# 6

# EXTENSIONS

The techniques of the previous chapters apply compile time analysis to static schedules for HSDF graphs that have no decision making at the dataflow graph level. In this chapter we consider graphs with data dependent control flow. Recall that atomic actors in an HSDF graph are allowed to perform data-dependent decision making within their body, as long as their input/output behaviour respects SDF semantics. We show how some of the ideas we explored previously can still be applied to dataflow graphs containing actors that display data-dependent firing patterns, and therefore are not SDF actors.

## 6.1 The Boolean Dataflow model

The **Boolean Dataflow (BDF)** model was proposed by Lee [Lee91] and was further developed by Buck [Buck93] for extending the SDF model to allow data-dependent control actors in the dataflow graph. BDF actors are allowed to contain a *control* input, and the number of tokens consumed and produced on the arcs of a BDF actors can be a two-valued function of a token consumed at the control input. Actors that follow SDF semantics, i.e. that consume and produce fixed number of tokens on their arcs, are clearly a subset of the set of allowed BDF actors (SDF actors simply do not have any control inputs). Two basic dynamic actors in the token flow model are the SWITCH and SELECT actors shown in Fig. 6.1. The switch actor consumes one Boolean-valued control token and another input token; if the control token is TRUE, the input token is copied to the output labelled T, otherwise it is copied to the output labelled F. The SELECT actor performs the complementary operation; it reads an input token from its T input if the control token is TRUE, otherwise it reads from its F input; in either case, it copies the token to its output. Constructs such as conditionals and data-dependent itera-

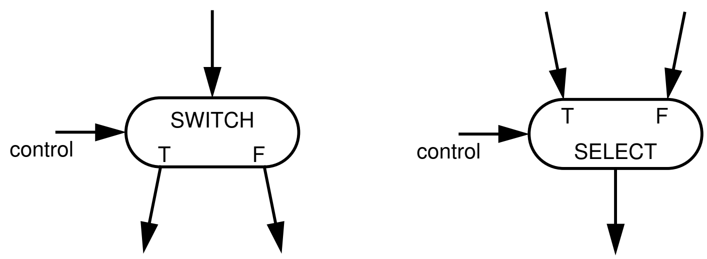



Figure 6.1. BDF actors SWITCH and SELECT

tions can easily be represented in a BDF graph, as illustrated in Fig. 6.2. The vertices A, B, C, etc. in Fig. 6.2 need not be atomic actors; they could also be arbitrary SDF graphs. A BDF graph allows SWITCH and SELECT actors to be connected in arbitrary topologies. Buck [Buck93] in fact shows that any Turing machine can be expressed as a BDF graph, and therefore the problems of determining whether such a graph deadlocks and whether it uses bounded memory are undecidable. Buck proposes heuristic solutions to these problems based on extensions of the techniques for SDF graphs to BDF model.

### 6.1.1 Scheduling

Buck presents techniques for statically scheduling BDF graphs on a single processor; his methods attempt to generate a sequential program without a dynamic scheduling mechanism, using `if-then-else` and `do-while` control constructs where required. Because of the inherent undecidability of determining deadlock behaviour and bounded memory usage, these techniques are not always

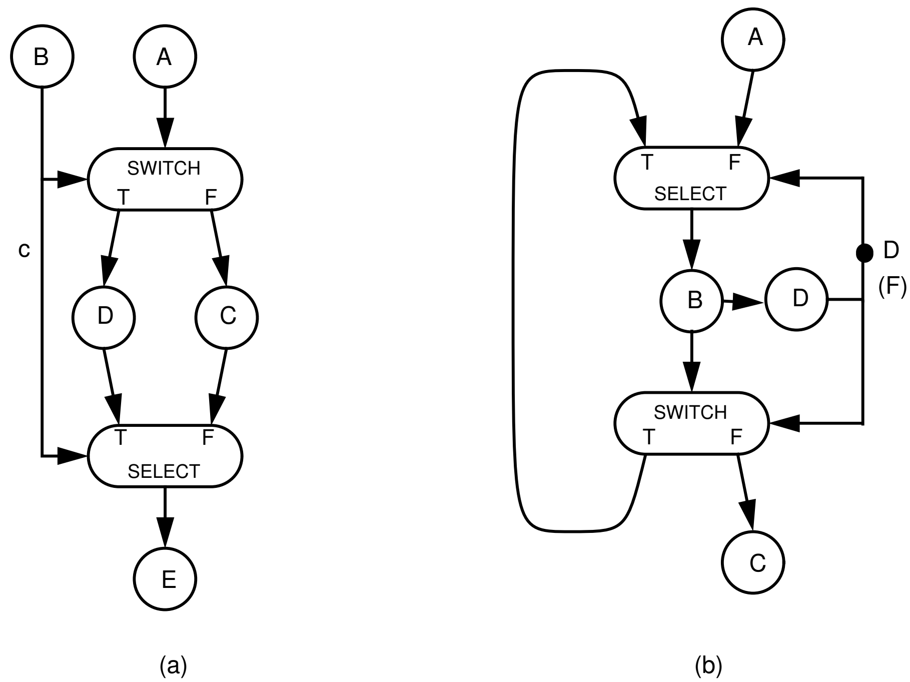



Figure 6.2. (a) Conditional (if-then-else) dataflow graph. The branch outcome is determined at run time by actor B. (b) Graph representing data-dependent iteration. The termination condition for the loop is determined by actor D.

guaranteed to generate a static schedule, even if one exists; a dynamically scheduled implementation, where a run time kernel decides which actors to fire, can be used when a static schedule cannot be found in a reasonable amount of time.

