LOWER BOUND ON LATENCY FOR VLIW ASIP DATAPATHS [*]

Margarida F. Jacome and Gustavo de Veciana
Department of Electrical and Computer Engineering
University of Texas, Austin, TX 78712
Tel: (512) 471-2051 Fax: (512) 471-5532
{jacome,gustavo}@ece.utexas.edu

Abstract

Traditional lower bound estimates on latency for dataflow graphs assume no data transfer delays. While such approaches can generate tight lower bounds for datapaths with a centralized register file, the results may be uninformative for datapaths with distributed register file structures that are characteristic of VLIW ASIPs. In this paper we propose a latency bound that accounts for such data transfer delays. The novelty of our approach lies in constructing the "window dependency graph" and bounds associated with the problem which capture delay penalties due to operation serialization and/or data moves among distributed register files. Through a set of benchmark examples, we show that the bound is competitive with state-of-the-art approaches. Moreover, our experiments show that the approach can aid an iterative improvement algorithm in determining good functional unit assignments – a key step in code generation for VLIW ASIPs.

1 Introduction

Lower bound estimates on latency for Data Flow Graphs (DFGs) executing on datapaths have been extensively investigated, see e.g., [11, 6, 10]. High-level synthesis tools have traditionally used these lower bound estimates to identify and prune inferior designs during design space exploration. While some of the bounding approaches give tight bounds when applied to datapaths with a *centralized register file*, they may be uninformative when applied to datapaths with *distributed* register file structures, see e.g., Fig.1. Since the datapaths of Very Large Instruction Word (VLIW) Application-Specific Instruction-Set Processors (ASIPs) typically exhibit such distributed storage structures [8, 7], there is a need to develop bounds that can be informative in this context. These bounds can in turn provide guidance during code generation for this important class of embedded processors – in particular, as discussed in the sequel, during the functional unit binding (assignment) phase of code generation.

In this paper, we propose an approach to lower bounding the execution latency of a DFG, for a *given* binding of the DFG to a datapath, which considers the impact of distributed register file structures on latency. In particular, we will focus on DFGs corresponding to *single basic blocks* within a loop body, since these are typically the time critical segments for the embedded applications and are likely to benefit the most from using VLIW ASIPs [8, 7].

In our DFG examples, we will use the convention of naming activities that require multiplication operations by m, ALU operations by a and a bus use by b, see e.g., Figs.1 and 2. The key issue underlying our work is as follows: when two activities *share* a data object, as $m1$ and $a1$ share $r1[i]$ in Fig.1, it is of interest to bind them to functional resources that *share* common register files – e.g., multiplier M1 and ALU A1 share register file RF1. By doing so, one can in principle avoid delays incurred in moving the result of $m1$ to a new register file before $a1$ can execute. The primary contribution of this paper is the development of a latency bound which directly accounts for such data transfer delays. Since for datapaths with distributed register files the delays associated with such transfers can be significant, the availability of tight lower bounds is critical in the context of VLIW ASIPs.

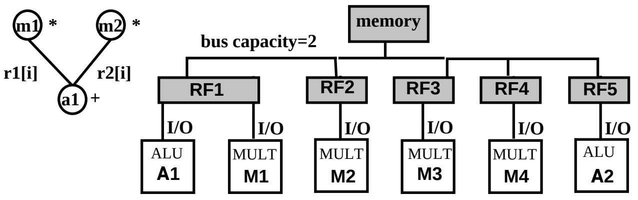

Figure 1: Segment of DFG and VLIW ASIP datapath.

In order to avoid delays due to data transfers, one might seek a binding of DFG activities to datapath functional resources, in which *shared* (result/operand) data objects reside in the same register files. However, in doing so, one may bind two activities, that could have been executed concurrently, to the same resource resulting in a *serialization* of the operations. For example, to avoid data moves between register files, one may bind both $m1$ and $m2$ to M1, so that their results are placed in RF1 from which $a1$ draws its inputs. By doing so, a serialization penalty will be incurred since $m1$ and $m2$ can no longer be executed concurrently. Thus, one can view the binding task as a tradeoff between 1) delays incurred from having to move data objects across distributed register files, and 2) delays incurred from needlessly serializing operations. Fig.2 exhibits two bindings for our example – on the left a binding attempting to avoid moves and, on the right, a binding avoiding serialization. Note that, in this simple example, both bindings lead to the same latency, but in general this will not be the case.

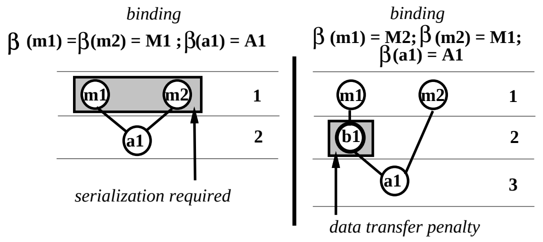
binding
β (m1) = β(m2) = M1 ; β(a1) = A1
m1
m2
a1
1
2
serialization required
binding
β (m1) = M2; β (m2) = M1; β(a1) = A1
b1
3
data transfer penalty

Figure 2: Serialization versus data transfers.

A second contribution of this paper is to develop a model, the *window dependency graph*, capable of capturing chains of increased execution delays caused by such operation serializations. This model proves to be useful in assisting incremental changes to bindings which tradeoff the delays resulting from data moves and opera-

*This work is supported by a National Science Foundation NSF CAREER Award MIP-9624321 and by Grant ATP-003658-088 of the Texas Higher Education Coordinating Board.

tion serialization. We argue that the proposed window dependency grap**h** can be of use during code generation for VLIW ASIPs.

The paper is structured as follows. Section 2 formally defines the problem to be addressed. Section 3 presents the proposed lower-bound on execution latency. Section 4 discusses how the information provided by the proposed lower bounding method may be used in exploring tradeoffs during code generation. Section 5 discusses related work and presents benchmark examples. Conclusions are given in §6.

