Scheduling and Binding Bounds for RT-Level Symbolic Execution*

Chuck Monahan Forrest Brewer

Department of Electrical and Computer Engineering
University of California, Santa Barbara, U.S.A.
chuck@bears.ece.ucsb.edu, forrest@ece.ucsb.edu

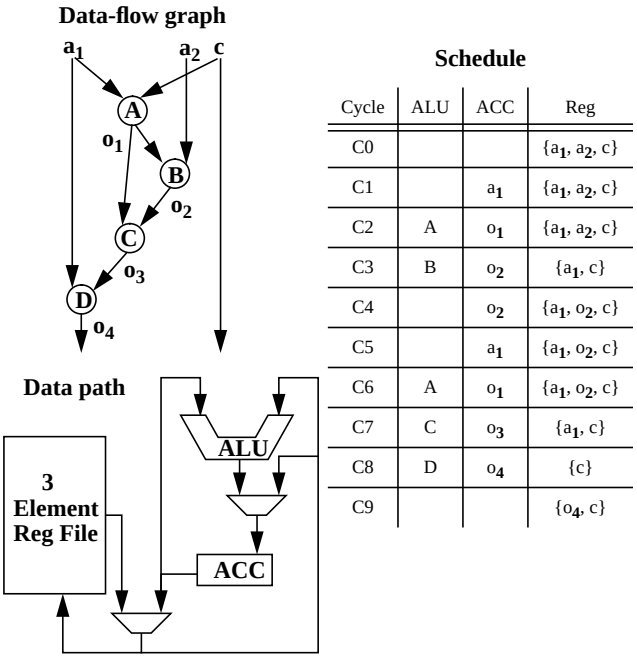
Abstract

This paper generalizes ALAP bounds for the exact scheduling problem on a pre-defined data path. Conventional bounds are inapplicable because of the possible requirement of re-computing operands for minimal schedule length. Efficient techniques are presented for constructing the new bounds which are sensitive to point-to-point delays via transitive memory units. An efficient operand mapping bound is also described. Based on these two bounds, time improvement factors of 50 have demonstrated in exact scheduling results.

1. Introduction

HLS techniques are typically aimed at relatively unconstrained synthesis tasks such as RT synthesis of a new design. However, the large cost of verification and the gaining popularity of data-path cores creates the need for constrained synthesis for a given (predefined) data-path. Miyazaki[4] proposed the first system for this task, however, that system used heuristic techniques which missed much of the inherent design freedom. In a similar vein, retargetable microcode compilers make limited use of the total possible connectivity of a data-path by restricting the communications to a small set of pre-defined µOps. Scheduling for these constraints can be managed by ASAP[4] or by tree matching[2,3]. By contrast, we proposed using RT-level symbolic data path execution to perform exact scheduling of DFG's (data-flow graphs) on a preexisting data path.[5] This technique utilizes reachable state analysis on an automata representing both the data path and operand constraints to perform a systematic analysis of the design freedom. The price of this increased design exploration is its substantial computational complexity since at each step, every feasible automata state that represents possible generation of operands is implicitly constructed. This process implicitly constructs every feasible schedule and is exact, however, it is potentially costly. In this paper, we describe novel scheduling bounds which allow exact results for the data-path constrained scheduling problem, but which substantially decrease the run time by removing states which can lead only to suboptimal schedules.

Reachable state analysis had been previously utilized for scheduling in the work of Yang [8] and of Coehlo [1], but these techniques were addressing scheduling without communication constraints. Scheduling for a pre-defined data path, however, presents new challenges. Fundamentally, the system cannot guarantee that an operand will be created only once in an optimal schedule. This is true even for optimal schedules. An operand may require recomputation when insufficient storage or switching logic exists, as occurs in the example shown in Fig. 1. In this figure, while executing the DFG, o_1 must be overwritten when o_2 is computed in order



Schedule

Cycle	ALU	ACC	Reg
C0			{a_1, a_2, c}
C1		a_1	{a_1, a_2, c}
C2	A	o_1	{a_1, a_2, c}
C3	B	o_2	{a_1, c}
C4		o_2	{a_1, o_2, c}
C5		a_1	{a_1, o_2, c}
C6	A	o_1	{a_1, o_2, c}
C7	C	o_3	{a_1, c}
C8	D	o_4	{c}
C9			{o_4, c}

Figure 1. Scheduling example.

to maintain the required set of operands. This potential for rescheduling operations means that no operation can be excluded solely on the merit that it has been previously scheduled. It is this exclusion of operations and their associated parent operands which permits conventional bounds to maintain a manageable size for the reachable state analysis.

