# Scheduling for ReMove and other partially connected architectures

Steven Roos

Internal Report
1-68340-44(2001)-05

Laboratory of Computer Engineering (CE)
Faculty of Information Technology and Systems
Delft University of Technology
P.O. Box 5031, NL-2600 GA Delft,
The Netherlands

July 2, 2001

# Scheduling for ReMove and other partially connected architectures


## Abstract

From a hardware point of view it is beneficial to design microprocessors with as few communication resources as possible. Unfortunately it is hard for compilers to generate code for machines in which the communication network is not fully connected. The three main problems are the selection of units and registers for operations and variables, the routing of the variable paths from the producing operation to the consuming operations, and avoiding deadlocks caused by overcharging the transport resources. The last problem is the hardest in practice.

We solve the instructions scheduling problem for these partially connected architectures by translating the problem into a two-dimensional space-time graph representation. From this representation we derive three scheduling algorithms. The first algorithm has the advantage that the deadlock test is easy, but the scheduling of the operand paths is hard due to its cycle-based nature. Therefore the other two algorithms use an operation-based approach which allows a more efficient scheduling of the operand paths at the cost of a more complex deadlock test. The third algorithm is the most advanced one. It integrates the routing of variables and the deadlock test and allows already scheduled paths to be rescheduled if that is beneficial. We expect that the third algorithm gives the best scheduling results.

# Contents

# 1 Introduction

Traditionally computer architectures have been designed around a number of centralized resources to which all other resources are connected with a fully connected communication network. This is very easy from the point of view of the compiler writer. The selection between instances of the same resource (e.g. choosing a register or a functional unit) is simple because all are equivalent in terms of communications. For example, two registers in the same register file are equivalent, so the choice between them is trivial. The fact that transport resources are never a bottleneck is also very easy for a compiler because it allows the compiler to ignore all communications.

For the hardware designer's point of view, this organization is not so good. The centralized resources tend to grow out of hand if the size of the machine increases. For example the number of register files ports depends linearly on the number of functional units and soon becomes impractically high on wide machines. The fact that all resources must be connected to the centralized resources, may require long wires to connect all remote resources. This is a problem because long wires are particularly slow.

The solution to the hardware problems is to physically distribute centralized resources over the processor and to avoid long wires where possible. A well known way to do it, is clustering. In a clustered processor each cluster is organized as a traditional architecture, and communication between clusters is handled by a separate network with a relatively low bandwidth and high latency. Clustering can be seen as an attempt to solve the hardware problems, while still being able to use the well-known compiler techniques within a cluster.

If we could solve the compiler problems that arise when resources are distributed and the connectivity of the transport network is reduced, than we would be able to support arbitrarily connected processors. It would allow us to support processors which are efficient in hardware, but do not have the artificial cluster boundaries found in clustered processors. Such an architecture is the ReMove architecture [19], which is described in more detail in section 2.

In this report, we will focus on the code generation for partially connected processors. As already mentioned, this type of processors are not easy to write compilers for. We will now identify the main problems that arise during the instruction scheduling.

- Selecting a functional unit for each operation. If we choose the wrong unit, then the transport of operands from the producing operation to the consuming operation may take too much time and therefore increase the schedule length.
- Finding a path for all operands. In a partially connected machine, it is not guaranteed that a variable will be used in a unit which is directly reachable from the unit in which that variable was produced. If there is no direct path between the producer and the consumer, the variable must be moved through one or more intermediate units or register files. Also if the consumer is not scheduled directly after the consumer, the compiler must decide where to store the variable temporarily.
- Coping with a limited amount of communication resources. Especially the restricted number of register file write ports can lead to problems because it may lead to a deadlock, as we will explain in section 3.

In compilers for architectures with a centralized register file, a variable is usually directly coupled with to a physical register. This can be done because all registers are equivalent, and it is never needed to move a variable to another register.[1] In the architectures we are considering, the direct relation between variables and registers is not practical. A variable may have to go through many registers, in register files as well as in functional units, in order to reach its destination, the operation which uses it. If there is more than one consumer, then it may even be advantageous to have separate paths to these consumers, and therefore have the variable stored in more than one register at the same time. It is this relationship between variables and registers that makes that most existing compilers are hard to use for partially connected architectures.

This report is organized as follows. In section 2 we will describe the ReMove architecture which is used as our main example of an architecture with limited connectivity. In section 3 we continue with an overview of the code generation problem for that type of architectures. This section also explains the deadlock problem in more detail.

In sections 4 to 6 we focus on three approaches for the instruction scheduling phase. These approaches differ in the way they handle the deadlock problem. Section 4 describes a cycle-based scheduling algorithm for which it is relatively simple to avoid deadlocks.

Operation-based scheduling has a clear advantage for ReMove and similar architectures because it allows the scheduling of operand paths to be delayed until the consumers have been scheduled. This is an advantage because if we know where an operand will be consumed,we can schedule the path directly into the correct destination without having to guess where the destination of the operand will be. We can therefore expect an operation-based scheduler to give better results than a cycle-based scheduler. Section 5 describes a method to detect deadlocks in an operation-based scheduler. It also describes an efficient method to schedule the operand paths. Section 6 describes an operation-based scheduling method which integrates the deadlock test and the scheduling of the operand paths. It also allows already scheduled operand paths to be rescheduled, which is a major difference with the method of section 5.

These three scheduling approaches all use the same algorithm to select the best unit for each operation. This algorithm is described in section 7 for the ReMove architecture. Other architectures may need other algorithms tuned for their geometrical features. Section 8 concludes this report.

[1] Variables in loops are sometimes moved to another register if the lifetime is longer than a full iteration. This case is usually handled by explicitly copying the variable into a new variable, and therefore into another register

# 2 ReMove architecture

One example of a partially connected architecture is the ReMove architecture. It is a computer architecture which resembles a traditional VLIW architecture which is modified on key points to simplify the hardware. The result is a highly scalable architecture that allows higher clock speeds or wider processors with equal costs compared to VLIW architectures. The power consumption also decreases. We want to achieve these results by making the bypass network visible to the programmer and by using short, single source busses wherever possible.

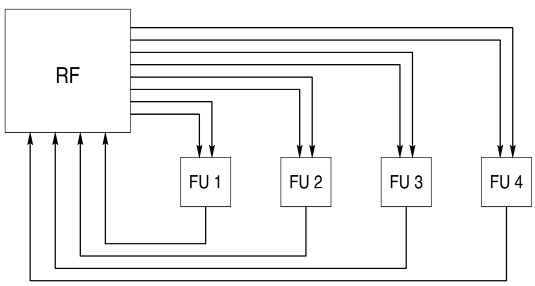


**Figure 2.1:** Datapath of a traditional VLIW. Although all VLIW processors do have an extensive bypass network, this network is not visible for the programmer, and therefore not shown in this figure.

The figures 2.1 through 2.5 show the differences between a traditional VLIW and ReMove in a step-by-step manner. Starting from a traditional VLIW (Figure 2.1) four major modifications are needed to come to the ReMove architecture.

- Explicit bypasses (Figure 2.2). In a traditional VLIW all operands for the functional units come from the register file, and the result is always written back into that register file. For hardware efficiency reasons there are bypasses, but they are not visible for the programmer. Figure 2.2 shows a VLIW in which the bypasses are made visible. If they are also visible to the programmer, the operands of the functional units are not registers in a register file, but output registers of a functional unit or the register file.

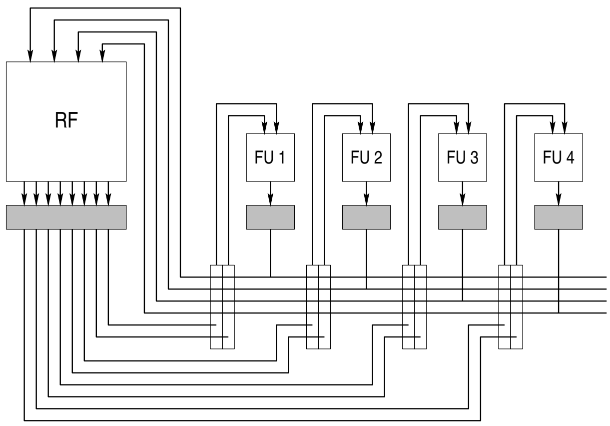


**Figure 2.2:** VLIW architecture in which all bypasses have been made visible. While in a traditional VLIW the operands for the functional units may only come from the register file, in a VLIW with explicit bypassing the operands may also come from an output register of a functional unit.

If this were the only difference, the scheduling algorithm would hardly change, apart from the encoding of the instructions. The same schedule as generated for the normal VLIW would still work because there are no fewer resources. The only place where there is a modification, is when scheduling the first cycle after a branch because the specification of the register operands and the operation is split. As the output of the register file is written into an output register before it can be used, the registers that will be used when executing the next instruction have to be specified in the current instruction. This is never a problem, except when it is not yet known which instruction will be the next as is the case for the last instruction before a branch.

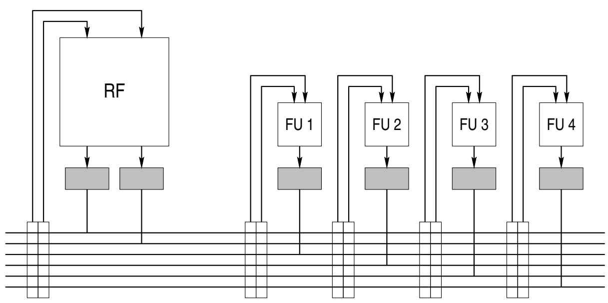


**Figure 2.3:** VLIW architecture with explicit bypasses and restricted register file bandwidth. In comparison to the architecture shown in Figure 2.2 this architecture has fewer read and write ports.

- Limited register file bandwidth (Figure 2.3). Many values that are generated during normal calculations have only one consumer, which is often scheduled directly after its producer. These values do not have to go through the register file. Also, a lot of operands are immediate values that are generated locally and do not come from the register file. These two factors make that the average bandwidth to and from the register file is significantly less than the worst case bandwidth of two reads and one write per functional unit. The cost and the size of the register file can be reduced significantly, while its speed is increased if the number of register ports is adjusted to the average instead of the worst case bandwidth as is customary for traditional VLIW architectures. It is estimated that on average around three times fewer register file ports are needed. Research on the Move project has shown that scalar code needs on average 0.5 read ports and 0.35 write per operation [11].

