Bus Minimization and Scheduling of Multi-Chip Systems

Michael Shelig<br>Edwin Hsing-Mean Sha

Dept. of Computer Science & Engineering<br>University of Notre Dame<br>Notre Dame, IN 46556

Abstract

This paper considers several different algorithms that reduce the required number of buses for multi-chip module design. An efficient polynomial time algorithm that calculates the minimum number of buses needed given a particular schedule is presented. We also present three algorithms that minimize the number of buses during scheduling. Experimental results are shown that illustrate the efficiency of the algorithms.

1 Introduction

The design of computer systems consisting of several chips is referred to as multi-chip module (MCM) design. Such systems are becoming increasingly important for several reasons. First, the complexity and functionality of computer systems being built is increasing at a dramatic rate. This makes it very difficult for many systems to be built in a single chip even with the most advanced computer-aided design tools. Second, multi-chip modules permit designs to be modular. Hence, design time can be dramatically reduced. Finally, multiple chips increase the testability of a system.

Systems built with this design style also require extra buses to connect the different modules. Since such buses are global buses, they contribute significantly to the overall area of the final design. Furthermore, these buses must also be routed, which is also a difficult problem. Hence, it is important to minimize the number of buses during MCM design. This paper presents several scheduling algorithms that minimize the number of buses in a multi-chip module design. Determining the number of buses then places an upper bound on the number of pins. Therefore, in this paper we focus on calculating the number of buses.

While this paper is mainly concerned with bus minimization, there are several other steps in multi-chip module design which may affect the number of buses. Partitioning, scheduling, and pin allocation all contribute to the number of buses that are needed. Partitioning [8, 11, 9, 6] involves minimizing the number of transfers between partitions. Obviously, this will effect the number of buses. This paper assumes that the system has previously been partitioned.

Some research on partitioning considers the quality of the final design. For example, Gupta and De Micheli [7] present an algorithm for partitioning and then use the area and latency of the resulting system as a criterion for evaluating the partition. Similarly [1] presents a partitioning algorithm based on stochastic evolution, which is similar to simulated annealing. It then determines the area and number of pins required to evaluate the partition. Our algorithm can be used as a later stage of theirs.

Most existing scheduling algorithms may be applied directly to the multi-chip module problem. Hence little research on scheduling has been done in conjunction with bus minimization. This may lead to a non ideal number of buses. As an example, let us look at Figure 1. If we presume that data values may be transferred any time between when they are produced and when they are used, the DAG in Figure 1(A) requires two buses while the DAG in Figure 1(B) requires only one bus. This paper introduces several new algorithms that take bus minimization into account during scheduling.

The final step in MCM design involves assigning each data transfer to a particular pin and a particular bus for each timestep. Pin and bus allocation is done in [3] as part of a comprehensive integer programming (IP) model. This model considers partitioning, scheduling, allocation, and pin and bus constraints and uses polyhedral theory to solve the IP problem efficiently. Nevertheless, even efficient IP techniques

Figure 1: A) An Example DAG that Requires Two Buses. B) A Similar DAG that Requires Only One Bus.
are computationally expensive. While our methods are not guaranteed to generate the best solution all the time, they generally yield very good results in most cases.

Haung and Parker have also done two papers which concentrate on pin allocation but also do bus allocation.[5, 4] The first of these assumes a fixed number of buses and pins for a partitioned system. It then attempts to determine if a pin-bus configuration exists that enables all necessary connections to be made. This is done by grouping I/O transfers together with each final group correlating to a single bus. Scheduling is done only after all buses have been assigned. The second paper attempts to minimize the number of pins on each chip, given a partition. Force-directed scheduling that takes into account the number of I/O operations is done first. Next, interchip connections are determined, with the bus allocation being a direct result of the pin allocation. This approach concentrates on limiting the number of pins during pin allocation. Our approach, in contrast to theirs, concentrates on limiting the number of buses during scheduling. Furthermore, it uses a different interconnection model which allows data transfers at more timesteps.

Section 2 introduces the interconnection models used by our algorithms. These models are based on the amount of time that a partition is able store a data value, which depends on the type of circuit being designed. A polynomial time algorithm, the PolyBus algorithm, that minimizes the number of buses for a previously scheduled graph is developed in section 3.1. While this algorithm is computationally efficient it does not address the issue of scheduling.