Clearly, an operation need not be considered for (re)scheduling if it could not have a measurable effect in the remaining execution cycles. These constraints on the scheduling of an operation are commonly referred to as ALAP (as late as possible) bounds. The use of ALAP bounds has been shown to be extremely effective for scheduling problems which lacked communication constraints.[7] The construction techniques for ALAP bounds must address two additional issues to make them applicable for scheduling on a predefined data path. First, due to the possible recomputation of operands, the ALAP bound of an operand must be based on the ALAP bounds of every child operation since we cannot guarantee that the first instance of an operation will satisfy all dependencies. The example in Fig. 1 demonstrates this requirement, where the o_1's ALAP bound is derived from the ALAP bound of o_3 instead of o_2. Secondly, the bounds must be extended to account for the limitations of the switching network, memory units and function units, instead of solely the number of function units as is conventionally done. The first issue is applicable to the problem, while the second

* This work has been generously supported by UC MICRO 96-142 and Mentor Graphics Corp.

TABLE 1. Behavioral Constraints

Behavior	Restrictions
Latch	$\|\Phi_i\| = 1$, $\|\Theta_i\| = 1$, and $\Sigma_i = \varnothing$
Register file	$\|\Sigma_i\| = \|\Theta_i\|$
Multiplexer	$\|\Phi_i\| > 0$, $\|\Theta_i\| = 1$, and $\|\Sigma_i\| \geq \log_2(\|\Phi_i\|)$
Function unit	$\|\Theta_i\| > 0$

issue makes the bound useful.

This paper presents a technique for constructing ALAP bounds based on operand routing constraints instead of function unit availability. This same technique is subsequently extended to further restrict the reachable state analysis by providing bounds on storage locations. Both of these topics are presented after a brief review of symbolic data path execution. This review consists of a description of the input requirements (Section 2) in which the flexibility of the representation is highlighted, an overview to the automata representation and reachable state analysis (Section 3.1), and an introduction into the incorporation of operand lifetime bounds (Section 3.2).

2. Input Formats

In this section, the format for specifying the data path and DFG are presented. The formats were selected to permit the specification of a wide variety of designs. The input format does include a few restrictions which are designed to clarify behavior that would be otherwise ambiguous.

2.1. Data path

The data path is modeled as a tuple (C, W). Each element, c_i, of C is a data path component defined by $(\Sigma_i, \Phi_i, \Theta_i)$. The set Σ_i defines the set of control lines which connect to component c_i. The set Φ_i defines an ordered set of unidirectional input ports for c_i. The final set, Θ_i, defines for c_i an unordered set of unidirectional output ports.[1] While two components may share common control lines, they must always have disjoint input and output port sets. The function $C(\theta)$ will be used to identify the associated component from an output port specification.

A number of useful data path attributes may be gathered from these definitions. The set Σ describes the complete set of control lines, $\{\sigma_1,\sigma_2,...,\sigma_n\}$, as defined by $\Sigma = \bigcup_i \Sigma_i$. The set Θ describes the complete set of output ports, $\{\theta_1, \theta_2, \ldots, \theta_m\}$, as defined by $\Theta = \bigcup_i \Theta_i$.

Operands are transported between output and input ports over the data path's set of wires, W. We impose the constraint that each wire, w_i, emanates from only one output port but may fanout to drive many input ports.[2] Therefore, each w_i is defined by an output port, θ_i, and a set of input ports. Additionally, each input port, ϕ, may only be driven by a single wire, which permits the function $W(\phi)$ to uniquely identify the wire connected to a specified input port.

Each data path component is assigned one of the four behavior types. Memory elements are represented by either latches or register files. Switching logic, used to conditionally transfer existing operands to different wires, is distinguished from combinational logic which creates new operands. In general, all switching components are modeled as multiplexers, and combinational logic blocks are referred to as function units. While modeling of external input and output ports pose no additional challenges for the proposed techniques, they are omitted for the sake of clarity.[5] The arrangement of these components and their connecting wires must ensure that each loop described by a consistent set of directional ports contains at least one memory device to prevent feedback races. All additional constraints are based upon the component's behavioral type and are summarized in Table 1. The behavior of many conventional data path components will not directly correspond to one of these base behaviors. Such components need to be either manually or automatically partitioned into the various functional components connected by wires as in the example in Fig. 2.

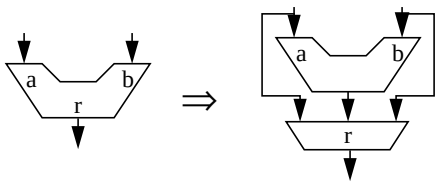


Figure 2. Expressing a function unit's feed-through capacity.

