it originates from G1 or G2. These vertices must be considered for further resource sharing.

The Local Phase is shown in Figure 4 (d). Smax is <x, +>; the corresponding vertices are shaded as in Figure 4 (b). The resulting CG is shown in Figure 4 (e) after one iteration of the Local Phase; it is shown again after a second iteration in Figure 4 (f). The area cost is the sum of the costs of 3 adders, 2 multipliers, and 1 divider; without resource sharing, the costs would be 5 adders, 4 multipliers, and 2 dividers.
6. DATAPATH GENERATION
The CG is an imprecise model of a datapath containing functional units and interconnections between units. The CG is imprecise because it lacks a formal description of many low-level details, such as bitwidth information, multiplexers, and registers. The CG is a macro-computation that contains within it the functionality to implement each of the instructions described by the original set of DFGs. This section describes how to generate both pipelined and VLIW datapaths from a CG.

Generating a pipelined datapath is simple. Each vertex in the CG is bound to a separate functional unit without any further resource sharing. Pipeline registers are placed between functional units. Multiplexers may need to be placed on the inputs of certain functional units to route data appropriately between units to implement each instruction.

The most area-efficient approach to generating VLIW hardware would be to use one instance of each functional unit. To improve performance, latency-constrained scheduling can be used to estimate the number of functional units required for the design. Latency-constrained scheduling is an NP-Complete Problem [7], but many polynomial-time heuristics have been proposed over the past twenty years.

6.1 Multiplexer Insertion
For a pipelined datapath, some given functional unit F may have a large number of predecessors. Any machine-level operation (e.g. addition) will require at most two predecessors and one successor. Consequently, multiplexers may be required on both inputs of the FU.

We divide functional units into three classes: unary operators (e.g. negation), binary, non-commutative operators (e.g. subtraction, division), and binary commutative operators (e.g. addition, multiplication).

Unary operators are trivial. All inputs must be multiplexed.

Let \( \bullet \) be a binary non-commutative operation. In other words, one cannot infer that \( a \bullet b \neq b \bullet a \). Each input to the operator must be appropriately labeled as to whether it is the left or right operand of \( \bullet \). Exactly two multiplexers are needed for the left and right input ports of the functional unit in the final design. Of course, if there is no more than one input to either the left or right port of the FU, the multiplexer can be omitted.

Figure 6 demonstrates the process of inserting multiplexers for unary and binary non-commutative operations.
A binary commutative operator \( \circ \) exhibits the property that for all inputs \( a \) and \( b \), \( a \circ b = b \circ a \). Let \( G = \{G_1, \ldots, G_k\} \) be a set of DFGs, and let \( G_c = (V_c, E_c) \) be their CG. For each DFG \( G_i = (V_i, E_i) \), let \( f_i: V_i \rightarrow V_c \) and \( g_i: E_i \rightarrow E_c \) define a mapping from \( G_i \) onto a subgraph \( S_i \subseteq G_c \). Let \( v_i \in V_c \) be a binary commutative operator. Define set \( U_c = \{u_{in} \in V_i \mid (u_{in}, v_i) \in E_i\} \) to be the set of input vertices to \( v_i \), and \( G(v_i) = \{G_i \mid \exists v \in V_i \forall v_i = f_i(v), 1 \leq i \leq k\} \) to be the subset of DFGs that have a vertex that maps to \( v_i \).

Finally, define a conflict graph, \( G_{conflict} = (U_c, E_{conflict}) \), where:

\[
E_{conflict} = \bigcup_{G_i \in G_c} \{(u_1, u_2) \mid \exists v_1, v_2 \in V_c \forall f_i(v_1) = u_1, f_i(v_2) = u_2\}
\]  

\( u_1 \circ u_2 \) enforces the constraint that \( u_1 \) must be connected to the left multiplexer and \( u_2 \) to the right, or vice versa. \( u_1 \) or \( u_2 \) could be connected to both multiplexers, if necessary. The occurrence of \( u_1 \circ u_2 \) in a DFG implies that there is an edge \((u_1, u_2)\) in \( E_{conflict} \).

Let \( L \) and \( R \) be the two input multiplexers to \( v_i \) in the final datapath \( D \). All inputs are either connected to \( L \), \( R \), or both. The vertices of \( U_i \) are partitioned into three disjoint sets:

\[
U_1 = \{u \in U_i \mid (u, L) \in D, (u, R) \notin D\}
\]  

\( U_2 = \{u \in U_i \mid (u, L) \notin D, (u, R) \in D\}
\]  

\( U_{12} = \{u \in U_i \mid (u, L) \notin D, (u, R) \notin D\}
\]

Define \( B_{conflict} = (U_1 \cup U_2, \{(u_1, u_2) \mid u_1 \in U_1 \land u_2 \in U_2\}) = G_{conflict} \).

**Theorem 4.** \( B_{conflict} \) is bipartite.

**Proof.** Assume to the contrary that \( B_{conflict} \) is not bipartite. Then \( \exists \) vertex \( u \) with adjacent edges \( e_1 = (u_1, u) \) and \( e_2 = (u_2, u) \), where \( u_1 \in U_1 \) and \( u_2 \in U_2 \), \( u_1 \) is thus connected to \( L \) but not to \( R \), and \( u_2 \) is connected to \( R \) but not to \( L \). To satisfy conflict edges \( e_1 \) and \( e_2 \), \( u \) must be connected to both \( L \) and \( R \), therefore \( u \in U_{12} \).

Given a binary commutative operator \( v_i \in V_c \), its input vertices \( U_i \), and a conflict graph \( G_{conflict} \), we wish to minimize the area of the resulting multiplexers that must be inserted into the datapath. This requires us to effectively balance the number of connections to the left and right multiplexers.