In section 3.2 three algorithms are presented for scheduling a dependency graph. All three algorithms attempt to maximize the number of timesteps that a transfer may take place in. In particular, transfers with little or no flexibility are avoided. Section 4 demonstrates the effectiveness of the algorithm for several input systems. Finally, section 5 draws conclusions from the results obtained and discusses ideas for future research.

2 Definitions and interconnection models

In this paper we are concerned about directed acyclic graphs (DAGs). A DAG is a directed graph \( \gamma = (OP, E, P, T, ti) \) where \( OP = \{o_i \mid 1 \leq i \leq n\} \) is the set of operations, \( type(o_i) \) is the type of operation \( o_i \) and \( part(o_i) \) is the partition which operation \( o_i \) is in, \( E \) is the set of edges which define the precedence from nodes in \( OP \) to nodes in \( OP \), \( T = \{t_k \mid 1 \leq k \leq m\} \) is the set of operation types, \( P = \{p_j \mid 1 \leq j \leq P\} \) is the set of partitions, and \( ti(t_k) \) is the computation time of a node of type \( k \). We assume \( ti(t_k) = 1 \forall k \) for the remainder of the paper. A scheduled DAG is one in which each \( o_i \) has been assigned to a timestep in which it starts to execute. Finally, the term DAG, dependency graph, and data flow graph will be used interchangeably for the rest of the paper.

Before we are able to reason about multi-chip model interconnections, we must determine what models we will use to simulate chip interconnections. In this section we present the two models used for chip interconnections. These are referred to as immediate transfer and flexible transfer.

In the immediate transfer model all values must be transferred as soon as they are produced. For example, in Figure 2(A) the values produced by node 1 and node 2 must both be transferred from partition 1 to partition 2 between timestep 2 and timestep 3. This will require at least two buses as well as two I/O pins for both partitions. However, if we were to reschedule the graph as in Figure 2(B) the value produced by node 2 would be transferred between timestep 3 and timestep 4. Hence we would need only one bus and one I/O pin for each partition.

In the flexible transfer model a value may be transferred between partitions at any time between when it is produced and when it is used. For example, in Figure 2(A), we may transfer the value produced by node 2 after either timestep two or timestep three. Hence there is no need to reschedule the graph to reduce the number of buses, as there was with the immediate transfer model. For the flexible transfer model the schedule in Figure 2(A) is preferable to the schedule in Figure 2(B) since it allows more flexibility in deciding when values are transferred.

3 Description of algorithm

3.1 Bus minimization of scheduled graphs

One common problem that arises in MCM bus minimization is determining the minimum number of buses needed given a scheduled dependency graph. We refer to this as the SCHEDULED BUS MINIMIZATION PROBLEM. In this section we present a polynomial time algorithm for determining the minimum number of buses needed to implement a scheduled dependency graph. We refer to this algorithm as the PolyBus algorithm.

In the SCHEDULED BUS MINIMIZATION PROBLEM a DAG, \( \gamma \), is given with \( D \) pieces of data that are to be transferred between \( P \) partitions. The \( D \) data to be transferred are denoted by

\[\text{Figure 2: A) The Best DAG for the Flexible Transfer Model. B) The Best DAG for the Immediate Transfer Model.}\]
$d_1, d_2, d_3, \ldots, d_p$, while the $P$ partitions are denoted by $p_1, p_2, p_3, \ldots, p_P$. Each node generates a unique data value, and this data is transferred to at most $(P - 1)$ partitions. Note that the data needs to be transferred only once even if several nodes in a receiving partition require it. Each of the $D$ data to be transferred between partitions may be transferred any time between the timestep it is produced, denoted by $ES_d$, and the first timestep it is used by a particular partition, denoted by $LS_{d,p}$. Hence, each value may be transferred at $LS_{d,p} - ES_d = 1$ timesteps.

We may represent each data transfer as a one-dimensional integer interval. We denote each interval by $I_{d,p}$, where $d_i$ is the data to be transferred and $p_i$ is the partition the data is to be transferred to. Each interval begins at $ES_d$ and ends at $LS_{d,p} - 1$. There will be one interval for each unique value that is to be transferred between any two partitions. Hence, there will be at most $D \times (P - 1)$ intervals. It is also obvious that there are at most $|E|$ intervals. Choosing an integer in the interval corresponds to assigning the corresponding data transfer to a particular timestep. If we have $b$ buses available, we may assign up to $b$ intervals to a particular integer.