2.2. Data-Flow Graph

Data-flow graphs specify the dependencies between operands and operations. For our system, these graphs form directed, acyclic hypergraphs. A DFG is a tuple (P, E) where P is a set of operands and E is a set of operations. Each operation is defined as the four-tuple $e(\theta, \vec{\sigma}, \Pi, p)$ where θ is an output port which will produce the result, $\vec{\sigma}$ is the control vector for the component, $C(\theta)$, Π is an ordered set of input operands, $(\pi_1, \pi_2, \ldots, \pi_n)$, where $\pi_i \in P$, and p is the resulting operand. The following restrictions are placed on operation specifications: 1) $C(\theta)$ must identify a function unit and 2) $|\Pi|$ must equal the number of input ports for the specified function unit. This second restriction permits a direct assignment of input operands to input ports; a "null" operand is used as a place holder for input ports with no associated input operand in a given operation. An operand p_1 is said to be a parent of operand p_2 and p_2 is said to be the child of p_1 iff $\exists e_i \in E \mid p_1 \in \Pi_i \cap p_2 = p_i$.

There are some non-traditional elements of our DFG model. 1) There are no restrictions on the number of operations which may generate any operand p_k as expressed in EQ. 1. In the presence of multiple operations, each operation provides a unique, alternative method to generate the operand.

$$|\exists e_i \in E \mid p_i = p_k| \geq 0 \quad (1)$$

2) The operation mapping explicitly lists a function unit's output port. Traditionally, this association is made by an operation map. But, the large disparity in function unit descriptions combined with the potential for highly tailored operations made such an operation table impractical. The enumeration of the associative and commutative operands as well as equivalent function unit listings, which are traditionally handled by the operation map, is accommodated through the use of alternative operations. 3) No two operands may be equivalent, where equivalency between two operands p_1 and p_2 is defined by EQ. 2. When equivalent operands are detected, they should be merged into a single operand; this can be done either automatically or manually.

$$\exists e_i, e_{j \neq i} \in E \mid (\Pi_i = \Pi_j, p_i = p_1, p_j = p_2, \theta_i = \theta_j, \vec{\sigma}_i \cap \vec{\sigma}_j \neq \varnothing) \quad (2)$$

3. Representing Data paths

This section will introduce our automata-based data-path model, discuss applications which may utilize it, and finally discuss a variety of optimizations and performance improvements.

1. Bidirectional ports are modeled by combinations of unidirectional ports, switching elements, and switching control restrictions.
2. Designs which drive a line from multiple sources typically utilize coordinated switching elements. Such designs are accommodated by merging these switching elements into a single functional component with a single source.

3.1. Automata model

A symbolic automata is used to represent the storage of operands in memory components, the motion of operands on the switching network, and the creation of operands in function units. In its most general form, this automata is defined by the five-tuple $(V, \Sigma, N, S_0(V), S_f(V'))$.

V represents a finite set of states. Each state represents the contents of each of the data path's memory components. This set may be partitioned into the various disjoint components $V = V_1 \times V_2 \times \ldots \times V_n$ where V_i is the contents of a single memory device. In general, $V_i \subseteq P_i$, where $P_i = P$. We define the set of variables V as the present state variables and create a second set of variables V′ for the next state variables.

The automata inputs are defined as the set of control lines, Σ, introduced in Section 2.1. State relations are defined by the transform relation N: $V \rightarrow V'$. This relation maps the set of feasible next states, given the set of present states and all possible control line settings. This relation is symbolically represented as N(V, V′). While N(V, V′) describes the transform relation for the entire automata, separate transform relations may be defined for each portion of the state space denoted by N_i for $\Sigma \times V \rightarrow V'_i$. The use of the control lines permits the complete transform relation to be expressed as:

$$N(V, V') = \exists_{\sigma \in \Sigma} [\bigcap_i N_i(\Sigma, V, V'_i)].$$

Such relations are well defined for a given state and control vector because of the restrictions placed upon the input format. First, the restriction that each bus has a single source means that any control vector describes a set of distinct paths through the switching network. Restricting latches to single operands and the use of control lines to select operands from register files means that only distinct operands may appear at any path source emanating from a memory device. Restricting the DFG to contain only unique operations means that only distinct operands may be produced by function units given a set of distinct operands at the inputs and a control vector. In the absence of a direct mapping between a given state/control-vector pair and the operation for a given device, the operand produced by that device is a special "null" operand. The absence of cyclical paths ensures that each path destination will have a distinct operand associated with the path's source.