The number of selection bits, \( B_{left} \) and \( B_{right} \), required for the left multiplexers are given by:

\[
B_{left} = \left\lceil \log_2 \left( \left| U_1 \right| + \left| U_{12} \right| \right) \right\rceil
\]  

\[
B_{right} = \left\lceil \log_2 \left( \left| U_2 \right| + \left| U_{12} \right| \right) \right\rceil
\]

To minimize the total area due to multiplexers, we must minimize \( B_{left} + B_{right} \). This is analogous to finding the Maximum Induced Bipartite Subgraph of \( G_{conflict} \), which is NP-complete [7]. Since there will be many binary commutative operators, we employ a simple linear-time breadth-first search heuristic [15].

Figure 7 (a) and (b) show a conflict graph, \( G_{conflict} \), with optimal and sub-optimal solutions. The induced bipartite subgraphs contain the edges shown in bold. For both solutions, \( B_{left} = 2 \); for the sub-optimal solution, \( B_{right} = 3 \), and for the optimal solution \( B_{right} = 2 \). In many cases, it may not be possible to achieve a perfect balance between the left and right multiplexers.

---

**7. EXPERIMENTAL RESULTS**

To realize a set of customized instructions in hardware, we generated a VHDL component library for a Xilinx VirtexE-1000 series FPGA using Xilinx Coregen [19]. Next, we converted the VHDL files to edif netlists using Synplicity Synplify Pro 7.0 [20]. We then placed and routed each netlist using Xilinx Design Manager [19], which provided area estimates.

Next, we selected 11 source code files from the MediaBench [12] application suite. To generate a set of custom instructions, we integrated an algorithm based on the work of Kastner et. al. [11] into the Machine SUIF compiler framework [21]. We used this software to generate a library of custom instructions for each application. These libraries are summarized in Table 1. We also developed a lightweight tool that allowed us to estimate the costs of synthesizing pipelined and VLIW instructions, as described in Section 6. Finally, we implemented the CG Construction Algorithm using MACSeq, as described in Section 5.

We compare the area estimates resulting from our resource sharing technique to the additive estimates utilized by ILP-based selection algorithms. Instead of trying to select an optimal subset of instructions, we simply synthesized all custom instructions for each application. Table 2 presents results for synthesizing both pipelined and VLIW datapaths.
The columns labeled ILP in Table 2 list the total area (in slices) that would be estimated by ILP-based approaches to selection that assume additive area costs. The columns labeled CG + Synthesis show the estimates achieved by our resource sharing technique.

The ILP invariantly overestimates the cost of synthesizing the instructions. The overestimates ranged from 19.53% (Exp. 6) to 78.90% (Exp. 4) for pipelined datapaths, and from 51.96% (Exp. 6) to 85.33% (Exp. 4) for VLIW datapaths. On average, the ILP over-estimated area costs by 55.41% for pipelined datapaths and 66.92% for VLIW datapaths.

Experiment 6 yielded the smallest area reductions in both experiments. Of the five instructions generated, only two contained multiplication operations (1 and 3 operations respectively). After resource sharing, 3 multipliers were required for the pipelined datapath and 2 were required for the VLIW datapath; the area of the remaining multipliers dominated the overall cost of the instructions.

The Xilinx VirtexE-1000 FPGA has 12,288 SLICES. Experiment 11, with all instructions synthesized independently, had pipelined and VLIW areas of 21,794 and 21,122 slices respectively, far in excess of the capacity of the VirtexE-1000. Resource sharing reduced the area estimates for this benchmark to 9,781 and 7,673 slices respectively, both well within the capacity of the target.

### 8. CONCLUSION

A silicon compiler must rely on high-level area estimates of custom instructions in order to determine how many can be synthesized on the device. In this work, we have demonstrated that the assumption of additive instruction area (inherent in ILP formulations that do not allow resource sharing) leads to hardware which is much larger than necessary. This paper has contributed an efficient and accurate polynomial-time heuristic that aggressively shares resources while synthesizing a set of given instructions. Experiments with eleven MediaBench [20] applications indicate that standard ILP formulations (without resource sharing) overestimate area costs by as much as 78.90% and 85.33% for pipelined and VLIW datapaths respectively, and 55.41% and 66.92% on average.

### 9. REFERENCES


---

**Table 2. Area estimation results.**

<table>
<thead>
<tr>
<th>Exp.</th>
<th>Pipelined Datapath</th>
<th>VLIW Datapath</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ILP (Slices)</td>
<td>CG + Synthesis (Slices)</td>
</tr>
<tr>
<td>1</td>
<td>10238</td>
<td>5930</td>
</tr>
<tr>
<td>2</td>
<td>3060</td>
<td>1652</td>
</tr>
<tr>
<td>3</td>
<td>5087</td>
<td>1946</td>
</tr>
<tr>
<td>4</td>
<td>6967</td>
<td>1470</td>
</tr>
<tr>
<td>5</td>
<td>8117</td>
<td>2924</td>
</tr>
<tr>
<td>6</td>
<td>2990</td>
<td>2406</td>
</tr>
<tr>
<td>7</td>
<td>9719</td>
<td>2810</td>
</tr>
<tr>
<td>8</td>
<td>5740</td>
<td>2630</td>
</tr>
<tr>
<td>9</td>
<td>10449</td>
<td>4881</td>
</tr>
<tr>
<td>10</td>
<td>10304</td>
<td>3768</td>
</tr>
<tr>
<td>11</td>
<td>21794</td>
<td>9781</td>
</tr>
</tbody>
</table>