2 Dataflow graphs, datapaths and bindings

A DFG will be modeled by a DAG, $G(A,E)$, where the nodes A represent *activities*, i.e., operations to be carried out on datapath resources, e.g., adds and moves, and the edges $E \subset A \times A$ represent *data objects* that are "produced" and "consumed" by activities during the flow of execution. Without loss of generality, we assume that an activity can consume at most two data objects, i.e., the in-degree of any node is at most 2. We focus on code segments corresponding to a single basic block within a *loop body*, thus the DFG shown in Fig.1 includes data object labels with iteration indices, e.g., $r1[i], r2[i]$. As discussed below, the DFG model will also include move (i.e., data transfer) activities, required for a given binding of functional activities to datapath resources.

Let R denote the set of datapath resources. These may include ALUs, multipliers and other functional units, as well as buses. For each resource $r \in R$, we let $c(r) \in \mathsf{Z}^+$ denote the capacity of that resource, e.g., an ALU would have a capacity of 1, signifying that it can perform 1 operation per step, whereas a bus resource might have a capacity 2, signifying that it can perform 2 concurrent data transfers.[1] For simplicity we will assume that all activities take a unit step to execute, but the approach can be extended to multicycle and/or pipelined functional units. The datapath is also specified in terms of its (distributed) register files, their connectivity to functional resources and, for simplicity, a shared bus with a given capacity, see e.g., Fig.1.

We assume that functional activities of the DFG have been bound to datapath resources, that is, each activity $a \in A$ is bound to a resource $\beta(a) \in R$ which is capable of carrying out that activity. Given such a binding and the register file connectivity, we identify data object moves that will need to take place between operations, and explicitly include nodes in the DFG corresponding to such moves. Move operations are bound to the datapath's bus. For example, if $\beta(m1) = M1$ and $\beta(m2) = M2$ then an additional node would be inserted between $m2$ and $a2$ to capture the delay to move the result of $m2$ in register file RF2 to register file RF1, see Figs.1 and 2.

3 Lower bound on latency

Recall that our first goal is to determine a *lower bound* on the execution latency for a *given* binding of a DFG to a datapath. The second goal is to generate information that can assist tradeoff exploration during functional unit assignment (binding). We will do this by first determining a *global lower bound*, *L*, on the latency and then, generating a *window dependency graph*, that will permit assessing the additional delays on activities that are incurred due to resource and/or precedence constraints.

[1]In general, one might consider binding activities to *clusters* of functional units sharing a common register file. In this case, one would define the capacity of a cluster to perform a particular type of operation, which would depend on the number of functional units capable of executing the operation in the cluster. This is in fact the manner in which the binding is specified but, to simplify notation, in this paper we will specify bindings directly to resources.

3.1 Global lower bound L

Various methods are available to determine global lower bounds on latency of the schedule, e.g., [11]. For concreteness, we will use the maximum of two simple bounds, however more sophisticated approaches can be used. We first perform an, as soon as possible, ASAP scheduling of the DFG to determine the minimum number of steps that would be required. Next we sum the total number of moves that were explicitly introduced between activities in the DFG with the total number of primary inputs/outputs that are required, and divide by the bus capacity to find the minimum number of steps that would be required to perform the required data transfers. The global lower bound L is given by the maximum of these two numbers.

3.2 Windows

We shall construct three types of *windows* associated with the problem at hand, *individual*, *basic*, and *aggregated* windows. A window, indexed by i, is specified by a four-tuple

$$w(i) = (s(i), f(i), r(i), A_i)$$

where $s(i)$ and $f(i)$ are the start and finish steps for the window, $r(i)$ is a datapath resource associated with the window, and A_i is a set of activities bound to $r(i)$ which ideally would be executed within the scheduling range $[s(i), f(i)]$.

To establish approximate scheduling ranges in which activities might be scheduled we use an ASAP scheduling of the DFG and, given the global lower bound L, perform an as late as possible (ALAP) scheduling of the DFG. Let the activities A be indexed $k = 1,2,\ldots|A|$, where $|A|$ denotes the cardinality of set A. For each activity $a_k \in A$, we define an *individual window* $w^I(k) = (s^I(k), f^I(k), \beta(a_k), \{a_k\})$ where $s^I(k)$, $f^I(k)$ denote the earliest and latest possible steps at which the activity could be executed, based on the ASAP and ALAP schedules, and $\beta(a_k)$ is the resource to which a_k is bound. Note that since the scheduling ranges associated with these windows were derived based on ASAP/ALAP schedules that disregard resource constraints, a schedule in with each activity lies within its individual scheduling range may not be feasible.

Individual windows provide an activity-centric point of view on scheduling constraints. However, there may be multiple activities bound to the *same* resource which share the *same* scheduling range. Given the set of individual windows, we shall construct a reduced set of $j = 1,\ldots n^B$ *basic windows* denoted by $w^B(j) = (s^B(j), f^B(j), r^B(j), A^B_j)$ where A^B_j is the *largest* set of activities bound to $r^B(j)$ with the same individual scheduling range $[s^B(j), f^B(j)]$. A basic window thus groups activities sharing a common resource and the same scheduling range.

Given the collection of basic windows, we then generate a collection of $i = 1,\ldots n^A$ *aggregated windows*, denoted by $w(i) = (s(i), f(i), r(i), A_i)$.[2] The set of aggregated windows includes all the basic windows as well as *mergings* of one or more basic windows, associated with activities bound to the *same* datapath resource. Only windows with scheduling ranges that abut or overlap with each other can be merged and only those with a maximal number of activities for the given scheduling range are kept. Thus each aggregate window corresponds to a maximal number of activities associated with a given scheduling range to be executed on a common resource. Aggregated windows, provide a resource/scheduling range centric view on the problem, by collectively capturing the aggregate resource demands on various ranges of steps.

Fig.3 exhibits a DFG including only additions and multiplications, and the various types of windows that would be generated.

[2]Note that to keep the notation simple we suppress the superscript A that would indicate that these are aggregate windows versus individual I or basic B windows.

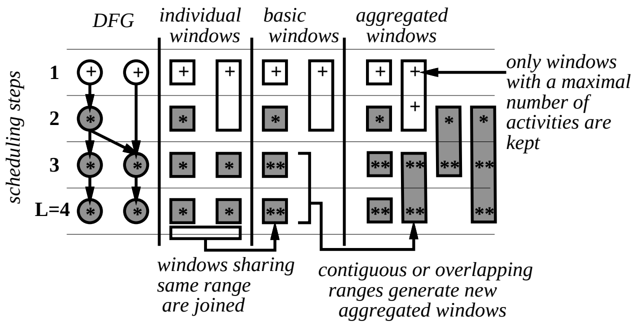
DFG
individual windows
basic windows
aggregated windows
scheduling steps
1
2
3
L=4
only windows with a maximal number of activities are kept
windows sharing same range are joined
contiguous or overlapping ranges generate new aggregated windows

Figure 3: Example of individual, basic and aggregated window construction.