$S_0(V)$ and $S_f(V')$ represent a set of initial and final states for the automata. The ability to specify sets of initial and final states gives the designer greater flexibility in determining both the proper initial and final state for the model. Schedules which link a state in $S_0(V)$ with a state from $S_f(V')$ may be found using symbolic reachable state analysis from $S_0(V)$. Towards this end we compute $S_j(V)$, the set of reachable states on the jth iteration of the clock. In general, this set is generated by computing:

$$S_j(V') = \exists_{v \in V} [S_{j-1}(V) \cap N(V, V')].$$

The complexity of each $N_i(\Sigma, V, V'_i)$ may be reduced significantly by using the set of present states, making the following computation more feasible:

$$S_j(V') = \exists_{v \in V} \left[\exists_{\sigma \in \Sigma} [\bigcap_i [S_{j-1}(V) \cap N_i(\Sigma, V, V'_i)]] \right].$$

A *bounded minimum-cycle scheduler* is defined as a system which identifies the set of state transitions satisfying $S_j(V') \cap S_f(V') \neq \varnothing$ where j is minimized and below a specified upper bound. Upon j reaching this upper bound, the scheduler reports the infeasibility of the scheduling problem. The resulting schedules are extracted through a backwards-reachable-state analysis of the relation set generated by the reachable state analysis.

3.2. Lifetime optimizations

On a given iteration, each individual relation, N_i, need only represent the set of states reachable from $S_j(V)$. It is sufficient if the relation is defined over any $S''_j(V)$ where $S_j(V) \subseteq S''_j(V)$. Therefore, the set of relations $N^{\circ}_{i,j}(\Sigma, V, V'_i)$ is utilized instead of $N_i(\Sigma, V, V'_i)$ to represent the state relation $\Sigma \times S''_j(V) \rightarrow V'_i$. This new set of relations are generated dynamically for each iteration of the reachable state analysis.

The dynamic construction of transform relations is performed as follows. On each clock cycle, the set of operands, P, may be partitioned into a set of dead and a set of potentially active operands. We define an optimal set of dead operands, D_j, to include operands whose storage in some memory element at clock cycle k does not affect the reachable state analysis. The set of active states, A_j, is defined as $A_j \equiv P - D_j$. Given this partition, the individual transform relations may be constructed from the conditions under which operand p_k is present in device c_i, represented as $N'_{i,k}(\Sigma, V)$. The specification of $N^{\circ}_{i,j}(\Sigma, V, V'_i)$ is constrained by the set of active operands, as in:

$$\left[p_k \in \bigcap_{(A_j \cap P_i)} N^{\circ}_{i,j,k}(\Sigma, V, V'_i) \right], \text{ where } N^{\circ}_{i,j,k}(\Sigma, V, V'_i) = \quad (3)$$

$$[v'_{i,k} \cap N'_{i,k}(\Sigma, V)] \cup [\bar{v}'_{i,k} \cap ((\Sigma \times S_j(V)) - N'_{i,k}(\Sigma, V))].$$

While this representation is much smaller than the general transform, it requires a set of ordered operands for the set D_j. The following observations can be made about constructing such a set: If D_j is defined as the optimal set of dead operands, a suboptimal set D'_j may be constructed where $D'_j \subseteq D_j$. Such a set is suboptimal since $N^{\circ}_{i,j}(\Sigma, V, V'_i)$ will be more complicated than strictly required but it is still exact.

To construct the set D'_j, we define an operand's lifetime as the first cycle on which it may be scheduled (birth) and the last cycle on which one of its children may be scheduled (death). Such a lifetime constitutes the cycles during which an operand must be present in the set of active operands. Operand p_k should remain in the set of dead operands until the following test is passed.

$$\bigcup_i (N'_{i,k}(\Sigma, V) \cap S_j(V)) \neq \varnothing$$

Unfortunately, no such simple test exists to determine when p_k should return to the dead list.

3.3. ALAP bound generation

While lifetimes can not be determined exactly, they may be bounded relative to the minimal cycle bound provided to the scheduler. From such a limit, ALAP bounds can be derived for the operation set based on the routing restrictions imposed by the data path. From these ALAP bounds, an upper bound on the death of an operand may be determined. Since an operand is no longer required after all of its children have been produced, the operand p_k is effectively dead once the cycle equals:

$$\max_{e_i \in E \mid p_k \in \Pi_i} [ALAP(e_i)].$$

Towards this end, we note that the DFG as well as the set of final states create requirements on an operand's possible storage in the data path. For example, each operation specifies an operand and input port pair which must be satisfied in addition to the resulting operand and output port pair. Each final state encoding specifies where an operand must be stored from which a set of input ports can be determined. To meet the requirements of the final states or the operations, an operand must traverse the data path to get from where it was created by an operation to the proper input port.

The data path provides many obstacles to the movement of operands. The switching network will often support only a limited amount of connectivity. While operands do traverse memory devices, they will suffer the delay of at least one cycle. These limitations combined with the constraints derived from the simultaneous transfer of multiple operands account for a majority of the cycles in a schedule. By formalizing the minimal delay of a single operand traversing from specific locations, useful scheduling bounds may be derived.