Scheduling becomes a lot harder when the number of register file ports is restricted. Care has to be taken that all required register file operands are written to output registers in time, at the cost of slower operation if some operands arrive too late. Even harder is the result register problem as it may result in incorrect behavior if left unsolved. All result registers have to be written back into the register file if they are not bypassed to all their consumers or copied to another output register. In the previous architecture these problems simply did not exist because the register bandwidth was designed for worst case and therefore guaranteed to be enough.

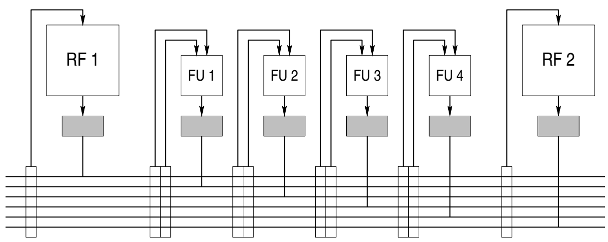


**Figure 2.4:** VLIW architecture with partitioned register files. This architecture has several small register files which are more efficient in hardware compared to the the single large register in traditional VLIWs.

- Partitioned register files (Figure 2.4). Often it is easier to design two small register files than one file that is twice as large and has twice the number of ports. The small files together are usually still smaller than the large register file, while their speed is also better. Therefore it can be an advantage to split the large centralized register file in the previous architecture into several smaller ones. A slightly higher number of registers or a few more ports may be needed to keep the performance of the small register files equal to the performance of the large centralized file, but usually this will not outweigh the advantages of a partitioned register file.

  Scheduling for partitioned register files is not very hard as long as the connectivity of the files is equal, as it is in the fully connected architecture shown in Figure 2.4. This principle was already used in the Move architecture. Results from that architecture show that even simple round-robin allocation of registers yields good performance.

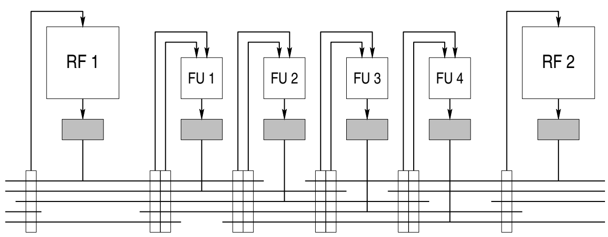


**Figure 2.5:** ReMove architecture. When compared to the architecture shown in Figure 2.4 this architecture has a communication network that is not fully connected, but consists of short busses that lead from a unit to a limited number of neighbors. The absence of long data busses makes this architecture very scalable as the width of the architecture does not influence its bus delay.

- Skewed interconnection pattern (Figure 2.5). The architecture as shown in Figure 2.5 contains all key elements of the ReMove architecture. Its skewed interconnection pattern, in which each unit is connected to a limited number of neighbors on each side, is intended as a way of achieving the benefits of clustering, but without the hard border between clusters and the cycle delay involved with inter-cluster communication. It is this step that solves the wiring problem. All data busses now have a short length, regardless of the width of the machine.

The skewed interconnection pattern makes the allocation units for operations important. The scheduler must put related operations on units that are close enough to each other to be able to communicate results directly. The distribution of registers over the register files also becomes more important than it was for the fully connected machine. If a value is in the wrong register file or in an output register of a remote unit, it will take one or more cycles of copying before it arrives at its destination. If the destination is an operation on the critical path, this may lengthen the schedule.

# Code generation 3

This document focuses mainly on the instruction scheduling for ReMove and other architectures with limited connectivity. The instruction scheduler is part of the back-end of compiler. As can be seen in the overview of the code generation trajectory in figure 3.1, the back-end translates a sequential intermediate representation of the compiled program into the parallel assembly code.

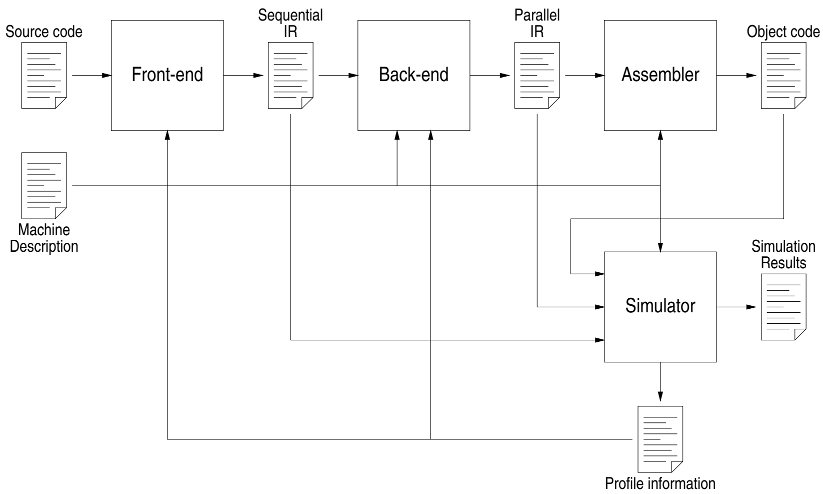


**Figure 3.1:** Flow chart of the ReMove tool set.

Instruction scheduling is the task of assigning a unit and a cycle to each operation in the sequential code. For the architectures we are considering, the instruction scheduler is also responsible for the scheduling of identity operations and register file reads and writes where needed to ensure that all values are transported from the producer to its consumers.

We will first describe the scheduling problem as a graph problem in section 3.1. The main problem encountered during scheduling is avoiding the deadlocks that may occur. This is explained further in section 3.2. After the current section, we continue with three approaches (sections 4, 5 and 6) to do the actual scheduling. The selection of the correct unit for all operations is handled in section 7.

## 3.1 Space-time representation of the scheduling problem

An instruction scheduling problem can be translated into the problem of finding a mapping from the data flow graph onto a model which represents the target machine and its connections spread out in time. That model is a graph in which each resource is represented in each cycle by a certain subgraph. The connections between the subgraphs represent the communications which are possible in the processor. This scheduling model of a machine will also be called a space-time graph. The scheduling algorithms explained in the following sections are based on the space-time graph.

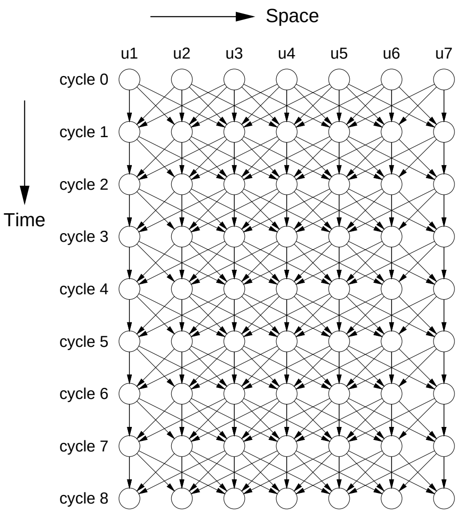


**Figure 3.2:** Graphic representation of the space-time graph of a ReMove processor with seven functional units and no register files.

As an example we will use a simplified model of the ReMove processor. For other processors, including Move, a similar model can be developed. Figure 3.2 shows the space-time graph of a ReMove processor with seven functional units and no register files. Every functional unit is represented by a single node in each cycle in this graph. In this machine each unit is connected to itself and to the two nearest neighbor units on both sides. In the graph this is reflected by edges from a unit in a certain cycle to the units to which it is connected in the next cycle.

Figure 3.3 shows how the code of a four-entry IDCT maps onto the model presented in figure 3.2. Every node with an '=' represents a copy, i.e. a unit which performs an identity operation in a certain cycle, in order to move an operand to the operation which uses it.

The example above contains only simple functional units, no register files and busses which can be written by only one unit. This is why each unit is represented by a single node in the example, and the busses are only represented by arcs. In general the model for a resource may be more complicated. It may contain more than one node, and when it is capable of processing more than one data item per cycle its nodes may have a capacity associated with them that is larger than one.[1]

We will now give the model for a register file with read and write ports, which has a far more complex model than the resources considered so far. It consists of three nodes per file per cycle,

[1] A node with a greater capacity than one can also be represented with several parallel nodes with a capacity of one each, but this is usually far less efficient.

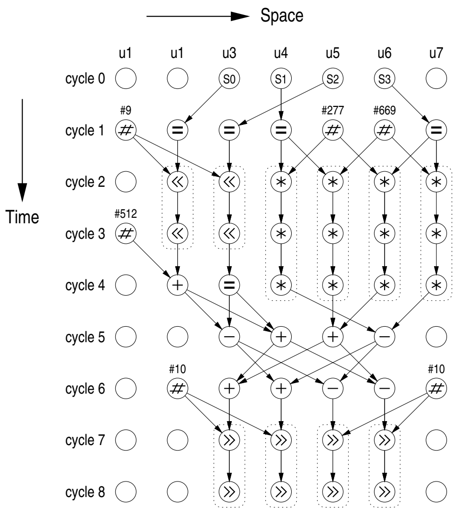


**Figure 3.3:** The data flow graph of a 4-point inverse DCT mapped onto the machine model of figure 3.2. We assume that shifts take two cycles, and that multiplies take three cycles on this machine, and that these operations are not pipelined.

and each of these nodes may have a capacity larger than one. The model also includes a number of edges, of which at least one is inter-cycle, which means that the destination node of the edge is in a later cycle than the source node.

One node in the register file model is reserved for the registers themselves, the second is for the write ports, and the third node represents the read ports. The capacities of these nodes are the same as the capacities of the corresponding resources. So the capacity of the node representing the file itself has a capacity equal to the number of registers in the file, and the capacity of the read and write port nodes is equal to the number of read and write ports, respectively. The incoming edges of the write port node and the outgoing edges of the read port node specify the connections of the register file with the outside world.