Given a value of $b$, we may determine if it is possible to schedule the given system with $b$ buses using the following algorithm. Begin at timestep 1 by assigning the current timestep, $CT$, to 1. Put all intervals which have not yet been assigned and which begin at or before $CT$ in a list, denoted by $SL$. Sort $SL$ by the ending time of each interval. Assign the first $b$ intervals in $SL$ to the current timestep. This will assign the $b$ intervals with the least remaining flexibility to the current timestep. We then increment $CT$ and repeat the above process until all intervals have been assigned to a timestep. If any interval can not be assigned before its ending timestep, we can not implement the original system with $b$ buses. Originally, the PolyBus algorithm begins with $b$ set to one. Each time the DAG can not be scheduled using $b$ buses, $b$ is incremented and the above process is repeated.

We now formally define the above problem and prove that the PolyBus algorithm is correct.

**SCHEDULED BUS MINIMIZATION Problem:**
Consider a scheduled dependency graph, $\gamma$, with $D$ pieces of data that are to be transferred between $P$ partitions.

**Question:** What is $\beta$, the minimum number of required buses?

**Theorem 3.1** The SCHEDULED BUS MINIMIZATION problem is solvable in $O((DP)^2 \log(DP))$ time.

In order to help us prove that Theorem 3.1 is correct we define the following.

$X$ is a set of 0-1 valued variables. Each $X_{d,p}$ in $X$ corresponds to $I_{d,p}$. $X_{d,p}$ is 0 if $I_{d,p}$ has not yet been assigned a bus, and 1 if it has. $U1_{CT}$ is defined as the set of intervals $I_{d,p}$, for which $ES_d \leq CT$, and $X_{d,p} = 0$, before buses have been assigned in timestep $CT$. Similarly, $U2_{CT}$ is the set of intervals $I_{d,p}$, for which $ES_d \leq CT$, and $X_{d,p} = 0$, after buses have been assigned in timestep $CT$. The set $D_{U1_{CT}}$ consists of all $d_i$ associated with $U1_{CT}$.

$S(U1_{CT})$ and $S(U2_{CT})$ are the sizes of $U1_{CT}$ and $U2_{CT}$, respectively. $M_k(U1_{CT})$ is the interval in $U1_{CT}$ which has the kth smallest $LS_{d,p}$. Hence, $M_1(U1_{CT})$ is the interval which must be scheduled the soonest. $M_k(U2_{CT})$ is defined similarly.

Theorem 3.1 is proved by showing that the PolyBus algorithm is correct. In order to do this, it is first shown that given a particular number of buses, $b$, the PolyBus algorithm is able to determine if all data transfers may be accomplished with $b$ buses. This is proved in Lemma 3.4. In order to prove Lemma 3.4, we first prove the following two properties.

**Lemma 3.2** Given a DAG, $\gamma$, scheduled in $r$ timesteps, with $D$ data that are to be transferred between $P$ partitions using $b$ buses, $S(U1_{CT})$ is the smallest value possible $\forall CT, 1 \leq CT \leq r$ if the PolyBus algorithm is used.

**Lemma 3.3** Given a DAG, $\gamma$, scheduled in $r$ timesteps, with $D$ data that are to be transferred between $P$ partitions using $b$ buses, $M_k(U1_{CT})$ is the largest value possible $\forall k, k \leq 1 \leq S(U1_{CT}), \forall CT, 1 \leq CT \leq r$, if the PolyBus algorithm is used.

**Proof:** We will prove Lemmas 3.2 and 3.3 concurrently by induction on the number of timesteps.

**Lemmas 3.2 and 3.3 are true at timestep 1**
At timestep 1, no buses have yet been assigned. Therefore, $U1_{CT}$ must be all intervals for which $ES_d = 1$. Hence, $S(U1_{CT})$ is minimal at timestep one and $M_k(U1_{CT})$ is maximal for all $k$.

**If Lemmas 3.2 and 3.3 are true at timestep $CT$, they are true at timestep $CT + 1$**
First it is noted that $LS_{d,p} > CT, \forall CT, 1 \leq CT \leq r, \forall j, 1 \leq j \leq P, \forall d_i \in D_{U1_{CT}}$. Otherwise the algorithm will have failed to meet the bus requirement at timestep $CT = 1$. The PolyBus algorithm chooses $b$ intervals at each CT, provided $S(U1_{CT}) \geq b$. If it is not, our algorithm simply chooses all $d_i, b$ from $U1_{CT}$.