Automatic parallel scheduling of general BDF graphs is still an unsolved problem. A *naive* mechanism for scheduling graphs that contain SWITCH and SELECT actors is to generate an Acyclic Precedence Graph (APG), similar to the APG generated for SDF graphs discussed in section 1.2.1, for every possible assignment of the Boolean valued control tokens in the BDF graph. For example, the if-then-else graph in Fig. 6.2(a) could have two different APGs, shown in Fig.

6.3, and APGs thus obtained can be scheduled individually using a self-timed

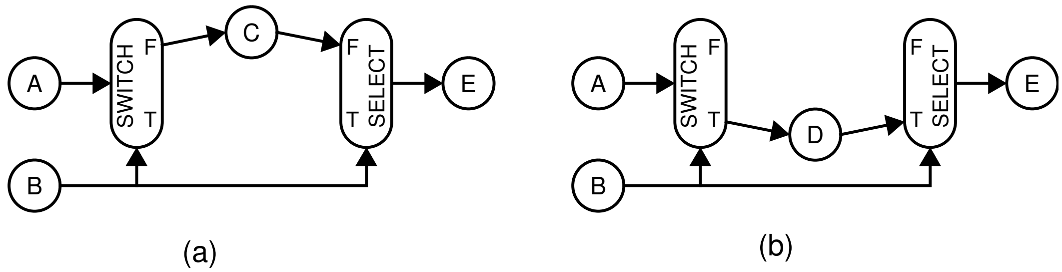



Figure 6.3. Acyclic precedence graphs corresponding to the if-then-else graph of Fig. 6.2. (a) corresponds to the TRUE assignment of the control token, (b) to the FALSE assignment.

strategy; each processor now gets several lists of actors, one list for each possible assignment of the control tokens. The problem with this approach is that for a graph with $n$ different control tokens, there are $2^n$ possible distinct APGs, each corresponding to each execution path in the graph. Such a set of APGs can be compactly represented using the so called Annotated Acyclic Precedence Graph (AAPG) of [Buck93] in which actors and arcs are annotated with conditions under which they exist in the graph. Buck uses the AAPG construct to determine whether a bounded-length uniprocessor schedule exists. In the case of multiprocessor scheduling, it is not clear how such an AAPG could be used to explore scheduling options for the different values that the control tokens could take, without explicitly enumerating all possible execution paths.

The main work in parallel scheduling of dataflow graphs that have dynamic actors has been the **Quasi-static scheduling** approach, first proposed by Lee [Lee88b] and extended by Ha [Ha92]. In this work, techniques have been developed that statically schedule standard dynamic constructs such as data-dependent conditionals, data-dependent iterations, and recursion. These constructs must be identified in a given dataflow graph, either manually or automatically, before Ha's techniques can be applied. These techniques make the simplifying assumption that the control tokens for different dynamic actors are independent of one another, and that each control stream consists of tokens that take TRUE or FALSE values randomly and are independent and identically distributed (i.i.d.) according to statistics known at compile time. Such a quasi-static scheduling approach clearly does not handle a general BDF graph, although it is a good starting point for doing so.

Ha's quasi-static approach constructs a blocked schedule for one iteration of the dataflow graph. The dynamic constructs are scheduled in a hierarchical fashion; each dynamic construct is scheduled on a certain number of processors, and is then converted into a single node in the graph and is assigned a certain *execution profile*. A profile of a dynamic construct consists of the number of processors assigned to it, and the schedule of that construct on the assigned processors; the profile essentially defines the shape that a dynamic actor takes in the processor-time plane. When scheduling the remainder of the graph, the dynamic construct is treated as an atomic block, and its profile is used to determine how to schedule the remaining actors around it; the profile helps tiling actors in the processor-time plane with the objective of minimizing the overall schedule length. Such a hierarchical scheme effectively handles nested control constructs, e.g. nested conditionals.

One important aspect of quasi-static scheduling is determining execution profiles of dynamic constructs. Ha [Ha92] studies this problem in detail and shows how one can determine optimal profiles for constructs such as conditionals, data-dependent iteration constructs, and recursion, assuming certain statistics are known about the run time behaviour of these constructs.

We will consider only the conditional and the iteration construct here. We will assume that we are given a quasi-static schedule, obtained either manually or using Ha's techniques. We then explore how the techniques proposed in the previous chapters for multiprocessors that utilize a self-timed scheduling strategy apply when we implement a quasi-static schedule on a multiprocessor. First we propose an implementation of a quasi-static schedule on a shared memory multiprocessor, and then we show how we can implement the same program on the OMA architecture, using the hardware support provided in the OMA prototype for such an implementation

## 6.2 Parallel implementation on shared memory machines

### 6.2.1 General strategy

A quasi-static schedule ensures that the pattern of processor availability is identical regardless of how the data-dependent construct executes at runtime; in the case of the conditional construct this means that irrespective of which branch is actually taken, the pattern of processor availability after the construct completes execution is the same. This has to be ensured by inserting idle time on processors when necessary. Fig. 6.4 shows a quasi-static schedule for a conditional construct. Maintaining the same pattern of processor availability allows static scheduling to proceed after the execution of the conditional; the data-dependent nature of the control construct can be ignored at that point. In Fig. 6.4 for example, the schedul-