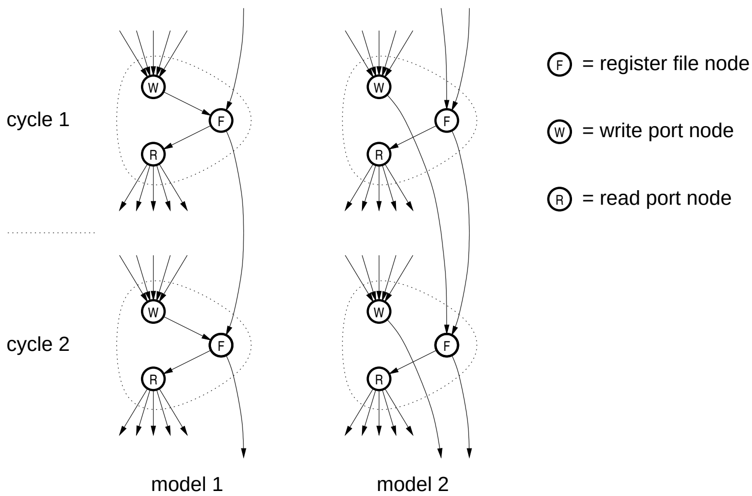


**Figure 3.4:** Space-time representation of two register files, covering two cycles. The register files differ in their behavior during a simultaneous read and write of the same register.

Figure 3.4 shows two examples of models for register files. The examples contain two cycles each, so the edges between nodes of the register file in different cycles are clear. The register files represented in those examples differ in the behavior of writes into the register file. Values which are written in the left hand file can be read in the very same cycle, so a read specified in the same cycle as a write to the same register, will yield the value that was just written. The register file on the right hand side however has the more traditional behavior that a read will retrieve the old value if a read and a write to the same register are executed in the same cycle.

## 3.2 Avoiding deadlocks in the scheduling algorithm

The architectures we are considering will typically have fewer register file write ports than functional units. This does not have to affect the performance significantly because research in the Move architecture has shown that less than 40% of the results has to be written into a register file. This is too large an opportunity for optimization to ignore when designing a partially connected processor.

The direct consequence of having fewer register file write ports than functional units is that we cannot assume that there is a write port available for every result that is generated. All results which have to be kept alive but cannot be written into a register file, will have to be stored in functional units using copy operations. The compiler has to take care that those units are available. If it is too aggressive in claiming functional units for operations, there may not be enough of them left for the required copy operations, and a scheduling deadlock will occur.

For live all results for which no register file write has yet been scheduled, the scheduling algorithm must make sure that it can find a path to the (yet unscheduled) consumers or to a register file, either direct or via copy operations. A result is called live if at least one of the operations that uses it has not yet been scheduled. If all consumers of a value have already been scheduled, it is obviously not necessary to keep it in a register any longer.

A deadlock occurs if at least one variable cannot reach its consumers, regardless of the position and time where all yet unscheduled operations will be placed. It is guaranteed that a valid schedule is still possible when we can prove that all live results can find a path to the end of the schedule.[2] Although this is a slightly conservative condition, it is hard to find a less conservative one that can be verified at any point during scheduling. The problem of avoiding deadlocks is one of the hardest problems encountered when scheduling for ReMove.

The proposed deadlock test checks that all results with at least one unscheduled consumer can be allocated on a path to the end of the schedule. If they are in an output register at the end of the schedule, then it is guaranteed that they can reach a register file in a later cycle because there are no scheduled operations which can stand in the way. Reversely, if all live results can reach a register file, then it is also possible that they can reach the end of the schedule because we assumed infinite size register file. Consequently, the deadlock test is also verified if we can allocate a path for all live results to a register file.

For a cycle-based scheduler this problem can easily be solved by a bipartite matcher because there is only one cycle to be checked. A scheduling algorithm based on this principle is presented in section 4.

Unfortunately, using a bipartite matching test does not work when the check should cover more than one cycle, which is needed for an operation-based scheduler. Operation-based scheduling has a clear advantage for ReMove because it allows scheduling of operand paths when their destination is known. An operand path is a path of an operand of an operation from the point in time and space where it is defined to the point where it is consumed. If we know where an operand will be consumed, we can schedule the path directly into the right direction without having to guess where the destination for the operand will be.

[2] We assume that the capacity of the register files is unlimited. If not, then even on a fully connected machine the scheduler could get into trouble for code that uses so many registers that a valid schedule without spill code is not possible.

A very simple way to avoid deadlocks in a operation-based scheduler is to claim a path to a register file for each result when an operation is scheduled. This claim can be released when the compiler can be sure that it is not needed any more, for example if all consumers have been scheduled. This is the approach taken by the Move compiler. It is however not very efficient because it temporarily claims too many resources and needs some spare resources to be able to generate efficient schedules. For Move, this effect is visible in the number of register file ports the scheduler needs to be able to generate efficient schedules. Hoogerbrugge [10] has shown that a balanced Move processor should have as much write ports as read ports on the register file, even though on average it uses far fewer write ports in the final schedule.

In section 5 we describe a more sophisticated method which is capable of checking for deadlocks in more than one cycle and allows an operation-based scheduler to be used for ReMove. It does not claim any resources which may be freed later, so it may be more efficient than the Move-approach described above.

# Cycle-based scheduling 4

A cycle-based scheduler tries to fill one cycle at a time. It will not try to schedule any operations in the next cycle if the current cycle is not yet completed. On the opposite, an operation-based scheduler uses an operation-by-operation approach. It tries to schedule an operation in the current cycle, but if it fails, it will try the next cycles until it has found a place where it can be scheduled. During operation-based scheduling, there may be several partially scheduled cycles, while during cycle-based scheduling only the current cycle may be partially scheduled.

For a cycle-based scheduler, the deadlock test as described in section 3.2 is relatively simple. The compiler needs to look only at the current cycle when it checks if it is possible to find a path to the end of the schedule for all results. This test is based on so-called 'bipartite matching' graph problem, and is described in more detail in section 4.1. In section 4.2, we continue with the algorithm for the cycle-based scheduler. One important part of the algorithm is not described there. The selection of units for the operations is the same for all scheduling algorithms described in this document, and is therefore described separately in section 7.

## 4.1 Deadlock test

It is easy for a cycle-based scheduler to check if it is possible to find a path to the end of the schedule for all results because it only needs to look at the current cycle. If it can find a register file write port or a free unit for all live results, then the check is satisfied. As nothing has yet been scheduled beyond the current cycle, all result values which are copied into a result register will be able to find a register file write port in the next cycle or beyond. In each cycle beyond the current one, the number of live results in result registers can be decreased by the number of register file write ports, until all result values are safely stored in the register files.

The check a cycle-based scheduler must perform, is a simple bipartite match, as shown in Figure 4.1. The bipartite matching problem is the well-know graph problem of finding a *perfect matching* in a *bipartite* graph. A bipartite graph is a directed graph $G(V, E)$ that consists of two sets of nodes, $V_1$ and $V_2$, such that all edges in the graph are directed from a node in $V_1$ to a node in $V_2$. A matching is a subset of edges $M \subseteq E$ such that for all nodes $v \in V$, at most one edge of $M$ is incident on $V$. A perfect matching is a matching in which all nodes have one incident edge in

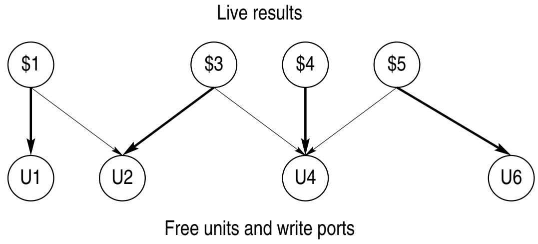


**Figure 4.1:** A cycle-based scheduler can use a bipartite matching to check if it is possible to find a write port or free unit for each live result. In this example 4 live results have to be matched with four free units or write ports. The arrows show which units can be reached by the result values, and the thick arrows show a possible matching.

M. The bipartite matching problem can be solved optimally in $O(nm)$ time where $n$ is the number of nodes and $m$ is the number of edges [3].

In the deadlock test there is a node for each live result at one side of the bipartite graph. The other side of the graph contains a node for each free unit and each unused register file write port. There is an edge between a value and a unit or write port node if there is a bus from the result register in which the value is stored to the destination unit or write port. If the maximum bipartite match of the resulting graph contains a edge for each live result, then it is possible to store all live results in a register file or in a result register in the current cycle, and consequently to have all live results stored in a register file in some future cycle.

Each cycle in the space-time graph as described in section 3.1 has a resemblance to the bipartite graph used for the deadlock test. This is not just a coincidence. The bipartite graph had variables on one side, and register files write ports and copy units on the other side. The units are resources which are directly found back as nodes in the space-time graph, and the write ports can also be found there. The variables on the other hand have no direct relation to the space-time graph. There is however an indirect relation. Each variable that is produced in a cycle must be located in a resource. It must be at a register file read port if it was read from the register file, or it must be at a unit if it was created that cycle or be moved there by a copy operation. These units and read ports can be found in the space-time graph.

The way the bipartite matching deadlock test works, makes it particularly fit for a cycle-based scheduler. It makes a choice for each cycle where and for which results a copy must be scheduled, just as a cycle-based scheduler makes a choice for each cycle which operations must be scheduled, and where they must be scheduled. The cycle-based scheduler can therefore schedule copies as normal operations, and use the placement strategies already used for the 'real' operations. We will now continue with a scheduling algorithm that uses this principle.

## 4.2 Cycle-based scheduling algorithm

The scheduling algorithm is responsible for the allocation of operations to units and time slots in the schedule. It also does the register allocation and generates spill code where needed.

Scheduling is done one block at a time, even for inter basic block scheduling. All operations which may be scheduled in the current basic block are put into a dependence graph in the form in which they would appear when the operation would be scheduled in the current block. For example, if an operation must become conditional when imported into the current block, the dependence graph contains a node for a conditional operation with the predicate (guard) and the conditionally overwritten variable as input operands. For intra basic block scheduling the dependence graph only contains operations from the current basic block itself, while for inter basic block scheduling the dependence graph may also contain operations imported from other basic blocks.