We define the function $\tau(\theta_i, w_j)$ to represent the minimal number of memory devices which lay on a path between output port θ_i and wire w_j. To compute this function, we construct a series of sets, $\tau_{j,x}$, where each set indicates the output ports which can be connected to wire w_j by traversing no function units, x memory devices, and as many multiplexers as required. The definition of $\tau_{j,x}$ depends upon the component, c_k, which drives the wire as in:

$$\tau_{j,x} = \begin{cases} \bigcup_{\phi \in \Phi_k} \tau_{W(\phi),x} & c_k = \text{multiplexer} \\ \theta_i \cup \bigcup_{\phi \in \Phi_k} \tau_{W(\phi),x-1} & \text{where } c_k = \text{latch or reg. file} \\ \theta_j & \text{otherwise} \end{cases}$$

The output port, θ_i, of a memory device could be dropped, but they are included for use in Section 3.4. Additionally, we note that $\tau_{j,x}$ only needs to be defined for values of x from 0 to the number of memory devices. Given this series of sets, $\tau(\theta_i, w_j)$ returns the lowest value x, where $\theta_i \in \tau_{j,x}$.

Each operation is assigned an ALAP bound based on the time it takes to route the resulting operand to its specified position. These bounds start with the set of operations which produce operands which explicitly appear in the final state specification are evaluated. The bound for these operations is computed from the delay required to store the resulting operand, p_k, into the memory devices specified in the final state set. Since the user can specify multiple, alternative final states, each state is evaluated separately to determine which provides the smallest delay, as in:

$$\text{last cylce} - \{ \min_{S_j \in S_f(V)} \left[\max_{v_{i,k} \in S_j} (\min_{\phi \in \Phi_i} \tau(W(\phi), \theta)) \right] \}$$

The evaluation of each final state must ensure that the operand is sent to all of that state's specified memory devices, but the operand is allowed to utilize the input port which provides quickest route to that memory device. The ALAP bound for the remaining set of operations is defined by EQ. 4. This bound determines that last cycle on which the operation result could be used as an input operand. The notation, ϕ_{e_2, p_k}, used in EQ. 4 identifies the input port of c_j (where $c_j = C(\theta)$) associated with operand p_k.

$$\text{ALAP}(e_1) = \max_{e_2 \in E \mid p_k \in \Pi_2} [\text{ALAP}(e_2) - \tau(W(\phi_{e_2, p_k}), \theta)] \quad (4)$$

These ALAP bounds differ dramatically from the computation of traditional ALAP bounds. One difference is the fact that this proposed method utilizes resource constraints in terms of routing restrictions instead the number functional unit types. However, the central difference stems from the accommodation of operand recomputation. Because of this capacity, EQ. 4 must use the maximum ALAP bound instead of the minimal bound. Use of the minimal bound indicates the last cycle on which an operation could fulfill the timing requirements of all of the subsequent operations. Since we have no guarantee that this operation result will be used for all of the subsequent operations, we determine the last cycle on which the result can fulfill the requirements for one of the subsequent operations. This policy means that input operands may have an ALAP bound which is later than the resulting operand as shown

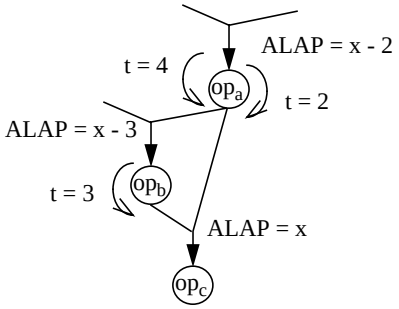


Figure 3. Fluctuating ALAP bounds due to operand fanout.

in Fig. 3. Fortunately, this policy accommodates the fact that not every operation must be executed since any operation may have an alternative.

3.4. Memory coloring

The advantage of the reachable state analysis which we have described is its exhaustive evaluation of the data path freedom. This can also be a disadvantage when the exhaustive nature evaluates the storage of operands in inappropriate memory devices. When this occurs, many states which are unproductive or inferior will be analyzed alongside productive and superior states thereby needlessly raising the complexity of the analysis. To limit this complexity, we shall identify which operands should never be stored in a particular memory device.

The DFG's association of operations and function units may be used to further constrain the construction of each $N'_{i,k}(\Sigma, V)$. The first observation is that the behavior of each function unit is limited to the set of associated operands. This notion of restrictions can be generalized to restrictions on the use of memory devices for storage of particular operands. For example, Fig. 4 depicts a component arrangement where operands in latch l must feed directly into the multiplier. Since use of the multiplier is only defined for a specific subset of operations, only those operands which would be defined as an input operand for such an operation need be stored in that latch. This underscores the basic principle that while a memory device may store any operand, there is an identifiable subset of operands which it need never store. We therefore redefine the set of operands, P_i, used by a memory device, c_i, to contain only these essential operands. Since $P_i \subseteq P$ instead of $P_i = P$, the complexity of each $N_i(\Sigma, V, V'_i)$ as well as the set of unique feasible states are reduced. Furthermore, the quality of the results of the reachable state analysis is maintained since these excluded operands can not effect that data path's ability to create operands.