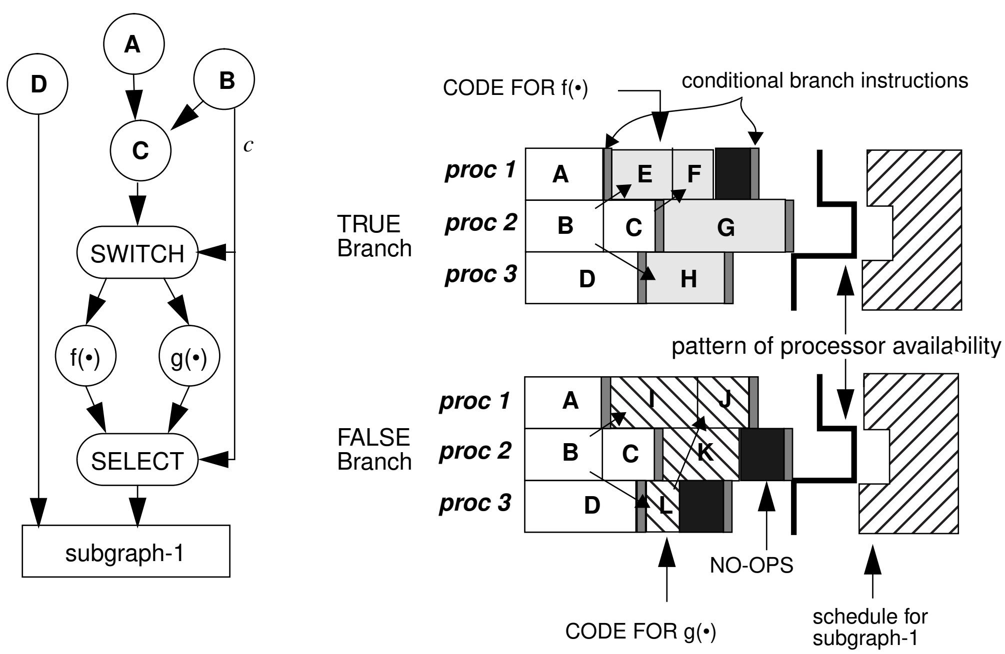



Figure 6.4. Quasi-static schedule for a conditional construct (adapted from [Lee88b])

ing of subgraph-1 can proceed independent of the conditional construct because the pattern of processor availability after this construct is the same independent of the branch outcome; note that “nops” (idle processor cycles) have been inserted to ensure this.

Multiprocessor implementation of a quasi-static schedule directly, however, implies enforcing global synchronization after each dynamic construct in order to ensure a particular pattern of processor availability. We therefore use a mechanism similar to the self-timed strategy; we first determine a quasi-static schedule using the methods of Lee and Ha, and then discard the timing information and the restrictions of maintaining a processor availability profile. Instead, we only retain the assignment of actors to processors, the order in which they execute, and also under what conditions on the Boolean tokens in the system the actor should execute. Synchronization between processors is done at run time whenever processors communicate. This scheme is analogous to constructing a self-timed schedule from a fully-static schedule, as discussed in section 1.2.2. Thus the quasi-static schedule of Fig. 6.4 can be implemented by the set of programs in Fig. 6.5, for the three processors. Here, $\{r_{c1}, r_{c2}, r_1, r_2\}$ are the receive actors, and

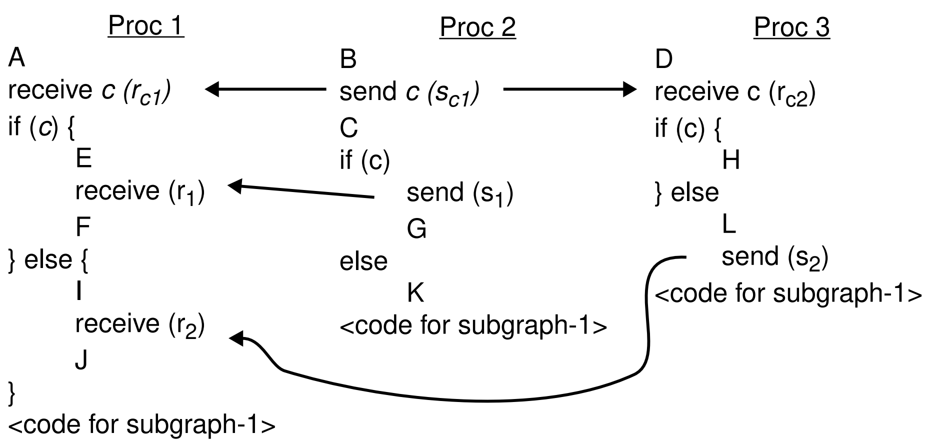



Figure 6.5. Programs on three processors for the quasi-static schedule of Fig. 6.4

$\{s_{c1}, s_1, s_2\}$ are the send actors. The subscript “$c$” refers to actors that communicate control tokens.

The main difference between such an implementation and the self-timed implementation we discussed in earlier chapters are the control tokens. Whenever a conditional construct is partitioned across more than one processor, the control token(s) that determine its behaviour must be broadcast to all the processors that execute that construct. Thus in Fig. 6.4 the value $c$, which is computed by Processor 2 (since the actor that produces $c$ is assigned to Processor 2), must be broadcast to the other two processors. In a shared memory machine this broadcast can be implemented by allowing the processor that evaluates the control token (Processor 2 in our example) to write its value to a particular shared memory location preassigned at compile time; the processor will then update this location once for each iteration of the graph. Processors that require the value of a particular control token simply read that value from shared memory, and the processor that writes the value of the control token needs to do so only once. In this way actor executions can be conditioned upon the value of control tokens evaluated at run time. In the previous chapters we discussed synchronization associated with data transfer between processors. Synchronization checks must also be performed for the control tokens; the processor that writes the value of a token must not overwrite the shared memory location unless all processors requiring the value of that token have in fact read the shared memory location, and processors reading a control token must ascertain that the value they read corresponds to the current iteration rather than a previous iteration.