All operations in the intermediate format are represented by operation nodes. This format also recognizes def nodes, use nodes and copy nodes. A def node defines an operand and is used to specify in which locations (memory, register files, or result registers) operands which are live-in for the current basic block can be found. A use node is for live-out variables what the def node is for live-in variables. It specifies in which location(s) a live-out variable must be allocated. Copy nodes are used where an explicit copy operation is needed to keep a variable live, and to copy a variable from one type of location to another. The input and the output of a copy node are the same operand.

Procedure 1 describes the scheduling algorithm for one basic block. It starts by creating the dependence graph of the scheduling region, where the scheduling region is the basic block itself and all other basic blocks from which operations may be imported. After the def and use nodes are added to the graph and the dependence edges are inserted, the algorithm calculates the ASAP (As-Soon-As-Possible) and the ALAP (As-Late-As-Possible) times for each node. It also calculates a minimum and an estimated scheduling time for each node. Both can be set to the ASAP time for simplicity. When a second level optimizer is used to iteratively improve the schedule, the optimizer may set the minimum and estimated scheduling times instead. The minimum scheduling time is used to prevent nodes from being scheduled too early, and the estimated scheduling time is used in the heuristic for the placement of nodes on units.

**Procedure 1** SCHEDULEBLOCK(Block $B$)

Create dependence graph *ddg* // *Including def nodes and use nodes*
CALCULATEASAP(*ddg*)
CALCULATEALAP(*ddg*)
CALCULATEMINSCHEDTIME(*ddg*) // *Hard minimum for the scheduling time*
CALCULATEESTSCHEDTIME(*ddg*) // *Estimated scheduling time*
SCHEDULEDEFNODES(*ddg*) // *Schedule live-in variables*
cycle $t \leftarrow 0$
**while** $\exists$ unscheduled node $\in B$ **do**
  SCHEDULECYCLE(*ddg*, $t$)
  $t \leftarrow t + 1$
**end while**

The algorithm then schedules the block cycle by cycle until all operations in the block are scheduled. If one or more def nodes do not have a location, they are scheduled before the first cycle. This happens when none of the predecessor blocks of the current block are already scheduled.[1] Similarly the use nodes, which represent the live-out variables, may have a fixed location. If that is the case, then a use node must be scheduled at the location that is dictated by the successor blocks.

The scheduling of the delay slots (if present) was ignored in the pseudo-code of the algorithm, but is not very hard. The jump that ends the basic block can optimistically be scheduled as any other operation. When the block has completely been scheduled we check if the number of scheduled cycles after the jump is more than the number of delay slots. If that is the case, we conclude that the jump was scheduled too early, and we set the minimal scheduling time of the jump node to the last cycle of the block minus the number of delay slots. Then we reschedule all cycles from the cycle in which the jump was scheduled until the end of the block.

Procedure 2 lists the algorithm for the scheduling of one cycle. Before scheduling, it tries to predict how many operands will be live during the next few cycles and whether there will be a register available for these operands. If there will be a register shortage, it adds spill code. As all operands that are in output registers at the start of the cycle will be lost when they are not explicitly copied to the next cycle, the scheduler adds copy nodes for some operands. If there is at

[1] When another successor of a predecessor of the current block has been scheduled, that block may already have fixed the placement of some operands. Theoretically it is even possible that a predecessor of a successor of a predecessor or a block even further away has allocated a location for a operand of the current block. To avoid such obscure relationships between blocks it is advisable to disallow a split point next to a join point.

**Procedure 2** SCHEDULECYCLE(DependenceGraph *ddg*, cycle *t*)

```
InsertSpillCode(ddg, t) // Speculatively add spill code
InsertCopyNodes(ddg, t)
repeat
   Create list L_ready with all nodes that are ready at cycle t
   for each node n ∈ L_ready in order of priority P_node do
      Create list L_units with all units that support n
      scheduled ← false
      while L_units ≠ ∅ ∧ ¬scheduled do
         select u ∈ L_units on priority P_unit
         ScheduleNode(n, u, t)
         if CheckCopyNodeAssignment(ddg, t) then
            scheduled ← true
         else
            UnscheduleNode(n)
            L_units ← L_units \ {u}
         end if
      end while
      if scheduled = false then
         UpdateAsapTime(n, t + 1)
         if EstSchedTime(n) ≤ t then
            UpdateEstSchedTime(n, t + 1)
         end if
      end if
   end for
   if CheckCopyNodes(ddg, t) then
      reschedule ← false
   else
      UnscheduleCycle(ddg, t)
      reschedule ← true
   end if
until reschedule = false
```

least one node that uses the operand and that node is not ready, a copy node for that operand is added. The scheduling algorithm guarantees that those copy nodes get scheduled in the current cycle. This approach is still too optimistic as there may be ready nodes which cannot be scheduled in the current cycle. That case is caught after the complete cycle has been scheduled. If after scheduling it turns out that there are unscheduled nodes that use operands for which no copy node was reserved, the scheduler adds extra copy nodes and reschedules the complete cycle.

The cycle scheduler visits all ready operation nodes and copy nodes and tries to schedule them in the current cycle. Which node is tried first is decided by a priority function. For each node, the scheduler identifies all units on which the node can be scheduled and calculates the preference $P_{unit}$ of these units for the node according to the method described in section 7. It schedules the node on the most desirable unit for which all requirements are met. A node can be scheduled on a unit if the following five requirements are met:

- The unit must be free, i.e. not used by another operation.
- The unit must support the operation.
- The unit must be able to read all source operands.
- If the node writes into a register file, there must be a free register to store the result.
- It must remain possible to schedule all remaining copy operations needed to keep variables live for more than one cycle. This is the deadlock test.

The function CheckCopyNodeAssignment is used to check if it is possible to find a unit for all copy nodes which have to be scheduled in the current cycle. This function implements the deadlock test described in section 4.1. Essentially it checks if there exists a bipartite matching between the set of unscheduled copy nodes and the set of available units, where each copy node can be scheduled on all units which haven't got an operation yet and which can read the operand that has to be copied. When a copy node is scheduled on a register file unit, the operand is written into the register file.

If there is no unit available on which the node can be scheduled, the scheduler postpones it to the next cycle. The ASAP time is increased by one and if the node was predicted to be scheduled in the current cycle the estimated scheduling time is also increased. When the complete cycle has to be rescheduled because extra copy nodes are needed, the ASAP time of all nodes is reset to its original value. The estimated scheduling time however keeps its new value during the next pass because it is likely that the node which could not be scheduled in the current cycle in the first pass will also fail to be scheduled in the next pass.

# 5 Operation-based scheduling

Scheduling for a partially connected architecture like ReMove can be done most efficiently if the paths from the producer to the consumers of a variable are not scheduled before it is known where the consumers will be placed. To be sure that no deadlocks can occur, we require that our deadlock condition can be verified at any point during scheduling. If the operand paths, i.e. the paths by which a variable is transported from its producer to its consumers, are left unscheduled until the consumers themselves are being scheduled, then the deadlock test must be able to handle multiple partially scheduled cycles.

For the cycle-based scheduler we proposed a bipartite match as deadlock test. In section 5.1 we will show that this test does not work as well for an operation-based scheduler because it has to be very conservative to work correctly. In section 5.2 we will describe a deadlock test based on the maximum flow graph problem which perfectly implements a deadlock test for an operation-based scheduler. It correctly detects all deadlocks, but is never too conservative, as the bipartite match test may be. In section 5.3 we will describe a simple algorithm to schedule the operand paths which combines well with the maximum flow deadlock test.

## 5.1 Multi-cycle deadlock test based on bipartite matching

Unfortunately the bipartite matching method does not work well when more than one cycle has to be checked. During operation-based scheduling, live results may be produced in several cycles, while write ports are also available in several cycles. Operations scheduled several cycles ahead may block the copying of results generated in earlier cycles. The bipartite matcher is not capable of handling these complexities.

A potential solution, which unfortunately does not work, would be to put the results generated in one cycle together in one bipartite graph, with on the other side all write ports available in all cycles. Unfortunately such a representation cannot model the resources needed to copy a live result to a later cycle in which it can be written into a register file. This solution may therefore fail to detect some deadlock situations.

A correct, but overly conservative solution would be to use a separate bipartite match for each cycle which must be checked. To make this work correctly, live results that are assigned to an output register by the bipartite matcher in a certain cycle must be transfered to the next cycle, in which the bipartite matcher for that cycle handles them again as live results. This means that there is an inter-cycle relationship which cannot be handled efficiently by the bipartite matchers. We will illustrate this with an example (see figure 5.1).

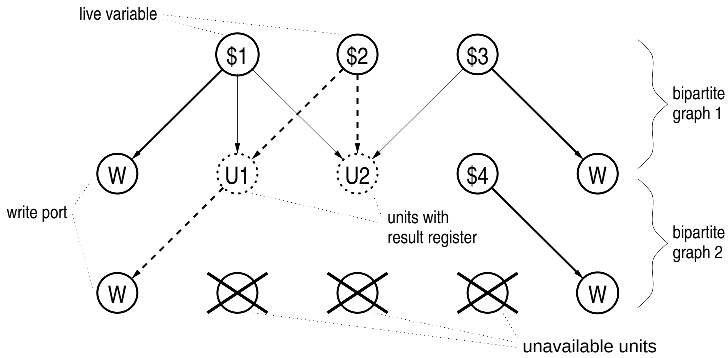


**Figure 5.1:** Example of a situation where a bipartite matching tester may incorrectly detect a deadlock. If the first bipartite match assigns variable \$2 to unit $U2$, then the second matching will not find a solution. But if the first matching would have coupled \$2 and $U1$, then a solution would have been possible.

Suppose that the first cycle which has to be checked, generates three live results, while there are only two free write ports on the register files, but several free units which can copy the third result. The bipartite matcher will correctly report no problems for the first cycle. However, in the second cycle there is now at least one result register used, but we do not know which one it is because the first cycle left several options. When creating the bipartite graph for the second cycle we cannot know into which register files the result from the first cycle can be written, because that depends on the result register where that result is stored.

