Abstract

In this paper we present a novel compiler-directed approach to
system-level partitioning for a given description of system func-
tionality in a hardware description language (HDL). The algorithm
is based on a definition-use analysis of the storage in the system
model to ensure that the resulting portions can be implemented in a
loosely-coupled multi-rate execution model with minimal synchro-
nization between the portions. The cost of the hardware-software
interface, in terms of amount of buffering required, is computed ac-
curately as a part of the partitioning cost function using a data-flow
reaching analysis. The proposed algorithm has been implemented
and experimental results show up to 65% improvement in buffer
sizes over the min-cut partitioning algorithm.

1 Introduction

One of the foremost problems in hardware/software co-design is
Hardware/Software Partitioning. The goal of hardware/software
partitioning is to define the boundary of the hardware and soft-
ware implementation of the system functionality. One of the major
differences among partitioning approaches is in the way communi-
cation between hardware and software is taken into account during
partitioning. In fact, if not considered a priori, this communication
could easily turn out to be a bottleneck in the final implementa-
tion, resulting in violation of the imposed performance constraints.
The research presented in this paper addresses the problem of ac-
curately estimating the communication cost for hardware/software
partitioning by using data flow analysis techniques.

The commonly used min-cut cost function measures the cost of
a given partition as the size of the edge cut-set for that partition [1].
However, in the case of a hardware-software partition, the commu-
nication between the two components is normally implemented
over buffered channels [2], and the interface cost is given by the
size of buffers required to implement data transfers in both direc-
tions without any loss of data. As not all data values need to be
sent at the same time from one portion to another, the actual size of
the buffer is, in general, much less than the total number of values
to be transferred. In this study, we show that a partitioning algo-
rithm based on an accurate interface cost function (computed using
data flow analysis over the input design) produces a better partition
than the one generated by the direct application of a min-cut graph
partitioning algorithm.

In our approach, the behavioral-level HDL description is parsed
and represented as a set of Control/Data Flow Graphs (CDFGs). A
definition-use analysis of operations in the CDFG yields a Def-
Use Graph that captures the flow of data values among operations.
Our buffer estimation algorithm is based on an analysis of the
communication pattern between the two components of a given
partition of the def-use graph. This estimation algorithm is used
within the KL iterative partitioning framework [3] as cost function
to provide a measure of the quality of the generated partition. The
cost function gives tight upper bounds on the size of interface buffers
required in order to fully support two-way communication between
the two components of a partition. The bounds are tight when all
data and control dependencies due to operations and also due to
resource limitations can be derived from the input specification of
the design. Even if some of the dependencies are left unspecified
or cannot be computed at the current level of abstraction of the
specification, the algorithm produces a looser estimate that still
guarantees correct execution of the two components.

The implementation of the partitioned system allows for multi-
rate execution of the components, which means that one component
can complete multiple iterations during a single iteration of the
other component. In addition, the components execute in a loosely-
coupled fashion since their relative rates of execution can vary over
a period of time.

1.1 Previous Related Work

The problem of buffer size estimation has earlier been studied in the
context of high-level synthesis. In [4] the authors present an algo-
rithm for buffer estimation under execution rate constraints. In [5]
the authors propose use of implicit state enumeration techniques
to determine the size of interface buffer between communicating
finite state machines. Description of HDL inputs with word-level
data operations, however, leads to substantial increase in the num-
ber of states in a finite state machine model to make this technique
impractical for embedded applications. In [6] the authors present
an integer programming formulation for buffer estimation on data-
flow graphs. The algorithm is based on balancing the flow of data
streams through data nodes and no control operations are modeled.

This paper is organized as follows. In Section 2, we define
the system model used in this work. In sections 3 and 4, we
describe our partitioning cost function and the algorithms for buffer
size estimation. Section 5 shows the results of using our cost
function within Kernighan-Lin’s partitioning framework in place of

---

1 This research was supported in part by NSF CAREER Award MIP95-01615.
the edge cut-set cost function. Finally, in Section 6 we summarize the contributions of this work.

2 System Model: The Def-Use Graph

Consider the HDL description of a design \( D = (O, S) \). The description consists of a set of operations \( O \) over a set of variables \( S \). The value of each variable in \( S \) is defined by some operation in \( O \) on the basis of the values of some other variables in \( S \). This relationship among operations forms the basis of our data-flow assisted communication cost estimation routines and is formalized in terms of the following sets:

- \( \text{usede} f(O) \) = set of operations that use at least one value defined by operations in \( O \).
- \( \text{defuse}(O) \) = set of operations that define the values of variables used by the operations in \( O \).