The need for broadcast of control tokens creates additional communication overhead that should ideally be taken into account during scheduling. The methods of Lee and Ha, and also prior research related to quasi-static scheduling that they refer to in their work, do not take this cost into account. Static multiprocessor scheduling applied to graphs with dynamic constructs taking costs of distributing control tokens into account is thus an interesting problem for further study.

## 6.2.2 Implementation on the OMA

Recall that the OMA architecture imposes an order in which shared memory is accessed by processors in the machine. This is done to implement the OT strategy, and is feasible because the pattern of processor communications in a self-timed schedule of an HSDF graph is in fact predictable. What happens when we want to run a program derived from a quasi-static schedule, such as the parallel program in Fig. 6.5, which was derived from the schedule in Fig. 6.4? Clearly, the order of processor accesses to shared memory is no longer predictable; it depends on the outcome of run time evaluation of the control token $c$. The quasi-static schedule of Fig. 6.4 specifies the schedules for the TRUE and FALSE branches of the conditional. If the value of $c$ were always TRUE, then we can determine from the quasi-static schedule that the transaction order would be $(s_{c1}, r_{c1}, r_{c2}, s_1, r_1, \langle\text{access order for subgraph-1}\rangle)$, and if the value of $c$ were always FALSE, the transaction order would be $(s_{c1}, r_{c1}, r_{c2}, s_2, r_2, \langle\text{access order for subgraph-1}\rangle)$. Note that writing the control token $c$ once to shared memory is enough since the same shared location can

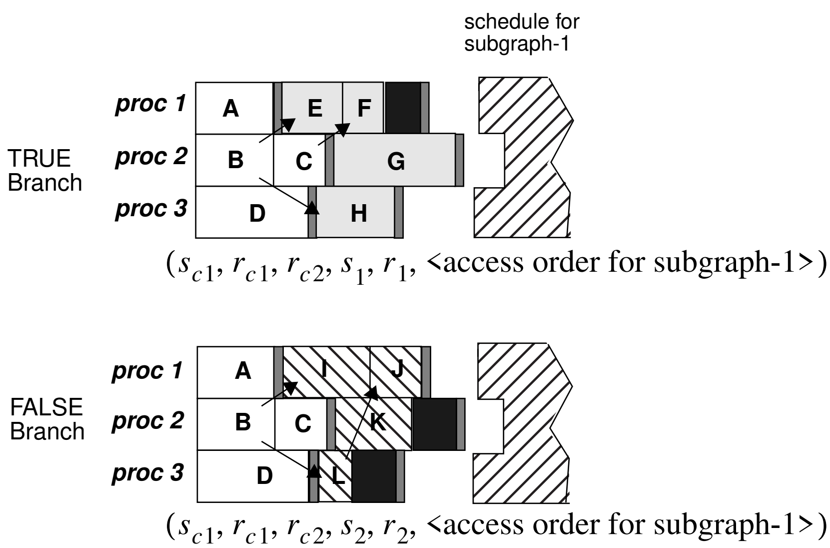



Figure 6.6. Transaction order corresponding to the TRUE and FALSE branches

be read by all processors requiring the value of $c$.

For the OMA architecture, our proposed strategy is to switch between these two access orders at run time. This is enabled by the preset feature of the transaction controller (Chapter 3, section 3.4.2). Recall that the transaction controller is implemented as a presettable schedule counter that addresses memory containing the processor IDs corresponding to the bus access order. To handle conditional constructs, we derive two bus access lists corresponding to each path in the program, and the processor that determines the branch condition (processor 2 in our example) forces the controller to switch between the access lists by loading the schedule counter with the appropriate value (address "7" in the bus access schedule of Fig. 6.7). Note from Fig. 6.7 that there are two points where the schedule counter can be set; one is at the completion of the TRUE branch, and the other is a jump into the FALSE branch. The branch into the FALSE path is best taken care of by processor 2, since it computes the value of the control token $c$, whereas the branch after the TRUE path (which bypasses the access list of the FALSE branch) is best taken care of by processor 1, since processor 1 already possesses the bus at the time when the counter needs to be loaded. The schedule counter load operations are easily incorporated into the sequential programs of processors 1 and 2.

The mechanism of switching between bus access orders works well when the number of control tokens is small. But if the number of such tokens is large, then this mechanisms breaks down, even if we can efficiently compute a quasi-static schedule for the graph. To see why this is so, consider the graph in Fig. 6.8, which contains $k$ conditional constructs in parallel paths going from the input to the output. The functions "$f_i$" and "$g_i$" are assumed to be subgraphs that are assigned to more than one processor. In Ha's hierarchical scheduling approach, each conditional is scheduled independently; once scheduled, it is converted into an atomic node in the hierarchy, and a profile is assigned to it. Scheduling of the other conditional constructs can then proceed based on these profiles. Thus the scheduling complexity in terms of the number of parallel paths is $O(k)$ if there are $k$ parallel paths. If we implement the resulting quasi-static schedule in the