We can conservatively make this method work by selecting a result register before proceeding to the next cycle. That may sometimes cause the check to return false, while saving all live results would have been possible if another choice would have been made in an earlier cycle. Figure 5.1 shows such a situation. The scheduler would still return valid results, but those result would be worse than needed. Sub-optimal results are most likely in large basic blocks, where many cycles may have to be checked, and in blocks with much parallelism (ILP), where many results are generated each cycle, and where careful scheduling is needed to exploit all available parallelism. Therefore we would like another deadlock test which also handles multiple partially scheduled cycles efficiently.

## 5.2 Multi-cycle deadlock test based on maximum flow in a graph

Scheduling for a partially connected architecture like ReMove can be done most efficiently if the paths from the producer to the consumers of a variable are not scheduled before it is known where the consumers will be placed. To be sure that no deadlocks can occur, we require that our deadlock condition can be verified at any point during scheduling. If the operand paths, i.e. the paths by which a variable is transported from its producer to its consumers, are left unscheduled until the consumers themselves are being scheduled, then the deadlock test must be able to handle multiple partially scheduled cycles.

A check whether it is possible to find a path for each live result to a register file write port is too complex for a checker based on bipartite matching if the check should cover more than one cycle, as required when the scheduling of operand paths is postponed until the consumers of a variable are scheduled. In this section we will describe a method which is capable of doing an optimal multi-cycle deadlock test. The method is based on finding the maximal flow in a flow graph.

A flow graph is a directed graph with a source node, where "material" is produced, and a sink node, where "material" is consumed. The material can flow from node to node through the edges. With the exception of the source and sink, nodes are not allowed to produce or consume material, so the amount of incoming flow must be equal to the amount of outgoing flow. The maximum flow is the maximum amount of material that can flow through the graph from source to sink without exceeding the capacity of the nodes.[1] Finding the maximal flow is a problem which can be solved in $O(nm \log(n^2/m) \log n)$ where $n$ is the number of nodes and $m$ is the number of edges [7].

In the ReMove scheduling problem, the topology of the processor can be translated into a flow graph through which the live results can flow to the register files. If the maximal flow through the network is equal to the number of live results that must be written into the register files, then there is a way in which each of these results can reach a register file write port, and the check condition is satisfied.

The space-time graph of a ReMove processor forms the base of the flow graph. The space-time graph depicts the units in space in one dimension, and also shows the time in the other dimension. The principle of the space-time graph was explained in section 3.1.

The flow graph is constructed from the space-time graph by the following procedure.

1. Cut off the graph at the cycle in which the first live result is defined. The removed part does not represent cycles in which values have to be copied, so we need not inspect it.
2. Cut off the graph after the last partially scheduled cycle. The removed part is unscheduled, and will therefore not prevent any results from being written into the register files.
3. Remove all nodes which represent a unit in a cycle in which that unit cannot be used for a copy operation, e.g. because the unit executes an operation in that cycle.
4. Add a source node and a sink node.
5. Add a node (called a result node) for each live result value. Give this node the outgoing edges that correspond to its unit and time in the space-time graph. Add an edge from the source node to each result node and set the capacity of each result node to 1.
6. All nodes that represent a functional unit or a register file at the last cycle must be connected with an edge to sink node. This represents the fact that a result value is safe if it has reached the last cycle. A deadlock occurs if at least one live result value cannot reach the last cycle.

Figure 5.2 shows an example of a flow graph. It is constructed from the space-time graph of a ReMove processor with four functional units and two register files. The empty spaces in the flow graph are the nodes representing units which cannot be used for copy operations any more in the corresponding cycle, e.g. because an operation is already scheduled on that unit at that cycle. The result nodes are placed at the positions that represent the unit and cycle of the creation of the results. In this example the maximum flow is 6, which is equal to the number of live results. We can therefore conclude that for this example it is possible to find a path for each live result to the end of the schedule.

This procedure is also capable of checking live results which are available at more than one node in the space-time graph. This may happen when a result is consumed by more than one operation, and at least one of these operations has already been scheduled. If copy operations are used to move the result to the location where a consumer is scheduled, then the result is not only available at the node (a unit at a certain time) where it was defined, but also at the nodes used for the copy operations.

If there is only one node (unit-time pair) on which the result available, then the result node has only outgoing edges to the nodes connected to that node. If the result is available on other nodes too (e.g. the nodes on which a copy node was scheduled to propagate the result), the result node has also outgoing edges to the nodes connected to all other nodes on which the result is available.

The maximum flow deadlock test may fail because the capacity of the register files is insufficient. This can easily be detected by redoing the test with unlimited register file capacity. If it is detected

[1] In the graph theory the capacity is traditionally an attribute of edges, but associating the capacities with the nodes is more practical in our case, and does not fundamentally change the problem.

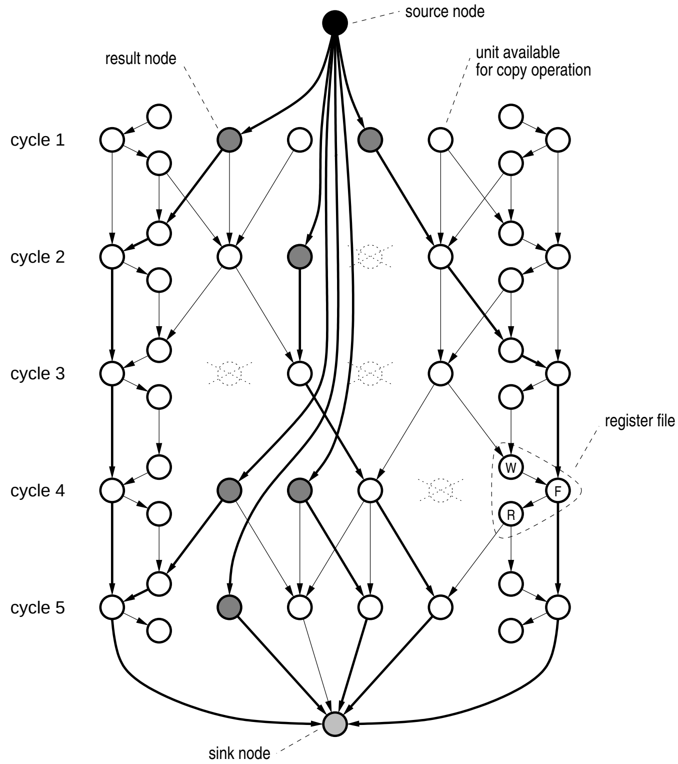


**Figure 5.2:** Example of a space-time graph (representing a ReMove architecture with four functional units and two register files) modified for the maximum flow deadlock test. In this case there is no deadlock because the maximum flow in the graph is equal to the number of live results. The thick lines show a possible solution.

that the register file capacity is the bottleneck, then the scheduling algorithm can decide to insert spill code, although it is not guaranteed that the spill code can be scheduled. Unless registers are reserved for spilling, there may not be enough room in the register files and in the functional units to store all temporary values needed during spilling.

## 5.3 Operand path scheduling

An operand path is a path of an operand of an operation from the point in time and space where it is defined to the point where it is consumed. If we know where an operand will be consumed, we can schedule the path directly into the right direction without having to guess where the destination for the operand will be. This was the main advantage of operation-based scheduling over cycle-based scheduling.

We will now present a simple algorithm by which operand paths can be scheduled. It combines well with the max-flow deadlock test, although we will see that there is still a small problem caused by the greedy character of the path allocation algorithm.

The operand path scheduling algorithm is based on a shortest path calculation. The graph on which the shortest-path algorithm is run, is the space-time graph from which all occupied nodes are removed. Each node in this graph should get a length (cost) which reflects the cost of using that node for the communication. The shortest path is then the path with the lowest cost.

The shortest-path algorithm should calculate the shortest path from the nodes in which the value is stored to the node in which it will be consumed. Having more than one node in which the operand is scheduled (e.g. due to already scheduled copy operations) is hardly more difficult than having only one source. The shortest path problem then has more than one source node, but that does not increase the complexity of the problem. An extra advantage is that split points are automatically chosen. A split point is a point in the space-time graph where a variable is split in two or more separate variables, each going to a subset of the uses of the original variable.

After a path for a variable has been chosen, we must check all live results to see if the path prevents any of these results from reaching a register file. This is done with the maximum flow check as described in section 5.2. If the check fails, we may try a few other paths before giving up.

There is a small problem with the shortest path methodology. It is possible that all chosen shortest paths will turn out to block some live results, no matter how far away we schedule the consumer. The shortest path algorithm may greedily chose a few nodes on its path that are required for the maximum-flow deadlock test to find a path to the end of the schedule for all results. These critical nodes may be on the shortest path to any possible location of the consumer. In that case the scheduling algorithm gets stuck while the deadlock test failed to predict that deadlock.

Fortunately there is a workaround for this situation. If the scheduling of an operation fails because of a deadlock caused by the scheduling of the operand paths, we can first move the operand to the end of the schedule (ignoring the consumer of the problematic operand) using the max-flow algorithm. From that point, the scheduling of the operand path is guaranteed to succeed as there are no critical nodes after the end of the schedule.

The deadlock situation as described above can exist because the operand path scheduling and the deadlock test are independent. The operand path scheduler may take decisions which are not good from the point of view of the maximum flow algorithm. The shortest path algorithm and the maximum flow algorithm are both flow algorithms, so we are left wondering whether it is possible to integrate the operand path scheduling and the deadlock test, and thus avoiding the problems caused by their interaction.

Another opportunity that is not handled in the algorithms described in this section is the possibility to reschedule paths that have been assigned before. It is not unlikely that better schedules are possible if previously scheduled paths are moved out of the way when the need arises. The next section will contain an analysis of the problem of routing all operand paths (including already allocated paths) integrated with a deadlock test, and propose a possible way to solve it.

# 6 Flow scheduling

The problem we are trying to tackle in this section is the scheduling of all operand paths in an operation-based scheduler without creating any deadlocks. It must be detected if a valid routing of the operand paths is not possible, even if other paths are rerouted.