For simplicity we have not labeled windows and activities. Note for example, that one of the addition activities can be scheduled at the earliest on the first step or at the latest on the second step, thus has an individual window with a scheduling range of two steps. Also note that the multiplication activities on the last two steps have the same individual ranges, and hence are collapsed into single basic windows associated with two activities. This better captures the resource demands on these last two steps. Finally, windows that abut or overlap with each other generate new merged aggregate windows. Thus the basic window associated with the activity having a range of two steps is merged with the smaller fully overlapping individual window of the same type. Also various larger windows containing only multiplication activities are generated, capturing the high resource requirements over larger ranges of scheduling steps. A complexity analysis for the window generation process can be found in §3.7.

3.3 Local delays - Resource constrained scheduling

Each aggregated window i corresponds to a set of activities A_i to be executed on resource $r(i)$ within a range of scheduling steps $[s(i), f(i)]$. In the best case, if there are no constraints on the activities in a window, they can be executed in only 1 step, e.g., step $s(i)$. However, usually, due to resource/precedence constraints, the activities associated with the window require several steps to execute, and in some cases might even exceed the upper limit $f(i)$on their scheduling range. To capture this effect we shall compute a lower bound on the *additional* number of steps, i.e., beyond the 1 step case considered above, that any feasible resource constrained schedule will require to execute the activities in A_i. We later define this bound as the *local delay*, $\lambda(i)$, of the window. The bound is obtained by considering the activities A_i in *isolation* i.e., only considering direct precedence constraints among them and the capacity of the resource to which they are bound.

We develop our bound for an arbitrary set of activities, $A' \subset A$ in the graph $G(A,E)$ which are to be executed on the *same* resource r - windows are thus a special case. Let $G(A',E')$ denote the subgraph of $G(A,E)$ which includes the activities A' and all edges $E' \subset E$ between activities in A'. This induced graph captures only direct precedence constraints among activities in A', optimistically dropping all others. Next perform an ASAP scheduling for the activities in the subgraph. Let $l = 1,\dots m$ denote the steps of this schedule, n_l denote the number of activities scheduled on step l, and m be the last non-empty step. Based on the above ASAP schedule, at best, the activities in A' can be completed in m steps. However, since these activities are to be executed on resource r with capacity $c(r)$, no more than $c(r)$ activities may be scheduled per step, i.e., $n_l \le c(r)$. The bound is based on the following observation: a feasible resource constrained schedule may not execute any activity prior to its execution step in the ASAP schedule for the subgraph and may schedule at most $c(r)$ activities per step. Alternatively, we make the *optimistic* assumption that once an activity on step l of the subgraph's ASAP schedule *completes* execution, *any* activity on step $l+1$ can be scheduled for execution. By relaxing constraints among the activities in A' and dropping constraints among A' and the rest of the DAG we can obtain the following local bound on the relative number of steps needed to execute the activities in A'.

Lemma 3.1 *Suppose $A' \subset A$ is a nonempty set of activities bound to a resource r with capacity $c(r)$ and let n_l denote the number of activities in the steps $l = 1,\dots,m$ of the ASAP schedule for the subgraph $G(A',E')$ defined above. Define* bound(A',r) *by*

$$\begin{aligned} x_0 &= 0, \\ x_{l+1} &= \max\{n_l + x_l - c(r), 0\}, \quad l = 1,\dots m, \\ \text{bound}(A',r) &= \lceil \frac{x_{m+1}}{c(r)} \rceil + m - 1. \end{aligned}$$

Then bound(A',r) *is a lower bound on the number of steps, beyond the first one, that any feasible resource constrained schedule would require to complete execution of the activities in A'.*

The proof of this lemma is straightforward and included in the appendix. The iteration which defines the bound corresponds to greedily packing activities, consistent with not beginning execution prior to their associated subgraph ASAP step, and not exceeding the resource's capacity.

With this result in hand we define the *local delay* for window i by $\lambda(i) = \text{bound}(A_i, r(i))$. Thus the last activity in window i must be executed on or after step $s(i) + \lambda(i)$. This must be the case since no activity in A_i can begin execution prior to $s(i)$ and according to Lemma 3.1 at least $\lambda(i)$ additional steps are required. If this exceeds $f(i)$ then the precedence/resource constraints will force activities to be executed outside the window's scheduling range, i.e., incur excess delays, providing valuable localized information on where a particular binding may be leading to scheduling delays.

3.4 Propagated delays - Key Lemma

Local delays capture delays incurred due to precedence/resource constraints within a given window. Due to dependencies among activities in different windows, additional delays may be propagated from one window to another. Without loss of generality consider two aggregate windows, indexed by 1 and 2. We shall define dependencies among windows as follows.

Definition 3.1 *We say that Window 2* **depends** *on Window 1 if among Window 2's activities, A_2, there are activities with direct data dependencies from activities A_1 in Window 1. More specifically let $P_{1,2} \times C_{1,2} := (A_1 \times A_2) \cap E$ be the set of* edges *on the DFG from activities in Window 1 to activities in Window 2, thus Window 2 depends on Window 1 if $P_{1,2} \times C_{1,2} \neq \emptyset$.*

We call $P_{1,2}$ and $C_{1,2}$ the set of *producer* and *consumer* activities associated with this dependency relation. Note that dependency is a directed relationship, i.e., in the above definition, Window 2 depends on Window 1. In the sequel we will use the following notation $P_a := \{b \in A_1 | (b,a) \in E\}$ to denote producers in Window 1 for an activity a and $C_b := \{a \in A_2 | (b,a) \in E\}$ to denote consumers in Window 2 for activity b. Also we define L_2 as the set of activities on first step of ASAP schedule for subgraph $G(A_2,E')$ induced by the activities in Window 2.