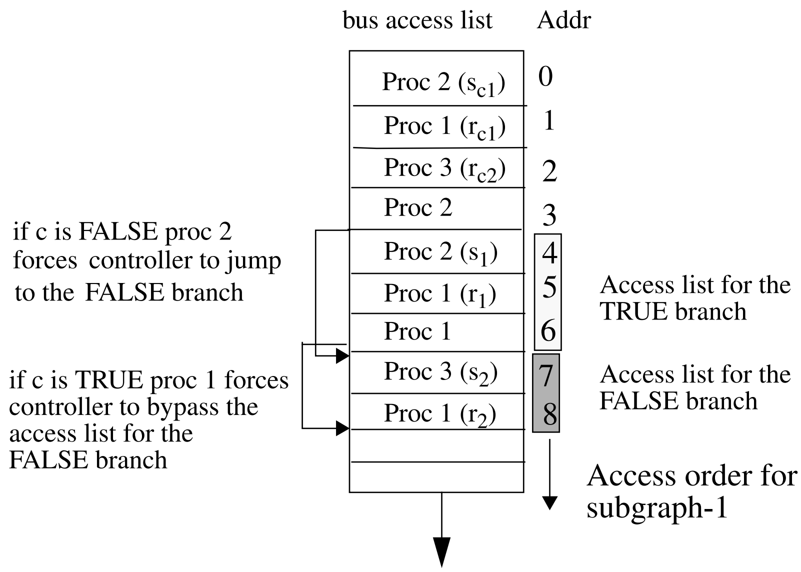



Figure 6.7. Bus access list that is stored in the schedule RAM for the quasi-static schedule of Fig. 6.6. Loading operation of the schedule counter conditioned on value of $c$ is also shown.

manner stated in the previous section, and employ the OMA mechanism above, we would need one bus access list for every combination of the Booleans $b_1,..., b_k$. This is because each $\mathsf{f}_i$ and $\mathsf{g}_i$ will have its own associated bus access list, which then has to be combined with the bus access lists of all the other branches to yield one list. For example, if all Booleans $b_i$ are true, then all the $\mathsf{f}_i$'s are executed, and we get one access list. If $b_1$ is TRUE, and $b_2$ through $b_k$ are FALSE, then $\mathsf{g}_1$ is executed, and $\mathsf{f}_2$ through $\mathsf{f}_k$ are executed. This corresponds to another bus access list. This implies $2^k$ bus access lists for each of the combination of $\mathsf{f}_i$ and $\mathsf{g}_i$ that execute, i.e. for each possible execution path in the graph.

### 6.2.3 Improved mechanism

Although the idea of maintaining separate bus access lists is a simple mechanism for handling control constructs, it can sometimes be impractical, as in the example above. We propose an alternative mechanism based on *masking* that handles parallel conditional constructs more effectively.

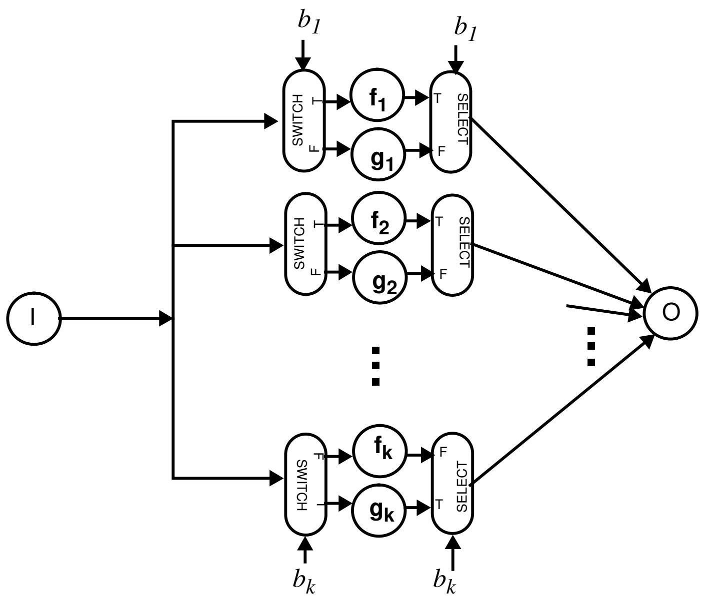



Figure 6.8. Conditional constructs in parallel paths

The main idea behind masking is to store an ID of a Boolean variable along with the processor ID in the bus access list. The Boolean ID determines whether a particular bus grant is “enabled.” This allows us to combine the access lists of all the nodes $f_1$ through $f_k$ and $g_1$ through $g_k$. The bus grant corresponding to each $f_i$ is tagged with the boolean ID of the corresponding $b_i$, and an additional bit indicates that the bus grant is to be enabled when $b_i$ is TRUE. Similarly, each bus grant corresponding to the access list of $g_i$ is tagged with the ID of $b_i$, and an additional bit indicates that the bus grant must be enabled only if the corresponding control token has a FALSE value. At runtime, the controller steps through the bus access list as before, but instead of simply granting the bus to the processor at the head of the list, it first checks that the control token corresponding to the Boolean ID field of the list is in its correct state. If it is in the correct state (i.e. it is TRUE for a bus grant corresponding to an $f_i$ and FALSE for a bus grant corresponding to a $g_i$), then the bus grant is performed, otherwise it is masked. Thus the run time values of the Booleans must be made available to the transaction controller for it to decide whether to mask a particular bus grant or not.