To identify the set of memory devices which should store an operand, we make the following observations. First, the data flow graph indicates the set of function unit input ports at which an operand can appear. Second, the final state requirements specify the set of memory unit input ports at which an operand can appear. Third, operands must travel to these input ports by a combination of wires, memory device, and multiplexers. Therefore, we define a collection of output ports for each wire, w_i, represented by

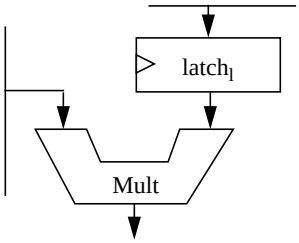


Figure 4. Dedicated latch

$\tau'_i = \bigcup_x \tau_{i,x}$, which can reach w_i without any bound on the number of cycles. Finally, given the list of wires connected to these input ports, we can determine all of the output ports which can route the operand p_k to any of the locations it may need to reach. Since we defined $\tau_{i,x}$ to include the output ports of memory devices, this list of output ports will exclude ports for memory devices whose outputs are not important for the use of operand p_k.

We define $\Omega(p_k)$ as the set of relevant output ports to operand p_k. Each port set is defined by both the final state specification and set of operations which use p_k, as in:

$$\Omega(p_k) = \{ \bigcup_{v_{i,k} \in S_f(V)} \left[\Theta'_i \cup \bigcup_{\phi \in \Phi_i} (\tau'_{W(\phi)}) \right] \} \cup \{ \bigcup_{e_l \mid p_k \in \Pi_l} \left[\tau'_{W(\phi_{e_l p_k})} \right] \}$$

where ϕ_{e,p_k} identifies the input port associated with operand p_k. When considering the requirements placed on the final state, it is important to include the output ports of memory device specified in the final state as well as the ports which can reach this device. From this, a new operand list, $P_i = \bigcup_{p_k \in P'} p_k \mid \Theta_i \subseteq \Omega(p_k)$, is generated for each memory device with which to simplify EQ. 3.

4. Results

A series of tools were developed to demonstrate the feasibility of these techniques. Each tool utilized an in-house BDD package and was run on a 141MHz SPARC Ultra with 416MB of memory.

4.1. Data paths

Our benchmarks utilize five different example data paths each of which exhibits different requirements. Four of these data paths are variations of the high level description of Texas Instruments' TMS32010 DSP processor. While the first design mirrors the TMS32010's data-path portion, the second design incorporates a second global bus to investigate the effect of added connectivity. The third and fourth designs mirror the first two except that a two-cycle pipelined multiplier is substituted for the single cycle multiplier. Each of these four designs were coded utilizing the base component behavior types introduced in Section 2.1 and are depicted as a composite in Fig. 5. The dashed lines represent a second bus which was added for the two-bus examples. The addition of the second bus permitted the A multiplexer and the t latch to be replaced by a single multiplexer, B. For those design with the pipeline multiplier, a latch and second pipeline stage are inserted between the multiplier and multiplexer C.