We let $\delta(i)$ denote a *lower bound* on the additional delay propagated to an aggregate window $w(i)$ from other windows. Thus, for a given $\delta(i)$, we can guarantee that any feasible schedule for the DFG will have an activity in A_i scheduled on or after step $s(i) + \lambda(i) + \delta(i)$, i.e., after the first scheduling step for the window plus its local and propagated delays. Our goal is to systematically find such incremental bounds, showing where combinations of resource and precedence constraints are likely to lead to propagation of delays across windows, which in turn will increase the latency of the schedule. The algorithm proposed below is based on recognizing two ways in which the activities in Window 1 can further delay the last activity in Window 2. The first is that there is a non-empty set of activities in Window 2 that can only be scheduled after completion of the last activity in $P_{1,2}$. The second is that depending on the minimum number of producers required by the activities in L_2 of Window 2, the start time for execution of the activities A_2 may need to be delayed. For a detailed discussion of the proposed algorithm see the proof of Lemma 3.2 in the appendix. Below we present a concrete example and discussion that should clarify the general idea.

```
**propagated-delay**(1 , 2)
**initialize** $P_{1,2}, P_a, C_b$ and $L_2$
**if** ( $P_{1,2} = A_1$ )              /* compute bound on last producer step */
        last-producer-step $= s(1) + \lambda(1) + \delta(1)$;
**else** start-step $= \min_{a_k}\{s^I(k) | a_k \in P_{1,2}\}$;
        last-producer-step = start-step $+$bound$(P_{1,2}, r(1))$;
                                        /* compute bound on last consumer step */
**if** ($c(r(1)) = 1$ and $\forall a \in L_2, |P_a| = 2$)
        last-consumer-step $= \max\{s(1)+2, s(2)\} + \lambda(2)$;
**else** last-consumer-step $= s(2) + \lambda(2) + \delta(2)$;
                                                        /* take the worst of the two */
num-consumers-for-last-producer $= \min_b\{|C_b| \mid b \in P_{1,2}\}$;
delay $= \lceil$num-consumers-for-last-producer$/c(r(2))\rceil$;
last-consumer-step $=$
        $\max\{$last-producer-step $+$ delay, last-consumer-step$\}$;
    /* compute pairwise propagated delay for Window 2 from 1 */
$\Delta(1,2) =$ last-consumer-step $- [s(2) + \lambda(2)]$;
                /* update worst case propagated delay for Window 2 */
$\delta(2) = \max\{\delta(2), \Delta(1,2)\}$;
```

Lemma 3.2 *Given two aggregate windows, Windows 1 and 2, with associated local and current worst case propagated delays $\lambda(1), \delta(1)$ and $\lambda(2), \delta(2)$ respectively, such that Window 2 depends on Window 1, then the algorithm* **propagated-delay** *above computes a (possibly tighter) updated worst case propagated delay $\delta(2)$ for Window 2, and a pairwise propagated delay $\Delta(1,2)$, i.e., the propagated delay resulting from Window 1.*

Fig.4 shows two windows, 1 and 2, such that Window 2 depends on Window 1. For this example, the dependency between two windows can be shown to further delay the execution of activities in Window 2 and thus increases the lower bound, $\delta(2)$, on the number of additional steps required to execute the activities A_2 in Window 2. Based on their local and current worst case propagated delays, our algorithm computes a new propagated delay $\delta(2)$ for Window 2.[3] The example in the Fig.4 captures one of the cases considered in our algorithm. In particular, that in which all of the activities in A_2 that could have been scheduled on step $s(2)$ (i.e., activity a_4), according to the ASAP schedule, depend on two producers in Window 1. Since the capacity $c(1)$ of the resource associated with Window 1 is only 1, this delays the beginning of execution for activities in Window 2, causing its last consumer to be scheduled on Step 4. Now, since this exceeds $s(1) + \lambda(1) = 3$, the dependency of Window 2 on Window 1 causes the worst case propagated delay for Window 2 to become 1.

We note that it is possible to obtain more aggressive estimates for propagated delays, however we have found the above to be adequate so far.

[3]As discussed in the sequel, we will initially set all worst case propagated delays to 0.

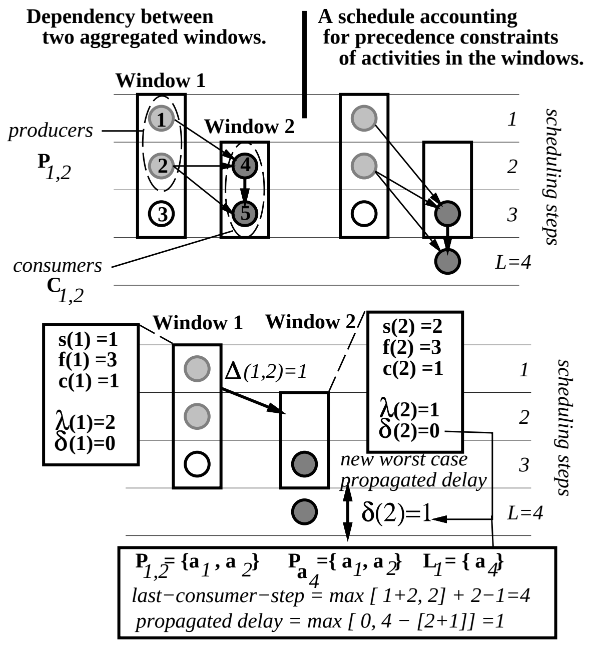
Dependency between two aggregated windows.
A schedule accounting for precedence constraints of activities in the windows.
Window 1
Window 2
producers
$P_{1,2}$
consumers
$C_{1,2}$
1
2
3
4
5
scheduling steps
L=4
s(1) =1
f(1) =3
c(1) =1
λ(1)=2
δ(1)=0
Δ(1,2)=1
s(2) =2
f(2) =3
c(2) =1
λ(2)=1
δ(2)=0
new worst case propagated delay
δ(2)=1
$P_{1,2}$ = {a_1 , a_2} P_{a_4} ={ a_1, a_2} L_1= { a_4}
last−consumer−step = max [1+2, 2] + 2−1=4
propagated delay = max [0, 4 − [2+1]] =1

Figure 4: Window dependencies and propagated delays.