In either case, $S(U2_{CT})$ must be minimal.

Further each time we choose an $I_{d,p}$ from $U1_{CT}$, $M_1(U1_{CT})$ is always chosen. Therefore $M_k(U2_{CT})$ is maximal $\forall k, k \leq 1 \leq S(U2_{CT})$.

We also know that $U1_{CT+1} - U2_{CT}$ must be the same no matter what $U2_{CT}$ is. Therefore, if $S(U2_{CT})$ is minimal, $S(U1_{CT+1})$ must also be. And if $M_k(U2_{CT})$ is maximal $\forall k, k \leq 1 \leq S(U2_{CT})$, $M_k(U1_{CT+1})$ must also be maximal $\forall k, k \leq 1 \leq S(U1_{CT+1})$. □

Lemmas 3.2 and 3.3 may now be used to prove Lemma 3.4.

**Lemma 3.4** Given a scheduled DAG, $\gamma$, scheduled in $r$ timesteps, with $D$ data that are to be transferred between $P$ partitions using $b$ buses, the PolyBus algorithm succeeds if and only if all data transfers can be satisfied by $b$ buses.
Figure 3: A) An Example DAG for the SCHEDULED BUS MINIMIZATION Problem B) The Intervals for the Above DAG C) A Modified DAG That Requires Only Two Buses D) The Intervals for the Modified DAG

Lemma 3.4 may then be used to prove Theorem 3.1. The complete proofs are not included due to space constraints, however they may be found in [10]. We present an outline of the proofs instead. Lemma 3.2 shows that $S(U_{1 CT})$ is minimal at each timestep. Hence the number of buses assigned at each timestep must be maximal and the Polybus algorithm will succeed if a solution exists. Lemma 3.3 shows that $LSM_i(U_{1 CT})$ is maximal at each timestep. Hence the PolyBus algorithm will not succeed if a solution does not exist. Lemma 3.4 may then be used to prove Theorem 3.1 constructively.

Figure 3 is used to illustrate the PolyBus algorithm. First, note that while there are seven unique values being transferred between nodes only five of these are transferred between different partitions. Further, while only five values need to be transferred between partitions, six intervals are required since $d_2$ is used by both partition 1 and partition 3. Also note that even though $d_3$ is used twice by partition 2, it only requires one interval.

After calculating the appropriate intervals we end up with the six intervals as shown in Figure 3(B). We first consider allocating these intervals using only one bus. Obviously, this is not possible since six transfers need to take place between three timesteps using only one bus. Therefore we attempt to allocate two buses to the given intervals. In this case it would appear to be possible to meet the scheduling requirements since we have two buses, three timesteps and six values to be transferred. In the first timestep, we allocate intervals 1 and 4 to the buses. However, in the second timestep, we cannot allocate intervals 2, 5 and 6. Hence, the given schedule requires three buses. This may be verified by repeating the above process for three buses.

If the initial schedule is changed to correspond with Figure 3(C), Figure 3(D) shows the required intervals. Note that only one node, node 4, has been changed. All other nodes are in the same timestep. Likewise, all intervals in Figure 3(D) are the same as in Figure 3(B) except for interval 1, which now ends at timestep 3. If we attempt to satisfy the given schedule using two buses, intervals 1 and 4 are allocated to the buses in the first timestep. In the second timestep, intervals 2 and 6 are allocated to the buses. Finally, intervals 1 and 3 are allocated in the final timestep. Hence, the DAG in Figure 3(C) requires only two buses. The next section will show how to modify a schedule so that less buses are required.

3.2 Bus minimization and scheduling for unscheduled graphs

Scheduling with resource constraint is known to be NP-complete. In this section we present three heuristics that are used to schedule DAGs using the flexible transfer model of interconnection. Each of these considers resource limitations on the number of buses as well as the number of functional units. Due to space considerations complete algorithmic details are not presented, however they may be found in [10]. The algorithms are based on value scheduling, a variation of list scheduling in which only nodes associated with a positive value are scheduled in the current timestep.