The sets, \( \text{usede} f() \) and \( \text{defuse}() \), are used to define a directed graph \( G \) with the set of operations, \( O \), as its nodes. A directed edge \((o_1, o_2)\) is placed between \( o_1 \) and \( o_2 \) if \( o_1 \in \text{defuse}(o_2) \), or equivalently, \( o_2 \in \text{usede} f(o_1) \). These sets are determined for each operation by doing a reaching definition analysis over CDFG representation of the given HDL specification [7]. We term the graph \( G \) a def-use graph.

The problem of partitioning \( D \) into two interacting components with the aim of minimizing the interface cost can be formulated as:

**Problem** Given a design \( D = (O, S) \), find a partition \( D = \{D_1, D_2\} \), where \( D_1 = (O_1, S_1) \) and \( D_2 = (O_2, S_2) \) such that

1. Sizes of \( D_1 \) and \( D_2 \) are constrained.
2. The interface cost, \( C = C_{12} + C_{21} = |b_{12}| + |b_{21}| \), is minimized.

The two components, \( D_1 \) and \( D_2 \), are implemented in two separate communicating processes. The interface cost \( C \) is defined to be the cost of communicating the values of variables between the two processes, and is the sum of the interface buffer sizes, \( b_{12} \) and \( b_{21} \), required to support this two-way communication. Each process gets the values of those variables that are used by it, but not defined by it, from the other process through a buffered channel and stores it in its local memory for subsequent "use". This means that the communication cost is exactly one for each element consumed, and does not depend on how many times that element will be used by the process.

3 Accurate Estimation of Interface Buffer Size

Consider the partition \( \{D_1, D_2\} \) of the design \( D \). Let the def-use graph for \( D \) be \( G \). We define a computation path in \( G \) from a node \( o_1 \) to another node \( o_2 \) as a directed path starting at \( o_1 \) and ending at \( o_2 \) with zero or more nodes in between. In general, there will be multiple computation paths between any two nodes in \( G \). If all the nodes on a computation path belong to the same component of the given partition, the path is called an intra-partition path, otherwise it is termed an inter-partition path. An inter-partition path requires the intermediate values to be communicated across the partition through interface buffers.

Consider an inter-partition computation path \( o_1 \rightarrow o_2 \rightarrow o'_2 \) in \( G \), with \( o_1 \) and \( o'_2 \) belonging to process \( D_1 \) and \( o_2 \) belonging to \( D_2 \). Let \( v'_2 \) be the value computed by \( o'_2 \) using the value, \( v_2 \), supplied by \( o_2 \). Assume that \( v'_2 \) is used in some other operation \( o_2' \) in \( D_2 \). Now, the computation of \( v_2 \) itself requires the value, \( v_1 \) from \( o_1 \) in \( D_1 \). This implies that \( v_1 \) and \( v'_2 \) cannot be in the buffer, \( b_{12} \), from \( D_1 \) to \( D_2 \) at the same time. The reason is that \( v_1 \) must be consumed (i.e., removed from \( b_{12} \)) in order to compute \( v_2 \) before \( v'_1 \) can even be computed. Therefore,

**Observation 1** If two values \( v \) and \( v' \) are in the buffer \( b_{12} \) at the same time, then there is no computational dependency between \( v \) and \( v' \) through any operation in \( D_2 \).

Note that, there could still be another data dependency between \( v \) and \( v' \) through some intra-partition computation path entirely within process \( D_1 \). Such intra-process data dependencies are not considered by our interface size estimation routine. (This makes our estimates independent of the specific operation scheduling policies adopted in the final implementations of the two processes.)

Therefore, the buffer size \( |b_{12}| \) can be determined as follows. Let \( O \) be the set of operations in process \( D_1 \) whose computed values need to be communicated across to process \( D_2 \). Consider a directed graph \( T_{12} = (V_{12}, E_{12}) \) with \( V_{12} = O \) and \( E_{12} = \{e | e = (o, o') \} \) if the value of \( o' \) depends on \( o \) through an inter-partition computation path that crosses, may be multiple times, over to \( D_2 \). We call this the transmission graph, \( T_{12} \), from \( D_1 \) to \( D_2 \). (A similar graph, \( T_{21} \) is defined in the reverse direction for communicating data from the process \( D_2 \) to \( D_1 \).)