3.5 Construction of the Window Dependency Graph

Let $W = \{1, \ldots n^A\}$ be an index set for the aggregated windows associated with the problem. We define a *window dependency graph* (WDG), $G(W, D)$, with n^A nodes representing aggregated windows, and including directed arc's $D \subset W \times W$ between nodes (aggregate windows) that depend on one another. That is, $(i, j) \in D$ if window j depends on window i. However, to avoid cycles, not all dependencies, i.e., arcs, are included in the graph. The following rule is used to prune edges.

Pruning Rule: Prune $(i, j) \in D$ if no producer activity can be executed on the first step $s(i)$ and/or last step of window i or if no consumer activity can be executed on the first step $s(j)$ of window j. That is, either $s(i) < \min_{a_k}\{s^I(k) | a_k \in P_{i,j}\}$ and/or $f(i) > \max_{a_k}\{f^I(k) | a_k \in P_{i,j}\}$ and/or $s(j) < \min_{a_k}\{s^I(k) | a_k \in C_{i,j}\}$, where $s^I(k)$ is the scheduling step for activity $a_k \in A$ in the ASAP schedule.

The intuition underlying this rule is that the dependency (arc (i, j)) should only be retained if, among all aggregate windows containing the same set of producer activities $P_{i,j}$, window i has the largest lower limit on its scheduling range, i.e., $s(i)$. Indeed, dependencies from aggregate windows starting earlier can be easily shown to result in the same or smaller worst case propagated delays, thus removing such dependencies will not compromise our lower bound on latency. Note, however, that our rule may actually remove more dependencies than those associated with aggregate windows including activities $P_{i,j}$ but starting the latest. Indeed, in some cases an aggregate window including a specific set of producer activities $P_{i,j}$ may not include a producer activity that can be executed on the first step of the window. A similar intuition accompanies the case in looking at consumers in the dependent window j. While in some cases this pruning may weaken the resulting bounds, it allows us to easily establish that the pruned WDG is acyclic, see the appendix for a proof. This in turn significantly reduces the complexity of our proposed algorithm.

Theorem 3.1 *A window dependency graph $G(W,D)$ pruned according to the above rule is acyclic.*

3.6 Algorithm to compute propagated delays

Given an acyclic window dependency graph $G(W,D)$, we next discuss how to compute the worst case propagated delay for all windows in the graph. We first set $\delta(j)=0$ for all $j \in W$. Then, starting from the source nodes (aggregated windows) in the window dependency graph, we iteratively determine the worst case propagated delay of each node j, $\delta(j)$, not yet considered, but whose parent nodes' worst case propagated delays are known, via

$$\forall i \text{ s.t. } (i,j) \in D \;:\; \textbf{propagated-delay}(i,j).$$

The propagated delay for each source node is assumed to be 0 upon initialization.

Theorem 3.2 *This iterative algorithm returns a set of propagation delays $\{\delta(i) | i \in W\}$ for windows in the graph.*

The proof of this theorem follows directly from Lemma 3.2.

The final lower bound, L^*, on the execution latency of the DFG, is given by the worst case lower bound over all windows in the WDG, i.e.,

$$L^* = \max_i \{s(i) + \lambda(i) + \delta(i) | i \in W\}.$$

The complexity analysis of the algorithm for computing propagated delays and L^* can be found in the next section.

3.7 Complexity analysis

In what follows we briefly discuss the asymptotic time complexity of the algorithms for creating the WDG and computing L^* for the WDG. The set of individual windows is created using ASAP and ALAP scheduling algorithms, and thus takes $O(|A|+|E|)$. Since the maximum number of edges incident on each activity (i.e., number of operands) is two, $|E| \leq 2*|A|$, and thus the generation of individual windows takes $O(|A|)$.

Next we discuss the generation of aggregate windows.[4] Note that the maximum number of aggregate windows per resource is given by $\sum_{i=0}^{L-1}(L-i)(i+1) \approx L^3$. Indeed for each resource, one can have at most L windows of size 1, $L-1$ windows of size 2, down to 1 window of size L. The simple algorithm currently used to create the aggregate windows is as follows. For each resource, we create a list of L^3 empty candidate aggregated windows, with corresponding ranges, ordered by start time. Each candidate aggregate window has a set of steps, from start step s to finish step f. Each such step is initialized as unused, and a window's local counter of unused steps is initialized to the number of steps contained in its range. In the first phase of the algorithm, for each individual window, we search for all candidate aggregate windows (defined for the corresponding resource) that contain its scheduling range. Whenever one is found, the individual window's activity is inserted in the aggregate window, and all steps that the individual window shares with the candidate aggregate window that are currently unused are marked as used. The counter of unused layers for the candidate aggregate window is then updated. This first phase takes $O(|A|L^4)$, since each of the $O(|A|)$ individual windows needs to iterate though the $O(L^3)$ candidate aggregate windows of its corresponding resource, and update unused layers at a cost of $O(L)$. In the second phase of the algorithm, each resulting candidate aggregate window is validated, by checking if its counter of unused layers is zero. If not, the candidate aggregate window is invalid, and is deleted from the ordered list of aggregate windows for the resource. If the candidate aggregate window is valid, we perform the ASAP schedule for the induced subgraph associated with the activities in the window, and compute the local delay $\lambda(i)$ of the window - the complexity of this step is $O(|A|)$. The second phase of the algorithm has a complexity of $O(|R|L^3|A|)$ since $O(|R|L^3)$ tentative aggregate windows must be considered.[5] The final number of aggregate windows is $O(|R|L^3)$.

Next we consider the algorithm for creating the pruned WDG's edges, and simultaneously computing the propagated delays between all aggregate windows. The worst case propagated delays for each window are first set to 0. We then sequentially consider the aggregate windows of all resources, ordered by start time. Suppose aggregate window j is selected for consideration, we shall call it the pivot. Next we select a candidate producer window for the pivot. (Due to the pruning rule, only aggregate windows whose start time is less than that of the pivot can be selected.) Next one verifies if the pruning condition holds (which takes $O(|A|^2)$) in which case the edge is not constructed between the aggregated windows and the next candidate producer window is considered. Otherwise, an edge (i,j) is created, and the algorithm for computing the pairwise propagated delay $\Delta(i,j)$ described in §3.4, is executed, and the value is associated with edge (i,j).[6] If the new pairwise propagated delay is greater than the current worst case propagated delay $\delta(j)$ of the pivot window, the value is updated. The algorithm to update worst case propagated delay of the pivot for a given candidate producer takes $O(|A|^2)$. Thus the computation of the bound (and simultaneous generation of the edges in the WDG), is done by applying the previous step to pairs of aggregate windows, and takes $O(|R|^2L^6|A|^2)$. In summary, the generation of the WDG and the computation of L^* have an asymptotic complexity of $O(|R|^2L^6|A|^2)$.