Flow scheduling is the name given to the approach presented in this section to solve that problem. It differs fundamentally from the approach described in section 5 on two points.

- Flow scheduling tries to route all operand paths at once, not only the unassigned paths. It will therefore reschedule paths that were already assigned if that results in a better schedule.
- Flow scheduling integrates the deadlock test. It handles all live results (results which have at least one unscheduled consumer) as normal paths that must be scheduled to the end of the schedule. A deadlock occurs if any path cannot be scheduled.

Even though we will see in section 6.1 that we cannot expect to find an algorithm that can approximate the optimal solution to the flow scheduling problem within any desired precision in polynomial time, we can still expect flow scheduling to be better than the algorithm presented in section 5. There are three reasons why flow scheduling would be better.

- Paths can be rescheduled if they are in the way of operations or other paths. It is therefore possible to schedule operations on units that are already allocated by an operand path if there is an alternative routing for that path. It is simple to think of examples where this makes a difference. One such example is shown in figure 6.1.
- Operand paths with more than one destination can be routed more efficiently because we can consider the complete operand path tree instead of only one destination at a time. In terms of graph problems, the best tree for a variable with multiple destinations is a minimum cost Steiner tree. The operation-based algorithm from section 5 merely calculates a set of shortest paths, which is a poor approximation of a minimum cost Steiner tree. Steiner trees will be explained in more detail in section 6.3.
- The deadlock test is integrated with the operand path scheduling. This integration avoids the problem signaled in section 5.3 that the path routing algorithm may trigger a deadlock because it fails to see a path that avoids deadlocks, while the deadlock tester had earlier recognized that path and therefore not reported a deadlock.

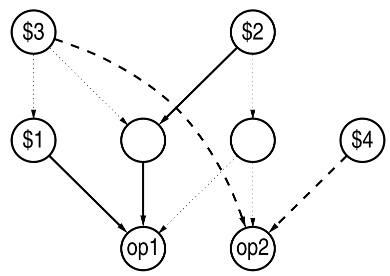


**Figure 6.1:** Example of a situation where the rerouting of an operand path may improve the schedule. Operation `op2($3,$4)` cannot be scheduled in this situation because there is no path for \$3 $\rightarrow$ `op2`. A simple rerouting of the path \$2 $\rightarrow$ `op1` would make a valid schedule possible.

In section 6.1 we will analyze the flow scheduling problem in terms of flow graph problems, mainly variants of the *multi-commodity flow* problem. In section 6.2 we describe an approximation algorithm for the multi-commodity flow problem proposed by Stein [22] and polished later by Klein [14] and Leighton [17]. This algorithm is the basis of our flow scheduling algorithm, though we had to modify it because in our case the variables (commodities) may have more than one destination. The operand paths of those variables with more than one consumer are ideally min-cost Steiner trees.[1] Steiner trees are described in more detail in section 6.3. The flow scheduling algorithm itself is presented in section 6.4.

## 6.1 Analysis of the flow scheduling problem

The problem of flow scheduling consists of finding for each variable the path from its producer to its consumers. We assume that the schedule of the operations is known, so we know when each resource (bus, register, unit, etc.) is claimed by those operations. The communication paths must use the resources that are not claimed by the operations.

The basis of the translation of the flow scheduling problem to a graph problem is the space-time graph, which was explained in section 3.1. All nodes that represent claimed resources must be removed from this graph.

The flow scheduling problem consists of finding a path for each variable from a source node to one or more destination nodes.[2] The source node is the node that corresponds to the resource that produces the variable at the cycle when it is produced. Analogously, the destination node corresponds to the consuming resource when it uses the variable. The capacities of the nodes must be respected, so the number of variables of which the path is allocated on a node must not exceed the capacity of that node.

A variable of which one of the destination is not yet known must be able to reach the end of the schedule. If that is the case, it is guaranteed that the variable can reach its consumer when it is scheduled. Therefore all live results have an extra destination node which is connected to all nodes representing register files or output registers of units in the last cycle. Therefore deadlocks can be detected by checking if all paths can find a path. If any path cannot be scheduled, then we have a deadlock, which must be solved by moving an operation to another location. Usually this will be the last operation that was scheduled.

The flow scheduling problem as described above does not have an exact equivalent in the graph theory. However if we restrict ourselves to one destination per flow, the problem is similar to the *multi-commodity flow problem.* Given an undirected graph $G = (V, E)$, a positive capacity

[1] Steiner trees have nothing to do with Clifford Stein, who proposed the approximation algorithm for multi-commodity flows. The similarity of their names is just a coincidence.

[2] If there is more than one destination node, then the path is in fact a tree.

function $u : E \to R^+$ and a set $\mathcal{T}$ of *connection requests* (also called *commodities*) consisting of a vertex pair $(s_i, t_i)$ and a positive demand $d_i$, the objective is to find paths from source $s_i$ to sink $t_i$ for as many commodities as possible without exceeding the capacities of the edges. The flow conservation constraints must be satisfied, i.e. the incoming flow must be equal to the outgoing flow for all commodities on all nodes except the source and sink.

We can consider four versions of the multi-commodity flow problem.

**Unsplittable flow (USF).** The USF problem has as an additional requirement that each commodity must be routed on a single path.

**Splittable flow (SF),** also called the fractional splittable flow problem. In contrast to the USF problem, the SF problem allows the demand to be split arbitrarily over several paths.

**Integer splittable flow (ISF).** This is a specialization of the SF problem where each path must contain an integral part of the demand. Obviously the demands must be integral in this case.

**Edge-disjoint paths (EDP).** This is a specialization of both the USF and the ISF problem in which the demands and the capacities are all equal to one. In practice, this means that each commodity must be routed on a single path which does not share any edge with the path from any other commodity.

The unsplittable flow (USF) problem can be compared to our flow scheduling problem. The commodities correspond to the variables in the scheduling problem, and the graph is the space-time graph from which all busy resources are removed. The flow scheduling problem differs from the unsplittable flow problem in four respects.

- The demand $d_k$ of the commodities is always one in the flow scheduling problem. This makes the problem slightly easier, and allows the requirement that the flow is *unsplittable* to be replaced with the requirement that the flow is integer valued. The flow scheduling problem is therefore also comparable to the ISF problem.
- In the flow scheduling problem, a commodity (a variable) may have more than one destination. This makes the flow scheduling problem fundamentally more difficult than the USF problem.
- The space-time graph on which we operate is directed and acyclic. At first sight this makes the problem easier, but it actually makes the problem harder. Most of the research on flow problems is done for undirected graphs and little is known of flow problems in directed graphs.
- In the flow scheduling problem the capacities are associated with nodes. This does not really make a difference because the problem can be converted into an equivalent problem in which the edges have capacities by replacing each node with a pair of nodes connected by an edge with the capacity of the original node. All incoming edges of the original node are connected to one of the nodes, and all outgoing edges to the other node.[3] This procedure is shown in figure 6.2.

The fractional splittable flow problem is the only problem mentioned above which can be solved optimally in polynomial time, though the only known approach is to rewrite it as a linear programming problem. In the special case in which the demands and the capacities are all equal to one, both the USF and the ISF problem reduce to the edge-disjoint paths (EDP) problem. This problem was already mentioned in Karp's original list of *NP*-hard problems [13]. The USF and ISF problems are therefore also *NP*-complete.

Even approximation algorithms for the USF and ISF problems are hard to obtain. The best known approximation algorithm for USF [1] achieves an approximation factor of $O(\sqrt{m})$, where $m$ is the number of edges. The approximation algorithms for the USF and EDP problems can roughly be divided into two categories, based on the strategy that is used.

[3] This transformation is possible in our case because we use a directed graph. In an undirected graph this transformation would not be possible because there is no distinction between incoming and outgoing edges.

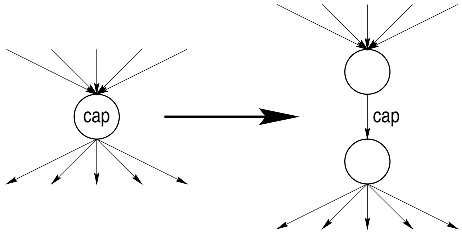


**Figure 6.2:** A graph problem in which the nodes have a capacity can be translated into a problem in capacity is an attribute of edges by splitting the each node. The edge between the new nodes has the capacity of the original node, while all other edges have unlimited capacity.

- The *rounding* approach consists of solving a fractional relaxation of the problem, and then use sophisticated rounding techniques to obtain an integral result. The fractional relaxation is typically a multi-commodity flow [21].
- Under the *routing* approach, a commodity is never split. This approach is hardly used, though a simple greedy algorithm has been proposed which produces reasonable results [16].

Guruswani et al. [9] proved that even the EDP problem on directed graphs is *NP*-hard to approximate with a better approximation factor than $O(\sqrt{m})$. In his thesis Kleinberg [15] makes an interesting statement about the USF problem on directed graphs in which all demands are equal and all capacities are a multiple of the demand. He claims that if there is a fractionally realizable flow, then there is also an realizable unsplittable flow. He does not specify how to find that unsplittable flow but it must be hard to do, otherwise it would contradict with the results by Guruswami.

The flow scheduling problem is also *NP*-hard because it is a fundamentally harder problem than the USF and ISF problems which were found to be *NP*-complete. Because the USF and ISF problems are hard to approximate, we cannot expect to find a scheduling algorithm which is guaranteed to approach the optimal solution within polynomial time.

Even though we cannot expect to find an optimal polynomial time algorithm for the flow scheduling problem, we can still expect that a reasonable approximation algorithm will produce better schedules than possible with the method proposed in section 5. Therefore we will now continue with an approximation algorithm for the SF and ISF problems. Our algorithm for the flow scheduling problem is derived from that algorithm, although the fact that commodities may have more than one sink in the flow scheduling algorithm forced us to make some modifications which invalidated the theoretical bounds on the complexity and the quality of the results.