Each edge in the graph \( T_{12} \) denotes the fact that the endpoints of that edge cannot be in the buffer \( b_{12} \) at the same time. Conversely, if two nodes, \( v_1 \) and \( v_2 \), in the transmission graph are not connected by an edge, then they could "potentially" end up in the interface buffer at the same time. Extending this argument further, we see that the interface buffer should have enough space to hold any independent set of nodes in the transmission graph, since all of them could potentially co-exist in the buffer. Therefore,

**Observation 2** The minimum buffer size, \( C_{12} \), of the communication channel from process \( D_1 \) to process \( D_2 \) is equal to cardinality of the largest set of independent nodes (i.e., the nodes that do not have any edge between them) in the transmission graph, \( T_{12} \).

For a general transmission graph, finding the maximum independent set of nodes is an NP-complete problem [8]. But, as we show in this work, in a number of cases, the undirected version of the transmission graph can be shown to belong to a specific class of perfect graphs, namely, comparability graphs [9].

**Definition 3.1** A simple, undirected graph \( G = (V, E) \) is a comparability graph if there is a partial order \( \leq \) on \( V \) such that vertices \( x \) and \( y \) are adjacent if and only if \( x \leq y \) or \( y \leq x \).

In Section 4.4, we present a polynomial time algorithm for computing the size of the largest independent set of nodes in a comparability graph. If the transmission graph does not turn out to be a comparability graph, we can compute approximate upper bounds to the cardinality of the largest independent set as discussed later in Section 4.4.

Based on the above discussion, the general framework for computing the communication cost can be stated as follows: First, compute the transmission graphs \( T_{12} \) and \( T_{21} \). Then, find either the exact or approximate cardinalities of the largest independent sets of nodes in both \( T_{12} \) and \( T_{21} \). Add the two numbers to get the total communication cost, \( C \).

4 Generating the Transmission Graph

In this section, we show how to construct the transmission graph from a given def-use graph. Sections 4.1, 4.2 and 4.3 deal with the three cases of constructing transmission graphs for simple, conditional and cyclic def-use graphs.
4.1 Simple Def-Use Graph
The def-use graph $G$ for an HDL description with no conditionals or loops is a simple, directed acyclic graph. So a computation path from $o$ to $o'$ followed by a computation path from $o'$ to $o''$ is also a computation path from $o$ to $o''$. We construct $T_{12}$ as follows: First, form the vertex-set $V_{12}$ of all those operations $o_1 \in O_1$ whose value is used on the other side of the partition. Now, compute the transitive closure, $G^*$, of the def-use graph $G$ of the design. If $G$ is represented in the adjacency matrix format as an $n$ by $n$ 0-1 matrix where $n$ is the number of operations in $G$, then $G^* = G + G^2 + G^3 + \cdots + G^n$, where the usual multiplication and addition operations have been replaced by Boolean-And and Boolean-Or operations respectively. Each computation path in $G$ becomes an edge in $G^*$. The edge-set $E_{12}$ is computed by adding an edge $(o_1, o'_1)$ to $E_{12}$ if $G^*$ has edges $(o_1, o_2)$ and $(o_2, o'_1)$, where $o_2$ belongs to $O_2$.

The graph $T_{12}$ satisfies the following property: if $o, o', o''$ are nodes in $V_{12}$, and $(o, o')$ and $(o', o'')$ are edges in $E_{12}$, then $(o, o'')$ is also an edge in $E_{12}$. Hence, $T_{12}$ represents a partial order on its set of vertices $V_{12}$, which implies that the undirected version of $T_{12}$ is a comparability graph.

4.2 Conditional Def-Use Graph
In the case of more complex control/data flow graphs incorporating if-then-else statements (but without any loops), the run-time execution of the process body could take any one of the many possible execution paths depending on the actual input set. So in order to consider the effect of all possible computation paths, each def-use edge is labeled with a Boolean expression denoting the conditions that must hold true for that def-use edge to be active in a particular execution.

The transmission graph $T_{12} = (V_{12}, E_{12})$ is constructed as follows: Determine the set of interface nodes $V_{12}$ in process $D_1$ whose values could potentially be communicated across to process $D_2$. For each pair of interface nodes, $o$ and $o'$, if there is an unconditional inter-partition dependence from $o$ to $o'$, then add a directed edge $(o, o')$ to the set of edges $E_{12}$. This implies that in all those execution runs in which the computation of the value of $o'$ is influenced by the value of $o$, the computation of $o'$ will have to wait for the completion of at least one inter-partition communication path between $o$ and $o'$. This ensures that the value of $o$ in the interface buffer will be consumed before the value of $o'$ can be computed, implying that values of $o$ and $o'$ cannot co-exist in the interface buffer. Unlike in the case of simple def-use graphs, $T_{12}$ may not define a partial order on the set of nodes $V_{12}$. Hence, it is not guaranteed that the transmission graph so obtained is a comparability graph.