More generally, a particular bus grant should be enabled by a product

(AND) function of the Boolean variables in the dataflow graph, and the complement of these Booleans. Nested conditionals in parallel branches of the graph necessitate bus grants that are enabled by a product function; a similar need arises when bus grants must be reordered based on values of the Boolean variables. Thus, in general we need to implement an *annotated* bus access list of the form $\{(c_1)ProcID_1, (c_2)ProcID_2, \ldots\}$ ; each bus access is annotated with a Boolean valued condition $c_i$, indicating that the bus should be granted to the processor corresponding to $ProcID_i$ when $c_i$ evaluates to TRUE; $c_i$ could be an arbitrary product function of the Booleans $(b_1, b_2, \ldots, b_n)$ in the system, and the complements of these Booleans (e.g. $c_j = b_2 \cdot \overline{b_4}$, where the bar over a variable indicates its complement).

This scheme is implemented as shown in Fig. 6.9. The schedule memory

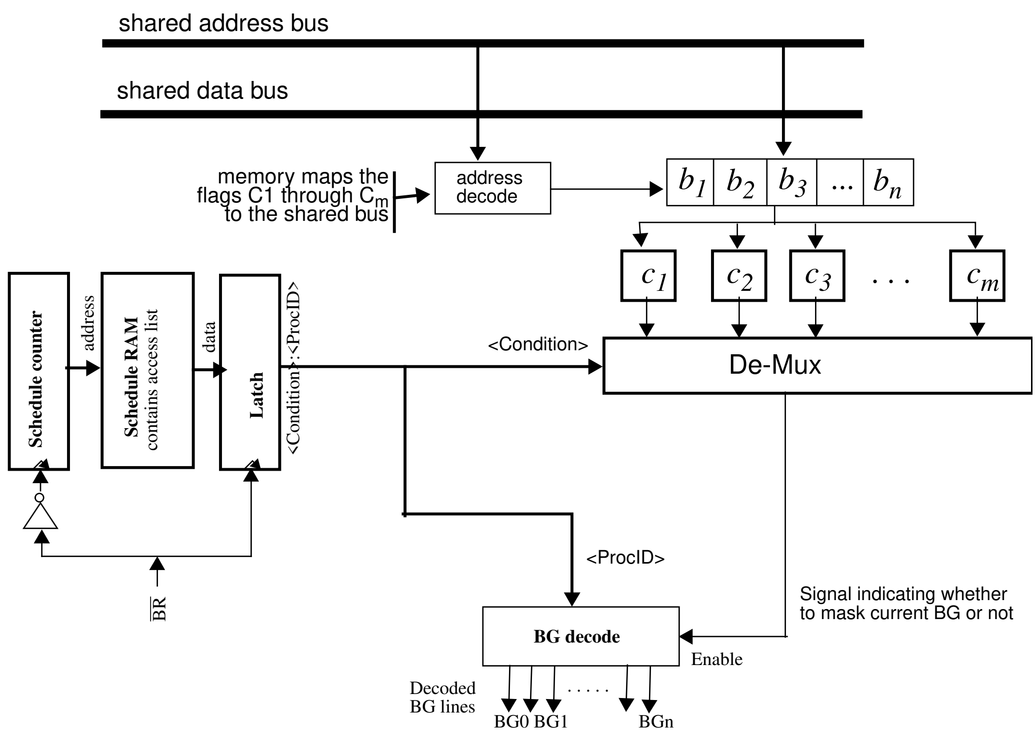



Figure 6.9. A bus access mechanism that selectively "masks" bus grants based on values of control tokens that are evaluated at run time

now contains two fields corresponding to each bus access: <Condition>:<ProcID>

instead of the <ProcID> field alone that we had before. The <Condition> field encodes a unique product $c_i$ associated with that particular bus access. In the OMA prototype, we can use 3 bits for <ProcID>, and 5 bits for the <Condition> field. This would allow us to handle 8 processors and 32 product combinations of Booleans. There can be up to $m = 3^n$ product terms in the worst case corresponding to $n$ Booleans in the system, because each Boolean $b_i$ could appear in the product term as itself, or its complement, or not at all (corresponding to a "don't care"). It is unlikely that all $3^n$ possible product terms will be required in practice; we therefore expect such a scheme to be practical. The necessary product terms ($c_j$) can be implemented within the controller at compile time, based on the bus access pattern of the particular dynamic dataflow graph to be executed.

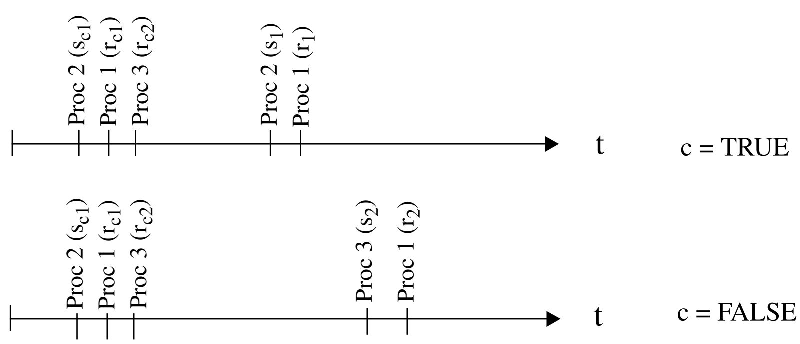
In Fig. 6.9, the flags $b_1, b_2, \ldots, b_n$, are 1-bit memory elements (flip-flops) that are memory mapped to the shared bus, and store the values of the Boolean control tokens in the system. The processor that computes the value of each control token updates the corresponding $b_i$ by writing to the shared memory location that maps to $b_i$. The product combinations $c_1, c_2, \ldots, c_n$, are just AND functions of the $b_i$s and the complement of the $b_i$s, e.g. $c_j$ could be $b_2 \cdot \overline{b_4}$. As the schedule counter steps through the bus access list, the bus grant is actually granted only if the condition corresponding to that access evaluates to TRUE; thus if the entry <$c_2$><Proc1> appears at the head of the bus access list, and $c_2 = b_2 \cdot \overline{b_4}$, then processor 1 receives a bus grant only if the control token $b_2$ is TRUE and $b_4$ is FALSE, otherwise the bus grant is masked and the schedule counter moves up to the next entry in the list.