For VLIW datapaths with multiple functional units (intended to explore parallelism in the DFG), L is typically much smaller than $|A|$. Moreover, the number of aggregated windows that needs to be considered in the various steps of the algorithm has in practice been (and is expected to be) much smaller than $|R|L^3$.[7] Thus, we expect the above theoretical asymptotic complexity to be very pessimistic for the class of problems of interest. For all the DSP benchmarks considered in §5, the total execution time has never exceeded 0.5 sec on an UltraSparc 1.

4 Window dependency graph and tradeoff exploration

In this section we discuss a simple binding heuristic which takes advantage of the window dependency graph (WDG) to explore tradeoffs between 1) reducing data transfers and 2) avoiding operation serialization, see §1. The experimental results in §5 exhibit the effectiveness of this heuristic based on the WDG, which in turn could be used by an iterative improvement binding algorithm.

As a starting point in the generation of our examples, we considered an initial binding that reduced moves between operations on the longest paths of the DFG. The idea is to bind activities on those paths such that their shared data objects remain on register files shared by the assigned functional units. The remaining binding of operations to functional units was performed to minimize serialization of concurrent operations. This process was done manually.

[4]For most practical cases, we expect that the intermediate step of generating basic windows will pay off, i.e., improve the overall efficiency of the algorithm, since it may significantly reduce the number of windows that need to be individually considered in the expensive merging step that follows. However, for the purpose of determining asymptotic complexity since one would still need to consider $|A|$ basic windows, the basic window generation step will be omitted in this analysis.

[5]Note that this second step of the generation of aggregate windows can (and should) be actually integrated in the final phase of the algorithm, but for clarity of the explanation, we consider it here independently.

[6]Note that the computation of $\Delta(i,j)$ for the WDG edges is truly not required for computing L^*. However, these values are informative if one wants to reason about binding modifications likely to improve latency (see discussion on §4 and §5).

[7]In practice, it has been consistently sub-quadratic in L.

Next, based on the window dependency graph, we determined our lower bound L^* on latency. If $L^* = L$, and L is in fact equal to the last step of the ASAP schedule for $G(A,E)$ (see §3), then the current binding is optimal[8]. Otherwise it may be desirable to *modify* the functional unit assignment to try to lower execution latency. Recall that each aggregate window i has a scheduling range $[s(i), f(i)]$, a local delay $\lambda(i)$, and a worst case propagated delay $\delta(i)$ such that $s(i) + \lambda(i) + \delta(i)$ is a lower bound on the last step activities in the window will be scheduled. We shall refer to the difference between this bound and $f(i)$ as the window's *excess delay*. The key insight in selecting which activity bindings to modify is to 1) find windows with *high positive* excess delays that 2) lie on "critical paths" of the WDG. Recall that a window represents a set of activities bound to a common resource that have to be (serially) executed over a given scheduling range. A window with a large positive excess delay is one for which serialization due to resource constraints and/or pairwise propagated delays from parent windows, Δ, lead to delays beyond this scheduling range. Thus, in order to reduce latency it may be worthwhile to reconsider the binding of activities in such windows. Note, however, that not all such windows are problematic. Indeed, only windows on the "critical paths" of the WDG, i.e., those leading to an increased overall latency, either directly or through a sequence of pairwise propagated delays, need to be considered. We identify "critical paths" on the WDG by backtracking from sink nodes (windows) in the WDG whose *final lower bound* on execution exceeds the global lower bound L, and traverse the graph up to parent windows with non-zero excess delays.

Still, not all windows with positive excess delay, and lying on the WDG's critical paths, would be candidates for iterative improvement on binding. Two simple rules can be used to determine windows for which a given binding is likely to be optimal. First, a window with no additional delays propagated from its producer windows and with an excess delay ≤ 1 need not have the binding of its activities reconsidered. Indeed, as shown in the example in Fig.2, the benefits of removing serialization in such cases will be canceled by the additional delay incurred by required move operations. Similarly, a window with a non-zero propagated delay from its producer windows and an excess delay ≤ 2 need not have the binding of its activities reconsidered. It follows that a WDG that only contains such windows is unlikely to have its latency improved by further modifying the binding. These simple heuristic rules proved to be effective when applied to the benchmarks in §5.

This concludes our brief qualitative discussion. As mentioned above, the purpose of this section is not to propose an algorithm to perform this complex trade-off exploration, but rather to show that the information contained in the WDG can be helpful to such an exploration process.

5 Related work and benchmark examples

In the context of distributed register files, if one wants to consider the deleterious effect of required data object moves on the latency of a schedule, one must explicitly consider a binding of the dataflow nodes to the functional units in the datapath. The basic problem formulated and addressed in this paper is thus different from those considered in [6, 11], for they assume no data transfer delays. However, one can apply these techniques to the dataflow after a binding function has been determined. Indeed, by making each functional unit a distinct resource type with capacity 1, and the bus a resource type with a specific capacity, these methods can also be made binding specific. Given this, one can compare the absolute quality of our lower bound with that reported in [6, 11]. With few exceptions [11] performs better than [6], thus we shall compare our work with an implementation of the algorithm in [11].

[8]Optimal at our level of abstraction, i.e., disregarding register files sizes and port assignments.

Table 1 summarizes our results. Several benchmark dataflows were bound to the datapath shown in Fig.1. Initial and improved bindings were obtained manually based on the simple heuristics discussed in §4. Columns 2 and 4 of the table show the minimum achievable latency for centralized and for distributed register file structures, respectively. Differences between these indicate the crudeness of assuming a centralized register file structure when it is in fact distributed. Starred entries are known to be optimal latencies over all possible bindings, thus the improvement heuristic was effective.

Our lower bound on latency L^*, shown in column 5, was consistently tight and for seven of the ten benchmarks outperformed [11].