4.3 Def-Use Graph with Cycles
Here we assume that the process repeats itself after completing one execution. The transmission graphs in this case are constructed as follows: We first compute a pre-transmission graph, $T'$ for each buffer from $G$ using the algorithm for constructing transmission graphs given in previous Subsection 4.1. (In this case, the def-use graph is cyclic due to the dependencies across iterations. So the pre-transmission graphs too will have cycles and self-loops.)

If a node in a pre-transmission graph does not have a self-loop, then this node could potentially be scheduled across iterations in such a way that all its different values end up in the buffer at the same time. The estimated buffer size will be infinite in such a case. Now, assuming that all the nodes in the pre-transmission graph, $T'$, do have self-loops, we construct the transmission graph $T$ from $T'$ as follows: If there is a directed edge from node $o$ to node $o'$ in $T'$ but no edge from $o'$ to $o$, then value of $o$ from one iteration can co-exist with the value of $o'$ from another iteration. Hence, in $T$ we do not put an edge between $o$ and $o'$. However, if both edges $(o, o')$ and $(o', o)$ exist in $T'$, then this means that $o$ must be consumed before $o'$ in the same iteration can be computed, and $o'$ must be consumed before $o$ for the next iteration can be computed. Thus, $o$ and $o'$ can never coexist in the buffer in such a situation. So we put an undirected edge $(o, o')$ in the refined graph $T$. Note that, unlike the previous cases of simple and conditional def-use graphs, the transmission graph in this case is an undirected graph.

### Algorithm 1: Buffer Size Estimation (BE)

1. Form the transmission graph $T$ for the current segment
2. Generate the flow network $F$ from $T$
3. Find the maximum flow in $T$
4. Compute the size of the maximum independent set of nodes in $T$ using Dilworth’s theorem.
5. Return buffer size as the size of maximum independent set

### 4.4 Algorithms for Buffer Size Estimation
We now present a polynomial time algorithm to compute the tight upper bound on the buffer size for the case when the transmission graph is a comparability graph. For such a case, a transitive orientation, $T$, of the transmission graph edges can be obtained using algorithm given in [9]. (Note that $T$ here could denote either $T_{12}$ or $T_{11}$, depending on the direction of data transfer being considered.) The transitive orientation $T$ is a directed acyclic graph (DAG) representing a partial order over its set of nodes. Hence, if the edges $(a, b)$ and $(b, c)$ are in $T$, then the edge $(a, c)$ is also in $T$.

By Dilworth’s theorem [9], the size of the maximum independent set of nodes in such a graph is equal to the minimum number of vertex-disjoint paths required to cover all nodes in $T$.

The problem of finding the minimum number of vertex-disjoint paths in a DAG can be solved in polynomial time by transforming it into a max flow problem on a flow network $F$ [10]. The flow network $F$ is derived from $T$ as follows: Let $T = (V, E)$, with the set of vertices $V = \{v_1, v_2, ..., v_{|V|}\}$. The vertex set, $V'$, in $F$ consists of a source vertex $s$, a sink vertex $t$, and intermediate vertices $\{x_1, x_2, ..., x_{|V|}\} \cup \{y_1, y_2, ..., y_{|V|}\}$, where $x_i$ and $y_i$ correspond to the vertex $v_i$ in $T$. The edge set, $E'$, of $F$ consists of edges connecting the source vertex to $x_1$’s, edges connecting $y_i$’s to the sink vertex, and an edge from $x_i$ to $y_i$ if there is an edge from $v_i$ to $v_j$ in $T$.

Thus, $E' = \{(s, x_1)|1 \leq i \leq |V|\} \cup \{(y_i, t)|1 \leq i \leq |V|\}$

All edges $e$ in $E'$ have a capacity $c(e) = 1$. If $f$ is the maximum flow in $F$, then the minimum of vertex-disjoint paths which cover $V$ in $G$ is $|V| - f$. Hence by Dilworth’s theorem:

$$\text{buffer size} = \text{max independent set} = \min \text{ number of vertex-disjoint paths } = |V| - f$$

Based on this result, we propose the Algorithm 1 for estimating buffer sizes (BE).