This scheme can be incorporated into the transaction controller in our existing OMA architecture prototype, since the controller is implemented on an FPGA. The product terms $c_1, c_2, \ldots, c_n$ may be programmed into the FPGA at compile time; when we generate programs for the processors, we can also generate the *annotated* bus access list (a sequence of <Condition><Proc ID> entries), and a hardware description for the FPGA (in VHDL, say) that implements the required product terms.

### 6.2.4 Generating the annotated bus access list

Consider the problem of obtaining an annotated bus access list $\{(c_1)ProcID_1, (c_2)ProcID_2, \ldots\}$, from which we can derive the sequence of <Condition><Proc ID> entries for the mask-based transaction controller. A straightforward, even if inefficient, mechanism for obtaining such a list is to use enumeration; we simply enumerate all possible combinations of Booleans in the system ($2^n$ combinations for $n$ Booleans), and determine the bus access sequence (sequence of ProcID's) for each combination. Each combination corresponds to an execution path in the graph, and we can estimate the time of occurrence of bus accesses corresponding to each combination from the quasi-static schedule. For example, bus accesses corresponding to one schedule period of the two execution paths in the quasi-static schedule of Fig. 6.6 may be marked along the time axis as shown in Fig. 6.10 (we have ignored the bus access sequence corresponding to subgraph-1 to keep the illustration simple).

The bus access schedules for each of the combinations can now be collapsed into one annotated list, as in Fig. 6.10; the fact that accesses for each combination are ordered with respect to time allows us to enforce a global order on the




*{(c) Proc2, ($\bar{c}$) Proc2, (c) Proc1, ($\bar{c}$) Proc1, (c) Proc3, ($\bar{c}$) Proc3, (c) Proc2, (c) Proc1, ($\bar{c}$) Proc3, ($\bar{c}$) Proc 1}* "annotated" list

Figure 6.10. Bus access lists and the annotated list corresponding to Fig. 6.6

accesses in the collapsed bus access list. The bus accesses in the collapsed list are annotated with their respective Boolean condition.

The collapsed list obtained above can be used as is in the masked controller scheme; however there is a potential for optimizing this list. Note, however, that the same transaction may appear in the access list corresponding to different Boolean combinations, because a particular Boolean token may be a "don't care" for that bus access. For example, the first three bus accesses in Fig. 6.10 appear in both execution paths, because they are independent of the value of $c$. In the worst case a bus access that is independent of all Booleans will end up appearing in the bus access lists of all the Boolean combinations. If these bus accesses appear contiguously in the collapsed bus access sequence, we can combine them into one. For example, "($c$) Proc2, ($\bar{c}$) Proc2" in the annotated schedule of Fig. 6.10 can be combined into a single "Proc 2" entry, which is not conditioned on any control token. Consider another example: if we get contiguous entries "($b_1 \cdot \overline{b_2}$) Proc3" and "($b_1 \cdot b_2$) Proc3" in the collapsed list, we can replace the two entries with a single entry "($b_1$) Proc3".

More generally, if the collapsed list contains a contiguous segment of the form:

$$\{\ldots, (c_1)ProcID_k, (c_2)ProcID_k, (c_3)ProcID_k, \ldots, (c_l)ProcID_k, \ldots\} \quad ,$$

we can write each of the contiguous segments as:

$$\{\ldots, (c_1 + c_2 + \ldots + c_l)ProcID_k, \ldots\} \ ,$$

where the bus grant condition is an expression $(c_1 + c_2 + \ldots + c_l)$, which is a sum of products (SOP) function of the Booleans in the system. We can now apply 2-level minimization to determine a minimal representation of each of these expressions. Such 2-level minimization can be done by using a logic minimization tool such as ESPRESSO [Bray84], which simplifies a given SOP expression into an SOP representation with minimal number of product terms. Suppose the expression $(c_1 + c_2 + \ldots + c_l)$ can be minimized into another SOP expression

$(c_1' + c_2' + \ldots + c_p')$ , where $p < l$. We can then replace the segment

$$\{\ldots, (c_1)ProcID_k, (c_2)ProcID_k, (c_3)ProcID_k, \ldots, (c_l)ProcID_k, \ldots\}$$

of the annotated bus access list with an equivalent segment of the form:

$$\{\ldots, (c_1')ProcID_k, (c_2')ProcID_k, (c_3')ProcID_k, \ldots, (c_p')ProcID_k, \ldots\} \quad .$$

We can thus obtain a minimal set of contiguous appearances of a bus grant to the same processor.