DFG	**Central. RF**	**Binding**	**Distrib. RFs**	**Lower Bds** Our L^*	[11]
FFT Butterfly [3]	4	initial	8	8	6
		imprvd.	5*	5	4
4th order Avenhous Filter [5]	7	initial	10	10	9
		imprvd.	9*	9	9
4th order IIR Filter retimed [3]	4	initial	9	9	8
		imprvd.	6*	6	5
Beamforming Filter (3 beams) [9]	4	initial	8	8	7
		imprvd.	6*	6	5
AR Filter [2]	8	initial	15	13	14
		imprvd	13	13	13

Table 1: Experimental results.

In addition, note that [6, 11] only generate bounds on the earliest possible execution time of *individual* nodes in the DFG, so, the information on serialization (for FUs and buses) that we capture via the WDG is not available. Since the latency of a schedule can *vary significantly* for different bindings, particularly for datapaths with distributed register files, our approach has a significant added value, in that it can provide guidance on how to modify binding functions to achieve lower latencies.

Code generation for VLIW ASIPs has been addressed extensively in the literature, see e.g., [8, 7]. Although discussing this work is beyond the scope of this paper, to further illustrate the relevance of the trade-off information captured by the WDG, we will briefly discuss the AVIV code generator[4]. This work specifically considers the same trade-offs, while deriving a functional unit binding/assignment for a given expression tree.

As discussed below, AVIV greedily prunes binding alternatives based on a *local* cost function. Given an expression tree, an ASAP schedule of the expression tree is performed, and nodes (operations) on the resulting levels are sequentially considered (in any order) from the lowest to the highest level. As the operations are considered, a search tree is constructed, representing possible binding alternatives. Heuristically inferior alternatives are immediately pruned - based on a *local* cost function. The cost associated with binding an operation to a functional unit is the sum of 1) the number of required data transfers given the bindings made for the ancestor nodes of that particular path of the decision tree, and 2) the number of operations at the current level that are assigned to the same functional unit, again considering the bindings for the ancestor nodes. While this greedy policy would execute faster than our lower bound algorithm, it makes decisions strictly based on local information. Thus, for example, it does not discriminate among operations that have different mobility (i.e., scheduling windows), which can compromise the overall quality of the binding. An iterative improvement algorithm using the WDG can instead create binding alternatives based on a more "global" view of such trade-offs, at the expense of an increase in runtime. This concludes our discussion of the relevance to code generation of the tradeoffs explicitly modeled in our approach.

6 Conclusion

We have proposed an approach to generating lower bounds on execution latency for DFGs on datapaths typical of VLIW ASIPs for a given functional unit binding/assignment. While the bound was found to be competitive with state-of-the-art approaches, its key advantage lies in capturing delay penalties due to operation serialization and/or data moves among distributed register files. In order to estimate such delays, the scheduling problem is relaxed (decomposed) into a number of simpler scheduling sub-problems, jointly represented using the window dependency graph model. Our results show that the relaxed, less computationally expensive, version of the scheduling problem results in tight bounds. Moreover, it can provide valuable information/guidance to heuristic binding algorithms for "clustered" VLIW ASIP datapaths. Functional unit assignment/binding is a key step of the difficult code generation problem for VLIW ASIPs. We are currently working on developing binding algorithms, supported by the window dependency graph mode, to address this problem.

References

[1] G. de Micheli. *Synthesis and Optimization of Digital Ciruits*. McGraw-Hill, Inc, 1994.

[2] R. Jain et. al. Experience with the Adam synthesis system. In *Proc. of DAC*, pages 56–62, 1989.

[3] V. Zivojnovic et. al. DSPstone: A DSP oriented benchmarking methodology. In *Proc. of ICSPAT'94*, Oct. 1994.

[4] S. Hanno and S. Devadas. Instruction selection, resource allocation and scheduling in the AVIV retargetable code generator. In *Proc. of the 35th DAC*, pages 510–15, June 1998.

[5] E. Ifeachor and B. Jervis. *Digital signal processing: A practical approach*. Addison-Wesley, 1993.

[6] M. Langevin and E. Cerny. A recursive technique for computing lower-bound performance of schedules. *ACM Trans. on Design Automation of Electronic Systems*, 1(4):443–56, 1996.

[7] C. Liem. *Retargetable compilers for embedded core processors*. Kluwer Academic Publishers, 1997.

[8] P. Marwedel and Gert Goossens, editors. *Code Generation for Embedded Processors*. Kluwer Academic Publishers, 1995.

[9] R. Mucci. A comparison of efficient beamforming algorithms. *IEEE Trans. on Signal Processing*, 32(3):548–58, 1984.

[10] M. Rim and R. Jain. Lower bound performance estimation for the high-level synthesis scheduling problem. *IEEE Trans. on CAD of ICs and Systems*, 13(4):451–58, 1994.

[11] G. Tiruvuri and M. Chung. Estimation of lower bounds in scheduling algorithms for high-level synthesis. *ACM Trans. on DAES (TODAES)*, 3(2):162–80, 1998.

A Proof of Lemma 3.1

The main idea underlying this lemma is that any relaxation of constraints, e.g., precedence or resource constraints, on the original resource constrained scheduling problem can only reduce the starting time of an activity in the corresponding optimal schedule. Hence, consider the subgraph $G(A', E')$ induced by the set of activities A', i.e., including only arcs in the original graph that are between activities in A'. This subgraph corresponds to a relaxation of all precedence constraints external to the set of activities A'. Next we perform an ASAP scheduling for the DFG $G(A', E')$ and let $l = 1, \ldots m$ denote the steps in this schedule, and n_l denote the number of activities scheduled on step l. Since these activities are to be executed on a resource r with capacity $c(r)$ the above ASAP schedule may not be feasible. To obtain a lower bound on necessary delay penalties due to the resource constraints we consider a new hypothetical resource constrained scheduling problem which further relaxes internal precedence constraints among the activities in A'. We assume that once an activity on step l of the subgraph's ASAP schedule is executed all n_{l+1} activities on step $l+1$ can be scheduled on the subsequent step.