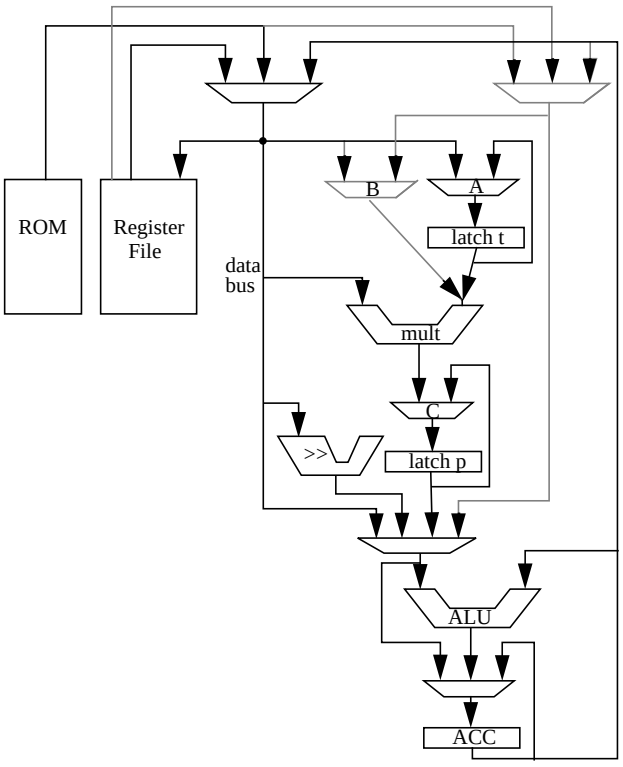


Figure 5. TMS32020 based data-path models

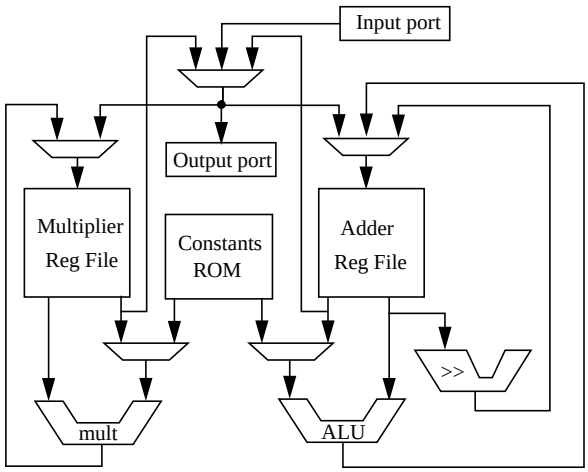


Figure 6. Dual register data path

For the fifth benchmark, we wanted to demonstrate our ability to accommodate dedicated register files in the data path. While such architectures are accommodated by a variety of compiler techniques, there are few such benchmarks in the literature. Therefore, we introduce a "Dual Register" data path which is used as our fifth benchmark and is depicted in Fig. 6.

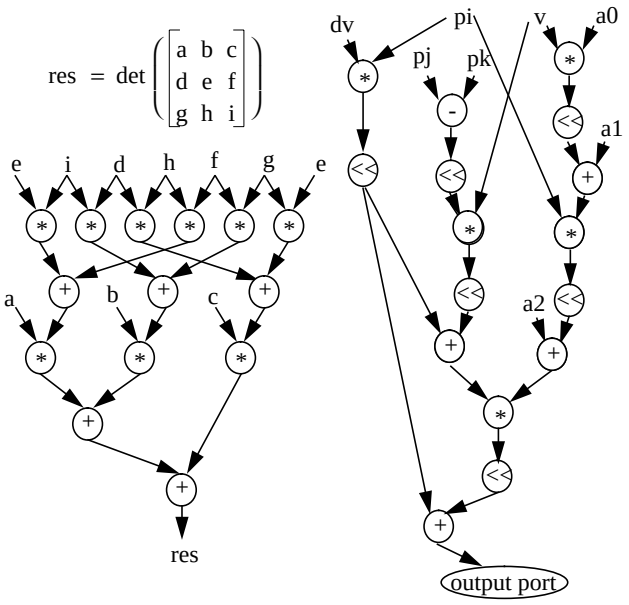
4.2. Data flows

The scheduling benchmarks utilized three DFG's: differential equation(diff_eq), 3x3 determinant (3x3_det), and differential heat release computation (dhrc). The determinant benchmark that we introduce is specified in Fig. 7a. Additionally, the dhrc benchmark contained many operations specific to memory index operations which were inappropriate for our model. Therefore, the modified dhrc benchmark shown in Fig. 7b was utilized. Each of these data flows specify commutativity for each operand pair under the assumption that the ALU supports both forms of subtraction. Finally, each of these DFG's were checked for redundant operations (as defined by EQ. 2) and automatically merged such operands during each of the executions.

4.3. Relative performance

Table 2 lists the run times for the bounded minimum-cycle scheduler. The benchmarks are organized by their DFG, listing the data path, the number of cycles associated with the minimum schedule length, and the execution times. The quality of the scheduling results are not modified by the proposed techniques since no heuristic pruning is involved, but the complexity of the reachable state analysis and the resulting run times are very dependent upon the pruning techniques employed.

The first column, "neither" lists the run times resulting from executing the reachable state analysis utilizing every previously published pruning techniques. The "color" column lists the run times when memory coloring is utilized. Substantial benefits are visible in data paths which contained memory devices dedicated to a function unit input such as the "t latch" in the single bus



a) 3x3 determinate b) modified dhrc

Figure 7. Example data-flow graphs.

tms32010 designs. The benefit for each of these single bus benchmarks is relatively uniform for each DFG. This result is expected since the reduction to the state space is dependent upon theDFG's operation set. The occasional increase in execution times associated with the other benchmarks reflects the overhead due to the computation of the τ sets.

The results for our proposed ALAP bounds are compared against two sets of run times. In addition, to the run times listed under "neither", a set of run times are listed corresponding to bounding the reachable state analysis with ALAP derived solely from the function unit resources, "FALAP". The results from using this traditional bound are mixed. Examples containing the dual-register file data path or the diff_eq DFG show only slight improvements in run times, if any. By contrast, the ALAP bounds derived from the complete set of data path resources, "DALAP", demonstrate a consistent set of improvements.