Another optimization that can be performed is to combine annotated bus access lists with the switching mechanism of section 6.2.1. Suppose we have the following annotated bus access list:

$$\{\ldots, (b_1 \cdot \overline{b_2})ProcID_i, (b_1 \cdot \overline{b_3})ProcID_j, (b_1 \cdot b_4 \cdot b_5)ProcID_k, \ldots\} \quad .$$

Then, by "factoring" $b_1$ out, we can equivalently write the above list as:

$$\{\ldots, (b_1)\{(\overline{b_2})ProcID_i, (\overline{b_3})ProcID_j, (b_4 \cdot b_5)ProcID_k\}, \ldots\} \quad .$$

Now, we can skip over all the three accesses whenever the Boolean $b_1$ is FALSE by loading the schedule counter and forcing it to increment its count by three, instead of evaluating each access separately, and skipping over each one individually. This strategy reduces overhead, because it costs an extra bus cycle to disable a bus access when a condition corresponding to that bus access evaluates to FALSE; by skipping over three bus accesses that we know are going to be disabled, we save three idle bus cycles. There is an added cost of one cycle for loading the schedule counter; the total savings in this example is therefore two bus cycles.

One of the problems with the above approach is that it involves explicit enumeration of all possible combinations of Booleans, the complexity of which limits the size of problems that can be tackled with this approach. An implicit mechanism for representing all possible execution paths is therefore desirable. One such mechanism is the use of Binary Decision Diagrams (BDDs), which have been used to efficiently represent and manipulate Boolean functions for the purpose of logic minimization [Bryant86]. BDDs have been used to compactly represent large state spaces, and to perform operations implicitly over such state spaces when methods based on explicit techniques are infeasible. One difficulty we encountered in applying BDDs to our problem of representing execution paths is that it is not obvious how precedence and ordering constraints can be encoded in a BDD representation. The execution paths corresponding to the various Boolean combinations can be represented using a BDD, but it isn't clear how to represent the relative order between bus accesses corresponding to the different execution paths. We leave this as an area for future exploration.

## 6.3 Data-dependent iteration

A data-dependent iteration construct is shown in Fig. 6.2(b). A quasi-static schedule for such a construct may look like the one in Fig. 6.11. We are assuming

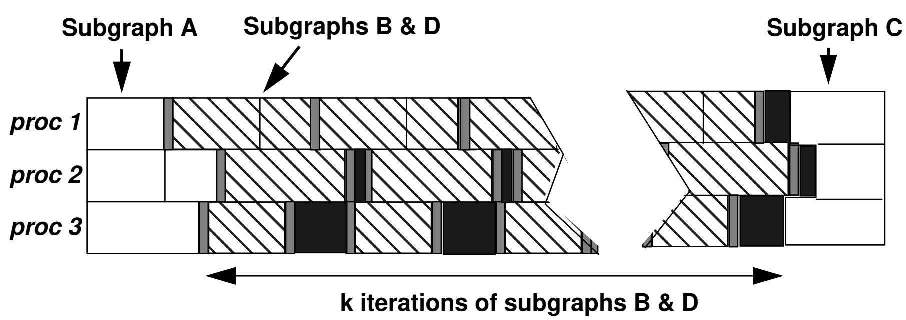



Figure 6.11. Quasi-static schedule for the data-dependent iteration graph of Fig. 6.2(b).

that A, B, C, and D of Fig. 6.2(b) are subgraphs rather than atomic actors.

Such a quasi-static schedule can also be implemented in a straightforward fashion on the OMA architecture, provided that the data-dependent construct spans all the processors in the system. The bus access schedule corresponding to the iterated subgraph is simply repeated until the iteration construct terminates. The processor responsible for determining when the iteration terminates can be made to force the schedule counter to loop back until the termination condition is reached.

This is shown in Fig. 6.12.

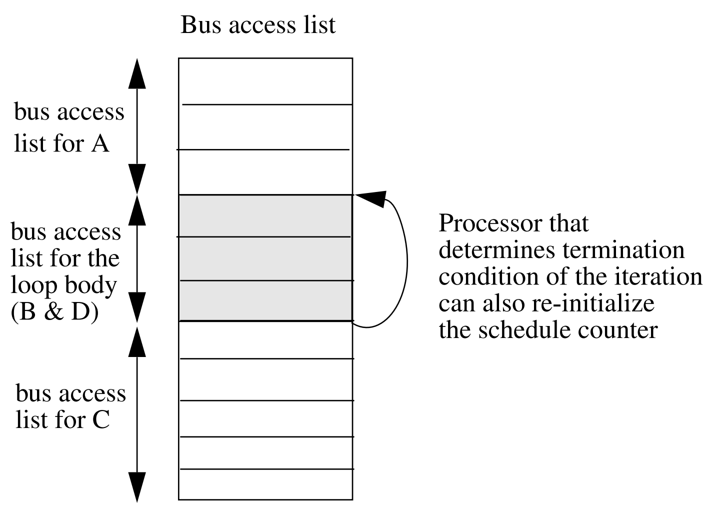



Figure 6.12. A possible access order list corresponding to the quasi-static schedule of Fig. 6.11.

## 6.4 Summary

This chapter dealt with extensions of the ordered transactions approach to graphs with data-dependent control flow. We briefly reviewed the Boolean Dataflow model, and the quasi-static approach to scheduling conditional and data-dependent iteration constructs. We then presented a scheme whereby the Ordered Memory Access board could be used when such control constructs are included in the dataflow graph. In this scheme, bus access schedules are computed for each set of values that the control tokens in the graph evaluate to, and the bus access controller is made to select between these lists at run time based on which set of values the control tokens actually take at any given time. This was also shown to be applicable to data-dependent iteration constructs. Such a scheme is feasible when the number of execution paths in the graph is small. We proposed another mechanism based on masking of bus accesses depending on run time values of control tokens, for handling the case when there are multiple conditional constructs in "parallel."