It is possible to speed up the computation of interface cost by using approximate bounds on the cardinality of the largest independent set of nodes in $T$. One such upper bound is the edge cut-set in the def-use graph, which is the same as the number of nodes in the transmission graph itself. Another approximate bound, which is tighter than the edge cut-set, is given below:

$$|b_{12}| \leq \max_{v \in V(T_{12})} \left[|V(T_{12})| - (|\text{pred}(v)| + |\text{succ}(v)|)\right]$$
Algorithm 2: Approximate Buffer Size (ApproxBE)

1. Form the transmission graph $T$ for the current segment
2. Find a vertex $v$ in $T$ with minimum number of neighbors (counting both predecessors and successor of $v$)
3. Return buffer size as (number of vertices in $T$ - number of neighbors of $v$)

The right-hand side in the above inequality refers to finding the most ‘unconnected’ node in the transmission graph, $T_{12}$, since the size of the largest independent set is bounded by this quantity. The approximate bounds can also be used when the transmission graph is a comparability graph and the computation of the exact bound requires more than a reasonable amount of time. Our second algorithm, $\text{ApproxBE}$ (Algorithm 2), is based on the above inequality and is computationally much faster and simpler than algorithm $\text{BE}$. $\text{ApproxBE}$ is applicable even when the transmission graph is not a comparability graph.

5 Experiment and Results

In this section, we show the partitioning results obtained by using our buffer size estimation routines as cost function in the Kernighan-Lin framework and compare them with the traditionally used Min-Cut algorithm. The cost function in the partitioning algorithm used the exact buffer size estimates from procedure $\text{BE}$ in one set of experiments (labeled the Min-Buffer algorithm) and the approximate buffer estimates from procedure $\text{ApproxBE}$ in the other set of experiments (labeled the Approx-Min-Buffer algorithm).

We ran all three routines on a number of randomly generated directed acyclic flow graphs ranging in size from 10 nodes to 70 nodes. The only constraint placed on the partitions was that the two segments should have equal number of nodes in the optimal partition. For each graph, all three programs were given the same initial partition. The results obtained are shown in table 1.

<table>
<thead>
<tr>
<th>Example</th>
<th>Initial Partition</th>
<th>Total Buffer Cost of final partition</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Size</td>
<td>Min-Cut</td>
</tr>
<tr>
<td>g10</td>
<td>10</td>
<td>6</td>
</tr>
<tr>
<td>g10.1</td>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>g20</td>
<td>20</td>
<td>7</td>
</tr>
<tr>
<td>g30</td>
<td>30</td>
<td>12</td>
</tr>
<tr>
<td>g30.1</td>
<td>30</td>
<td>14</td>
</tr>
<tr>
<td>g40</td>
<td>40</td>
<td>11</td>
</tr>
<tr>
<td>g40.1</td>
<td>40</td>
<td>12</td>
</tr>
<tr>
<td>g50</td>
<td>50</td>
<td>12</td>
</tr>
<tr>
<td>g60</td>
<td>60</td>
<td>14</td>
</tr>
<tr>
<td>g70</td>
<td>70</td>
<td>19</td>
</tr>
</tbody>
</table>

Table 1: Partitioning Results on Randomly Generated Graphs

We observe a decrease of 65% in the total buffer cost for algorithm $\text{BE}$ over Min-Cut in case of example g70. An interesting observation is that with an increase in the size of the flow graph, the amount of decrease in the buffer size due to the routines $\text{BE}$ and $\text{ApproxBE}$ also increases.

We also experimented with the $\text{BE}$ and $\text{ApproxBE}$ routines by starting the Min-Buffer and Approx-Min-Buffer algorithms with the “best” partition generated by the Min-Cut algorithm (table 1). We don’t observe a significant change in the final buffer costs due to this change in the initial partition. This leads us to believe that the minimal cost partitions generated by Min-Buffer and Approx-Min-Buffer algorithms are actually near to the minimum attainable interface cost partitions.

6 Conclusions

We have presented a method for the partitioning of behavioral HDL descriptions that uses data flow analysis within a graph partitioning framework. The interface buffer size is shown to be an accurate and realistic measure for estimating the communication cost. The buffer estimation algorithm presented here gives us tight upper bounds on the sizes of interface buffers. Incorporating buffer estimate in the KL framework allows us to directly generate partitions that are better than the ones generated by using the older cost estimate of the size of edge cut-set.

Our partitioning cost model does not make use of any kind of structural knowledge about the given design. The question of specifying the implementation style (hardware or software) for either of the partitioned components is left open for the later structural design phase. Consequently, the results produced by our algorithms are independent of the actual implementation styles chosen for the components, and are equally valid for all three kinds of partitions: hardware/hardware, hardware/software, and software/software.

REFERENCES