3.2.1 Value Scheduling Evaluation Function

During value scheduling we must decide which nodes to schedule in the current timestep. Our evaluation algorithm calculates the inflexibility of each node that would be scheduled in the current timestep if unlimited resources were available. The inflexibility of node $n$ if it were scheduled in the current timestep, $CT$, is denoted by $\delta I_n$. The evaluation algorithm also calculates the inflexibility that would be associated with each node if it were scheduled in the next timestep, or $\delta I_{n+1}$. We then use the difference between these two quantities, $\delta I_n - \delta I_{n+1}$, to determine the increase in inflexibility, $\delta I_n$, if the node were scheduled in the next timestep instead of this timestep. Nodes with large $\delta I_n$ are scheduled first. All nodes with $\delta I_n$ greater than some small negative value are scheduled as long as enough resources are available. This means that some "ready" nodes may not be scheduled even though additional resources are available. By not scheduling ready nodes we spread out the data transfers which helps to minimize the required number of buses.

The inflexibility of each node is the sum of the inflexibilities of all data transfers that are associated with the node. We denote the inflexibility of data transfer $dt$ if the node under consideration were scheduled in $CT$ by $IT_{d|CT}$. If we denote the set of data transfers associated with node $n$ as $D_{n}$, then $\delta I_n = \sum_{dt \in D_n} IT_{d|CT}$. In order simplify the explanation of how to calculate $IT_{n|CT}$, it is first assumed that
all nodes (including successor nodes) have previously been scheduled except for the ones which are currently being scheduled. In this case the inflexibility of each dt is normally the inverse of the flexibility, \( F_{d,CT} \). The flexibility of a data transfer is the number of timesteps in which the transfer may take place. If an interval associated with a data transfer starts at time \( s_a \) and ends at time \( s_b \), then the flexibility of this transfer is \((s_b - s_a) - 1\). Hence the inflexibility of node \( n \) at time \( CT \) is the sum of the inverses of the flexibilities of the data transfers associated with that node, or \( I_{n,CT} = \sum \delta dF_{s, CT} \). This is similar to calculating the effective conductance of a set of resistors which are in parallel.

The above rule for calculating inflexibility of a data transfer is not used in one important case. If the data transfer is an intrapartition data transfer the inflexibility is set to zero. Intrapartition data transfers where only one of the two nodes associated with the data transfer is unscheduled do not count toward the inflexibility since they do not affect the buses.

As an example, consider the DAG in Figure 4. Let us assume that all nodes except nodes 1 and 2 have been previously scheduled, and we wish to schedule both nodes 1 and 2 in timestep 4. However, due to resource constraints, only one of the two nodes may be scheduled in timestep 4. Therefore we must calculate \( \delta I_{s} \) and \( I_{s} \). First, let us calculate the flexibility of the data transfer between nodes 3 and 1, which we will denote by \( dt_{31} \). If we schedule node 1 in timestep 4, \( dt_{31} = (4 - 2) - 1 = 1 \). Likewise, \( dt_{41} = (4 - 2) - 1 = 1 \) and \( dt_{15} = (8 - 4) - 1 = 3 \). Therefore, \( I_{s} = 1 + \frac{1}{3} = 2.33 \). Similarly, if we schedule node 1 in timestep 5, \( dt_{31} = 2, \ dt_{41} = 1, \ dt_{15} = 2, \) and \( I_{s} = \frac{1}{2} + \frac{1}{3} + \frac{1}{2} = 1.5 \). Therefore, \( \delta I_{s} = -0.83 \). The negative value of \( \delta I_{s} \) indicates that it is preferable to schedule this node in the next timestep. In the same manner we find that \( I_{s} = \frac{1}{2} + \frac{1}{3} + \frac{1}{2} = 1.33 \), and \( I_{s} = \frac{1}{2} + \frac{1}{3} + \frac{1}{2} = 1.83 \). Therefore \( \delta I_{s} = 0.50 \), and it would be preferable to schedule this node in the current timestep if possible. Since \( \delta I_{s} \) is greater than \( \delta I_{s} \), we schedule node 2 in timestep 4 instead of node 1.

For data transfers that are between the node we are scheduling and a scheduled node we may determine the flexibility exactly. However, for data transfers in which a node has not been scheduled the flexibility must be estimated. Intuitively, the estimated flexibility is the average amount of spacing between nodes for the longest path between the unscheduled node and the output nodes. Due to space constraints further details on calculating the estimated flexibility are not presented here but may be found in [10].