## 6.2 Approximation of the multi-commodity flow problem

If we allow fractional results, the multi-commodity flow problem can be solved exactly in polynomial time. However, the only known exact algorithm is based on linear programming and has a complexity of $O(k^3n^3\sqrt{km}\log(nDU))$ [23], where $n$ is the number of nodes and $m$ is the number of edges in the graph. $k$ is the number of commodities and $D$ and $U$ are the largest demand and the largest edge capacity. Finding an $\epsilon$-approximate solution takes $O(k^2n^2\sqrt{km}\log(n\epsilon^{-1}DU))$ time using a linear programming approach.

Better running times can be achieved with combinatorial approximation methods. The foundations for these methods were laid by Shahrokhi and Matula [20] in the late eighties. Stein [22] Klein [14] and Leighton [17] have developed the method into a complete algorithm. While they concentrated on the problem of minimizing the congestion of the edges (also called the *concurrent flow* problem), the algorithm was later extended for the minimum-cost multi-commodity problem, in which the objective is to minimize a cost which depends on a cost function associated with the edges [18, 12, 8, 6]. The best approximation algorithms achieve a $\tilde{O}(\epsilon^{-2}knm)$ bound on the total running time[4], so the multi-commodity flow problem takes approximately as much time as $k$ single-commodity flow problems, ignoring polylog factors.

The combinatorial approximation algorithms are all based on the same principle: the minimization of a potential function by iteratively rerouting a part of the flow. The potential function is the sum of the products of the flow $f(vw)$ through an edge $vw$ and the cost $l(vw)$ of that edge, where the cost of that edge depends exponentially on the congestion $\lambda(vw)$ of the edge. In each iteration a commodity is taken and a better path for that commodity is calculated. If there is a better path, than a fraction $\sigma$ of the flow of that commodity is rerouted on the better path, thus decreasing the potential function. The algorithm ends if no commodities can be improved by more than a certain threshold value.

We will now present the approximation algorithm which was proposed by Stein and polished by Klein. It is meant for the unit-capacity case, which means that the capacity of all edges is one. It does not include all optimizations and refinements that were proposed later, but it functions well to show the principles of the algorithm. The theoretical background, including the proof of correctness and the analysis of the running time are far to elaborate to include here, but can be found in [14].

We remind that this algorithm finds a solution to the *concurrent flow* problem, which is a version of the fractional multi-commodity flow problem in which the goal is to minimize the congestion of the edges. More formally, the objective is to minimize $\lambda = \max_{vw \in E} \lambda(vw)$, where the congestion $\lambda(vw)$ of edge $vw$ is defined as the quotient of the flow through the edge and the capacity of the edge, i.e. $\lambda(vw) = f(vw)/u(vw)$. In this case, all capacities are assumed to be one, so the goal is in fact to minimize $\lambda = \max_{vw \in E} f(vw)$.

**Procedure 3** ConcurrentFlow(graph $G$, precision $\epsilon$, set of commodities $\mathcal{T}$)

For each commodity $i$: create a simple path from $s_i$ to $d_i$ and route the demand $d_i$ flow on it.
granularity $\sigma \leftarrow \min_i d_i$
target $\tau \leftarrow \max_{vw \in E} f(vw)/2$
**while** $f$ is not $\epsilon$-optimal **do**
    **while** $\sigma_i$ does not satisfy granularity condition **do**
        $\sigma \leftarrow \sigma/2$
    **end while**
    $f \leftarrow$ Reduce($G$, $f$, $\tau$, $\epsilon$, $\sigma$)
    $\tau \leftarrow \tau/2$
**end while**
**return** $f$

Procedure 3 contains a the outline of the approximation algorithm in pseudo-code. It defines a target $\tau$ for the maximal congestion, defines the granularity $\sigma$ and calls subroutine Reduce (see procedure 4). Reduce returns a flow which either satisfies the target $\tau$, or is $\epsilon$-optimal. In the latter case the algorithm ends, otherwise the target is halved, the granularity is recalculated and Reduce is called again.

The congestion reduction procedure Reduce (procedure 4) contains a loop which in each iteration takes a $\epsilon$-bad commodity and tries to find a better path for that commodity. It then reroutes $\sigma$ units of flow onto the better path.

A path $P$ for commodity $i$ is called $\epsilon$-good if it satisfies equation:

$$l(P) - dist_l(s_i, t_i) \leq \epsilon' l(P) + \epsilon' \frac{max_{vw \in E} f(vw)}{min(D, kd_i)} \sum_{vw \in E} l(vw)$$

[4]The notation $\tilde{O}()$ hides factors poly-logarithmic in $n$ and $\epsilon^{-1}$.

**Procedure 4** REDUCE(graph $G$, flow $f$, target $\tau$, precision $\epsilon$, granularity $\sigma$)

$\alpha \leftarrow (7+\epsilon)\tau^{-1}\epsilon^{-1}\log(7m\epsilon^{-1})$
**while** $\max_{vw \in E} f(vw) \geq \tau$ **and** $f$ is not $\epsilon$-optimal **do**
  **for each** edge $vw \in E$ **do**
    $l(vw) \leftarrow e^{\alpha f(vw)}$
  **end for**
  Call FINDPATH($G$, $f$, $l$, $\epsilon$) to find an $\epsilon$-bad flow path $P$ and a short path $Q$ with the same end points as P
  Reroute $\sigma$ units of flow from $P$ to $Q$
**end while**
**return** $f$

In other words, a path is $\epsilon$-good if it is either almost as short as the shortest possible path $dist_l(s_i, t_i)$ from $s_i$ to $t_i$ (with respect to the edge cost function $l$), or it is so short that it hardly contributes to the total cost. A path that is not $\epsilon$-good is called $\epsilon$-bad. If all commodities are routed on $\epsilon$-good paths, then the flow $f$ is guaranteed to be $\epsilon$-optimal.

Of course the desired precision $\epsilon$ can only be reached only if the granularity $\sigma$ is not too large. Therefore the concurrent flow algorithm reduces the granularity before each call to REDUCE until a certain granularity condition is met. Klein [14] uses a granularity condition which is guaranteed to be small enough in all cases:

$$\sigma \leq \epsilon^2 \frac{\tau}{102\log(7m\epsilon^{-1})} \tag{6.1}$$

This is in fact a rather conservative condition. In many cases a larger $\sigma$ can be chosen, which has the advantage that the potential function decreases faster [6].

The algorithm as described here generates a solution to the *fractional* splittable flow problem. In our case, the variables cannot be split, so we would rather have a solution to the *integral* splittable flow problem or the unsplittable flow problem. Stein and Klein have indicated in their work how their algorithms should be modified to generate integral results. They claim that a reasonably good result can be obtained if we stop as soon as the granularity should be reduced below 1. This causes the algorithm to reroute flows in integral amounts, and therefore avoids fractional results. It should be able to achieve a congestion $\lambda = \lambda^* + O(\sqrt{\lambda^* \log n})$, where $\lambda^*$ is the optimal congestion for the fractional concurrent flow problem, and $\lambda^* = \Omega(\log n)$.

Unfortunately, the granularity condition is impractical in our case. The definition was specified in equation 6.1. The target congestion $\tau$ should be at most one. If we take for $\epsilon$ the maximal value[5] of 0.1, a desired congestion $\tau$ of 1, and a reasonable number of edges $m = 100$, then $\sigma$ should be at most $1.1 \cdot 10^{-5}$, which is far lower than the desired value of 1.

Because of this problem, calculating integral solutions of the concurrent flow method using this procedure is therefore only practical if the capacity of the edges is far higher than the unit of flow. In our case the capacity is one, so this method is of limited practical value. The condition $\lambda^* = \Omega(\log n)$ is also not satisfied in our case. Stein's algorithm for concurrent flows can therefore not be used directly for our problem.

## 6.3 The minimum-cost Steiner tree problem

A major problem with all routing methods based on disjoint paths or unsplittable flows is that they are unable to handle variables which are used more than once. This corresponds to commodities which have more than one sink node. Many of the algorithms proposed for unsplittable flow problems are not easily extended to support flows which have more than one destination.

[5]The theory behind this algorithm has not been proven for $\epsilon > 1/10$

If we want to include commodities with multiple destinations in an unsplittable flow version of the multi-commodity flow algorithm described in the previous section, then the FINDPATH function must be extended. In the case of Stein's and Klein's algorithm it simply calculates a shortest path from the source to the only destination of the commodity. In our case, we would like to find a tree-shaped path that starts from the source node (the node that creates the variable) to the destination nodes (the nodes that consume the variable). The cost of the edges of this tree should be minimal.

The problem we want to solve, is known as the minimum-cost Steiner tree problem, or in short the Steiner tree problem. It is a minimum interconnection problem. Given a weighted graph in which a subset of vertices is identified as terminals, find a minimum-cost connected subgraph (tree) that includes all the terminals. The tree may include nodes that are not terminals.

The Steiner tree problem is *NP*-hard, and even approximations are difficult to obtain. Unless $NP \subseteq DTIME[n^{O(\log\log n)}]$, it is hard to to approximate the Steiner tree problem in a directed graph to a factor better than $\ln k$, in which $k$ is the number of terminals [4]. On the other hand, it is trivial to obtain an approximation factor of $k$ by connecting all terminals to the root with a shortest path.

For the special case of directed acyclic graphs, as in the flow scheduling problem, Zelikovsky [24] has proposed an approximation algorithm which achieves a factor of $k^{1/i}(2+\ln k)^{i-1}$ for any integer $i > 0$. The runtime of that algorithm is $O(\alpha + k^i n^{i-1})$, where $k$ denotes the number of terminals, and $n$ is the number of nodes in the graph. The $\alpha$ stands for the complexity of the all-pairs shortest paths algorithm, which can be done in $O(nm+n^2\log n)$ in this case.

Fortunately the Steiner tree problem is hardly ever a problem during flow scheduling. The reason is that most variables have only a small number of uses in the same scheduling region. In load/store architectures, most operations have two operands and one result. This means that we can assume that the average number of uses per write is around two, regardless of the application. It will be even less if many operations have constants (immediates) as one of their operands. From this observation alone, we can conclude that many variables will be used 0, 1 or 2 times. This qualitative observation is confirmed by the quantitative results by Sohi and Franklin [5]. Around 70% of the integer results is used only once, and this number is even higher for floating point results. Less than 10% of the variables is used four or more times.