This new hypothetical problem can be solved directly using a greedy algorithm that schedules activities as soon as possible. Let x_l denote the number of activities that are eligible for execution prior to step l but, due to capacity constraints, will need to be scheduled on step l or later. Thus on step l the total number of activities eligible for execution is $n_l + x_l$, however only $c(r)$ can be scheduled, thus x_{l+1} (see Eq. 1) activities will be postponed to the next step. Naturally since the schedule starts on step 1, $x_0 = 0$. Note that which activities are are actually scheduled on a given step is irrelevant, since we can always assume that at least one actually belongs to step l of the ASAP schedule, and thus all activities on the next step will become eligible for execution. The iterative computation in (1) finishes on step m where x_{m+1} corresponds to the number of activities that had to be postponed, if any, beyond the last step m of the ASAP schedule due to resource constraints.

$$\begin{aligned} x_{l+1} &= \max\{n_l + x_l - c(r), 0\}, \quad l = 1, \ldots m, && (1)\\ \text{bound}(A', r) &= \lceil \frac{x_{m+1}}{c(r)} \rceil + m - 1. && (2) \end{aligned}$$

From there on we can compute the additional number scheduling steps required to execute the postponed activities, if any, i.e., $\lceil \frac{x_{m+1}}{c(r)} \rceil$ Finally, to obtain our bound we subtract 1 since the bound is on the number of *additional* steps beyond the first one, that are required to execute the activities.

B Proof of Lemma 3.2

The goal of **propagated-delay** is to find a lower bound on the last step on which activities in Window 2 will be executed.

We first consider lower bounds on the time the last producer activity in Window 1 is scheduled. If $A_1^a = P_{1,2}$ then, by definition of the local delay and worst case propagated delay of Window 1, the last activity must be scheduled on or after step

$$\text{last-producer-step} = f(1) + \lambda(1) + \delta(1).$$

If $A_1^a \neq P_{1,2}$ then, using the result in Lemma 3.1, the last producer must be scheduled on or after step

$$\text{last-producer-step} = \text{start-step} + \text{bound}(P_{1,2}, r(1))$$

where start-step $= \min_{a_k}\{s^I(k) | a_k \in P_{1,2}\}$ corresponds to the earliest possible step on which an activity in $P_{1,2}$ may be scheduled. Now, since at least one consumer activity in Window 2 depends on the last producer activity, the last consumer step must strictly exceed the last-producer-step computed above. In fact there are at least

$$\text{num-consumers-for-last-producer} = \min_b\{|C_b| \mid b \in P_{1,2}\}$$

consumers depending on the last producer. Thus we set the "delay" variable equal to

$$\text{delay} = \lceil \text{num-consumers-for-last-producer}/c(r(2)) \rceil,$$

so the last consumer step must exceed the last-producer-step + delay.

Next we find a lower bound for the last step on which an activity in the dependent Window 2 will be executed. Let $G(A_2, E')$ be the subgraph of $G(A, E)$ which includes the activities A_2 and all the edges $E' \subset E$ among these activities. Suppose we perform an ASAP schedule for this subgraph, and let L_2 denote the set of activities on the first step of that schedule. Also for any activity $a \in A_2$, let P_a denote its producer activities in Window 1, i.e., $P_a = \{b \in A_1 | (b, a) \in E\}$.

We consider two cases. We first test if $c(r(1)) = 1$ and $\forall a \in L_2, |P_a| = 2$. Since every activity in L_2 depends on two producer activities in Window 1 and the capacity of the resource associated with the producer window is 1, no activity in the dependent Window 2 can begin execution prior to step $s(1) + 2$ or, of course, its own starting step $s(2)$. Thus the following lower bound follows immediately from Lemma 3.1:

$$\text{last-consumer-step} = \max\{s(1) + 2, s(2)\} + \lambda(2).$$

Note that due to the pruning rule discussed in §3.5, $s(1) + 1 \leq s(2)$ thus when $\forall a \in L_2, |P_a| \geq 1$ the the analogous bound to the above would degenerate to $s(2) + \lambda(2)$, i.e., would leave the current propagated delay of the window unchanged.

If the condition for the previous case is untrue then we make the optimistic assumption that activities in Window 2 can begin execution on the first step of the window $s(2)$, even though there may be dependencies on Window 1. This gives the following bound

$$\text{last-consumer-step} = s(2) + \lambda(2) + \delta(2).$$

Thus we have two lower bounds for the step on which the last activity in the dependent window is executed.

Finally, we take the maximum of these two bounds, i.e.,

$$\begin{aligned} &\text{last-consumer-step} = \\ &\quad = \max\{\text{last-producer-step} + \text{delay}, \text{last-consumer-step}\}. \end{aligned}$$

The pairwise propagated delay associated with Window 2's dependency on Window 1 is then given by

$$\Delta(1,2) = \text{last-consumer-step} - [s(2) + \lambda(2)].$$

The worst case propagated delay associated with Window 2, $\delta(2)$, is then updated by taking the worst of the old propagated delay, and the just computed pairwise propagated delay

$$\delta(2) = \max\{\delta(2), \Delta(1,2)\}.$$

C Proof of Theorem 3.1

We shall prove the theorem by contradiction. Suppose there exists a cycle in the pruned window dependency graph $G(W, D)$. Without loss of generality suppose the cycle visits nodes (windows) $1, 2, 3, ..j$ and then back to 1. Given our pruning rule, aggregate Window 1 must have a producer activity, say $a_1 \in P_{1,2}$, that can execute on the last step $f(1)$ of the window's scheduling range. Thus $f(1)$ would correspond to position (step) of a_1 in the ALAP schedule used to define that activity's individual window. Since Window 2 contains at least one activity b_2 that depends on a_1, in the same ALAP schedule b_2 must be scheduled on a step beyond $f(1)$. Thus the final step $f(2)$ in the scheduling range of Window 2 must satisfy $f(2) \geq f(1) + 1$. Using this same argument until we reach Window j we can show that $f(j) \geq f(1) + j - 1$. Since Window 1 also depends on Window j, the pruning rule guarantees that at least one producer activity $a_j \in P_{j,1}$ in Window j can execute on step $f(j)$. Now, since there exists an activity in Window 1 that depends on a_j, Window 1's last step $f(1)$ must be at least $f(j) + 1$. Clearly this is a contradiction since this would imply that $f(1) \geq f(j) + 1 \geq f(1) + j$.