Fig. 8 demonstrates how these benefits are realized for a particular example. Here we see the constant growth in the set of

TABLE 2. Exact scheduling results

Data Flow	Data Path			# Cyc	Run Time (sec)				
					neither	color	FALAP	DALAP	both
diff eq	tms 32010	1-cyc mult.	1 bus	17	250	125	240	195	97
			2 bus	12	23	23	23	19	19
		pipe mult	1 bus	17	565	350	539	301	191
			2 bus	13	109	109	97	51	37
	dual register file			12	15	15	16	16	16
3x3 det	tms 32010	1-cyc mult.	1 bus	20	4,744	1,486	3,627	2,098	686
			2 bus	13	266	278	188	92	94
		pipe mult	1 bus	20	11,214	5,412	8,109	2,625	1,395
			2 bus	13	798	812	474	77	78
	dual register file			22	414	419	411	382	398
dhrc	tms 32010	1-cyc mult.	1 bus	22	2,507	486	2,258	790	168
			2 bus	19	1,050	1,052	664	333	277
		pipe mult	1 bus	23	16,045	2,353	13,242	1,180	273
			2 bus	21	1,533	1,560	1,031	325	326
	dual register file			19	102	105	106	37	37

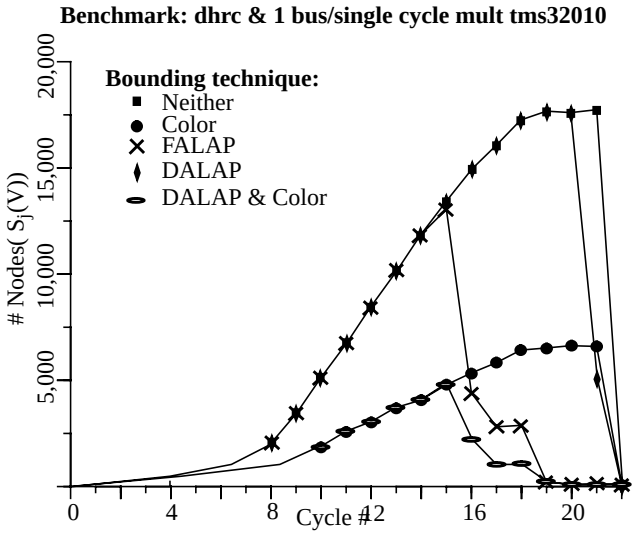


Figure 8. Cycle by cycle performance comparison.

reachable states, until the set is intersected with the set of final states. By employing the ALAP bounds, the size of the reachable state set is reduced as the analysis approaches the anticipated final clock cycle as elements are removed which have no impact on the solution set. Finally, when these techniques are combined with the memory coloring (corresponding run times are in listed in column "both") the size over all cycles is limited by reducing the set of states from which the reachable state analysis must consider.

5. Conclusion

This paper has shown that ALAP bounds can be posed for the pre-existing data path scheduling problem and that they are a powerful means of reducing the scheduling complexity. The primary constraint of a data path is its ability to route operands between various required points. This constraint was utilized to pose superior ALAP bounds. Additionally, the analysis of the movement freedom proves useful in pruning memory bindings which are inappropriate for specific operands. Although the operands may be stored at these locations, their presence can not effect the solution set and only complicates the state representation. These bounds have possible application in generalizations of this problem such as floorplanning constraint scheduling.

6. References

[1] C. N. Coelho Jr, G. De Micheli, "Dynamic Scheduling and Synchronization Synthesis of Concurrent Digital Systems under System-Level Constraints", *Proc. IEEE Int. Conf. Computer-Aided Design*, 1994

[2] R Leupers, P. Marwedel, "Retargetable Generation of Code Selectors from HDL Processor Models" in *Proceedings of European Design & Test Conference (ED & TC)*, Paris/France, 1997

[3] P. Marwedel, G. Goosens (eds.), *Code Generation for Embedded Processors*, Kluwer Academic Publishers, 1995.

[4] T. Miyazaki and M. Ikeda, "High Level Synthesis Using Given Datapath Information", *IEECE Trans. Fundamentals*, Oct 1993

[5] C. Monahan, F. Brewer, "Symbolic Modeling and Evaluation of Data Paths", *32nd Design Automation Conference Proceedings*, San Francisco, 1995.Conference Proceedings, San Francisco, 1995.

[6] C. Monahan, *Symbolic Data Path Modeling*, Ph.D. Thesis University of California, Santa Barbara, 1997

[7] A. Timmer, *From Design Space Exploration to Code Generati on*, Ph.D. Thesis Eindhoven University of Technology, 1996

[8] J. C.-Y. Yang, G. De Micheli, and M. Damiani, "Scheduling and Control Generation with Environmental Constraints based on Automata Representations", *IEEE Trans. CAD/ICAS*, Feb. 1996.