### 3.2.2 Bus minimization algorithms

The first algorithm, known as timestep addition attempts to maximize the flexibility of transfers during scheduling. First a lower bound on the number of timesteps is determined by presuming unlimited resources. Next, a schedule is generated using value scheduling with the evaluation function presented in Section 3.2.1 and the number of timesteps calculated above. The number of buses is not taken into account during the value scheduling. Only the number of timesteps and the number of functional units is considered. Once the DAG has been constructed using value scheduling the total number of buses required is calculated using the PolyBus algorithm. If the required number of buses is more than the desired number of buses, the number of timesteps is incremented, and the above process is repeated.

The second algorithm, known as timestep insertion initially calculates the same schedule as timestep addition. However, if after calculating the first schedule, the bus requirement can not be met, additional timesteps are inserted at those points which violate the bus constraint.

The third algorithm, known as immediate allocation scheduling schedules a DAG for a given number of buses. The basic value scheduling algorithm is modified so that at each timestep, the algorithm determines if the unit being scheduled will violate the bus constraints. If so, these units are not scheduled until a later timestep.

### 4 Experimental results

This section presents the experimental results for several linear systems/partitions. After presenting the results we briefly explain what input systems were used to generate them. For further details on the input systems, see [10]. We then analyze the results and compare the algorithms' performances. Our algorithms are coded on a Sparc 10 and run in only seconds for these examples.

Figure 5 shows the results for the fifth order wave elliptical filter, using four different partitions/nodes per partition. A few comments should be made to help explain the table. When PT is 3 in Figure 5, the filter has been partitioned using the groupings found in [11]. In order to have more nodes in each partition, the nodes in similar groups in [11] were combined into one partition. In order to test the algorithm in extreme cases, we also considered the situation where all nodes are put into their own partition. PT is 34
<table>
<thead>
<tr>
<th>Filter</th>
<th>PT</th>
<th>AP</th>
<th>MP</th>
<th>BS</th>
<th>TA</th>
<th>TI</th>
<th>IA</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOWE</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>20</td>
<td>19</td>
<td>19</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>18</td>
<td>16</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>17</td>
<td>17</td>
<td>17</td>
<td>2</td>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>1</td>
<td>21</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>17</td>
<td>17</td>
<td>17</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>34</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>49</td>
<td>49</td>
<td>49</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>28</td>
<td>26</td>
<td>25</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>1</td>
<td>18</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>5</td>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**PT** = Partitions; **BS** = Number of Buses
**AP** = Adders Per Partition
**MP** = Multipliers Per Partition
**TA** = Timestep Addition
**TI** = Timestep Insertion
**IA** = Immediate Allocation

Figure 5: Timesteps needed for several partitions and number of functional units

For this case, when **MP**, the number of multipliers per partition is blank, this indicates that all nodes in the DAG were transformed into addition nodes before scheduling. Finally, the filter referred to as **3FOWE** is three copies of the fifth-order wave elliptical filter duplicated and combined into partitions as explained in [10].

Figure 5 shows that immediate allocation is superior to timestep addition and timestep insertion. This is to be expected since immediate insertion schedules as many nodes as possible in each timestep. Only the ordering of the scheduling must be determined. On the other hand, in timestep addition and timestep insertion, the ideal number of nodes to schedule in a timestep is not known. Figure 5 also shows that timestep addition is the worst of the three algorithms. Theoretically, timestep addition should be better than timestep insertion, since it reschedules the graph when the bus requirements are not met, instead of simply adding timesteps with no nodes. However, due to the non-ideal evaluation function found in the value scheduling, timestep addition is not as efficient as timestep insertion. This could be changed if the value scheduling evaluation function were improved.

### 5 Conclusion

In this paper, we considered three algorithms for decreasing the number of buses required in multi-chip modules. There are many models of inter-chip communication in multi-chip module design. This paper considers two such models. The first, immediate transfer, requires that all values are transferred between partitions as soon as they are available. The second, called flexible transfer, allows values to be transferred any time between when they are produced and when they are used.

Several algorithms are presented to help minimize the number of buses required in a multi-chip system. A polynomial time algorithm is that minimizes the number of buses given a schedule is presented. The third part of the paper presents three algorithms that may be used to schedule a previously partitioned data flow graph if we wish to minimize the number of buses. Results that illustrate the savings which are possible with these algorithms are presented for several DAGs/partitions.

### References