For the Steiner trees in the flow scheduling algorithm this means that the number of terminals $k$ is usually small. A tree with one destination terminal is a simple shortest path. If the variable has two destinations, then there is one ***star point*** where the path from the root splits into two paths to the destinations. As there are only $n$ nodes on which the path can be located, the best position can be found in $O(n)$ time after the shortest paths for the terminals have been calculated. When there are three destinations of a variable, then the number of star points is two. The optimal location of these points can be found in $O(nm)$ time[6], which we have judged as too large. If the number of destinations gets beyond three, the time required for an optimal solution grows out of hand very fast. We therefore abstain from complex algorithms to calculate the optimal Steiner tree for variables with three or more consumers.

We use a simple approximation algorithm for the Steiner tree if there are three or more destination terminals. According to Charikar et. al. [2] a simple $\sqrt{k}$-approximation is possible by choosing an intermediate node $v$ and constructing the shortest path from the root to $v$ and the shortest paths from $v$ to all terminals. The intermediate node $v$ is a star point. If the number of destination terminals is two, then this tree is optimal. If there are more terminals, then we construct this tree and assign the path to the closest terminal. Then we repeat the process with the remaining terminals. The process is shown in procedure 5.

This algorithm is simpler than the algorithms proposed by Zelikovsky [24] and Charikar [2]. Its complexity is $O(km + k^2 n)$ instead of the $O(n^{3\log k})$ time required for the $O(\log^2 k)$ approach by Charikar. The performance is guaranteed to be within a factor $\sqrt{k}$ of the optimal because it

[6]The factor $m$ comes from the time required to calculate the shortest path from the first star point to all possible locations of the second star point. This can be done in $O(m)$ time because we assume that the graph is stored in topological order. The $O(m)$ hides the testing of $O(n)$ locations for the second star point

**Procedure 5** STEINER(graph $G$, node $r$, node set $X$)

Tree $T \leftarrow \{r\}$
**for each** node $u \in X$ **do**
  Call SHORTESTPATHTODST$(G, u)$ to calculate shortest path length $d(v, u)$ from each node $v$ in $G$ to $u$.
**end for**
**while** $|X| > 1$ **do**
  Call SHORTESTPATHFROMSRC$(G, r)$ to calculate shortest path length $d(r, v)$ from $r$ to each node $v$ in $G$.
  $v_{best} \leftarrow nil$
  $d_{best} \leftarrow \infty$
  **for each** node $v \in G$ **do**
    $d_v \leftarrow d(r, v) + \sum_{u \in X} d(v, u)$
    **if** $d_v < d_{best}$ **then**
      $v_{best} \leftarrow v$
      $d_{best} \leftarrow d_v$
    **end if**
  **end for**
  $u_{short} = u \in X | d(v_{best}, u)$ is minimal
  $X \leftarrow X - u_{short}$
  $T' \leftarrow path(r, v_{best}) \cup path(v_{best}, u_{short})$
  $T \leftarrow T \cup T'$
  **for each** node $v \in T'$ **do**
    Add zero-length edge from $r$ to $v$
  **end for**
**end while**
$u \leftarrow$ the only node left in $X$
$T \leftarrow T \cup path(v_{best}, u)$
**return** T

improves on $\sqrt{k}$-approximation algorithm. The result after the first star point ($v_{best}$) is found, is already $\sqrt{k}$-optimal according to Charikar [2], and the rest of the algorithm only improves the result.

There is a small caveat with the steiner tree algorithm. Some resources may connect to only one successor resource. An example of such a resource is the bus in a Move architecture. A move bus can write its data to only one destination socket. If a socket has been chosen for a bus, then the the other sockets become unreachable for that bus. Translated to the space-time domain, this means that there are nodes which are not allowed to have more outgoing edge carrying data. A steiner tree cannot have a star point at such nodes.

Although this does not seem to be a hard problem, it turns out to be hard to solve it in our steiner tree creation algorithm. The mere selection of a path to one destination terminal may make other terminals unreachable. This feature makes the problem fundamentally harder than the original steiner tree problem. It is even hard to generate a correct tree, without trying to find the shortest tree.

If we modify steiner tree algorithm to include the constraint that star points can only be placed at nodes that are allowed to have more than one successor, then the algorithm may sometimes fail to find a correct tree, while with other weights on the nodes it may have had no problems. Obviously such behavior can cause problems in the flow scheduling algorithm (see section 6.4) in which the steiner tree algorithm may be executed many times for the same commodity.

In the space-time model for the ReMove architecture nodes which can have only one successor do not exist. As long as ReMove is our main target, we contend ourselves with a very simple workaround which is far from optimal, but at least makes routing possible for all partially connected architectures. If we do not succeed in finding a correct tree during flow scheduling while we know that such a tree exists, then we keep the old tree as if it were the best tree that can be found with the current cost function.

## 6.4 The flow scheduling algorithm

This section describes the flow scheduling algorithm as it will be used by our compiler. We will mainly focus on the routing of the operand paths. The selection of the right unit for each operation is explained in more detail in section 7.

**Procedure 6** ScheduleBlock(block $B$, machine $M$)

```
graph G ← CreateSpaceTimeGraph(M)
commodity set K ← { all live-in variables of B }
while not all operations in B have been scheduled do
  operation o ← SelectOperation(B)
  cycle c ← ASAP(o)
  while o is not scheduled do
    if All free units that support o have been tried for cycle c then
      c ← c + 1
    else
      unit u ← SelectUnit(M)
      btp ← NewBacktrackPoint()
      if TrySchedule(G, K, o, u, c) then
        Commit(btp)
      else
        Reject(btp)
      end if
    end if
  end while
end while
```

Procedure 6 describes the overall scheduling algorithm for one block. It starts with the creation of the space-time graph on which the flow scheduling (the routing of the operand paths) works. The operations are scheduled in an operation-based fashion. The operations are picked one-by-one and the scheduler continues to try scheduling an operation until it has found a cycle and a unit on which the scheduling succeeds. The procedure which attempts to schedule an operation on a specified unit in a certain cycle is called `TrySchedule`, and is shown in procedure 7.

**Procedure 7** TrySchedule(graph $G$, commodity set $K$, operation $o$, unit $u$, cycle $c$)

```
UpdateCommodities(G, K, o, u, c)
return FlowSchedule(G, K)
```

The two main tasks of `TrySchedule` are the updating of the commodities and calling `FlowSchedule` to route the commodities. The commodities contain the information for the variables, where they are produced and where they are consumed. The updating involves the creation o a new commodity for the result of the operation and the addition of new destinations to existing commodities for the operands. If the operation is the last user of the variable, then we must remove the consumer at the last cycle, which was there to avoid deadlocks by ensuring that the variable at least had a way to the end of the schedule. The routing of the commodities themselves is done by the `FlowSchedule` procedure.

The flow scheduling algorithm is inspired by the multi-commodity flow approximation algorithm as described in section 6.2. It is modified to be able to support commodities with more than one destination. The basic functioning is similar: we take a commodity (variable) which has a conflict and reroute that commodity onto another path which has less conflicts. We stop if all conflicts are removed. If there is no commodity which can be routed onto a less costly path and we have still conflicts, then the algorithm must report a failure.

The multi-commodity flow algorithm worked because it was possible anywhere during the execution of the algorithm to decrease the potential function by rerouting a portion of one commodity. This guarantee cannot be made for the flow scheduling algorithm. Although we can define a similar potential function, we cannot ensure that it can be decreased as easily because we have to reroute complete commodities instead of only a small portion. The granularity for the flow scheduling algorithm is therefore too large to ensure similar approximation bounds.

A direct consequence of the large granularity of the flow scheduling problem is that a greedy algorithm like the multi-commodity flow algorithm does not necessarily lead to the global optimum. We may get stuck in local optima where we need two or more modifications to be able to decrease the potential function.

An important role in the flow scheduling algorithm is reserved for the cost function that specifies the cost of a node. We will investigate two possible heuristics to calculate this cost.

- Calculate the cost of the edge only based on the current state of the algorithm. This would mean that the cost of a node is a only function of the capacity and the current amount of flow through that node and possibly some other static factors as the type of the resource represented by the node.
- Construct the cost based on the history of the scheduling algorithm. This may be useful to increase the cost of nodes which are often found to have conflicts. Nodes which are very popular by many commodities will therefore get a high cost and will only be used if there is no alternative.

Intuitively we may predict that the first heuristic is prone to get stuck in local optima, while the second algorithm is more likely to be able to find creative solutions when two or more commodities start fighting over a node. However, the second heuristic is more difficult to implement correctly, as it becomes harder to detect when the algorithm still makes progression. The end condition is therefore not well defined.

A possible way to keep the first algorithm from getting stuck, may be to increase the cost of conflict nodes over time like the $\alpha$ increases when the required precision becomes higher in the fractional multi-commodity flow algorithm. Another option is to accept equally expensive or even slightly more expensive paths if we are stuck. This idea is also used in simulated annealing.

The cost function that we have chosen for flow scheduling tries to integrate two of the ideas mentioned above. It increases the cost of conflicts over time, and it increases the cost of nodes which had a lot of conflicts during earlier executions of the algorithm. The problem of the termination condition is solved by using a number of steps, and keeping the cost function equal during a single step. It can therefore easily be detected if there is still improvement within a step. After each step has ended, the cost function is modified. The cost of conflict is increased, and the history of the step that has just ended is included in the cost function.

The flow scheduling algorithm itself (`FlowSchedule`) is defined in procedure 8. The outermost while-loop iterates over the steps. At the beginning of each step the costs of all nodes are calculated. The only other position where the costs may be changed is when the flow has been changed, which happens just before and after a new Steiner tree for a commodity has been calculated. The cost function is displayed in procedure 9.

The previous path of a commodity is removed is before the new Steiner tree is calculated to get fairer costs. If we would not do this, the Steiner algorithm may incorrectly think that some nodes are overflowed because it has routed both the old and the new path on it. We avoid this by removing the old tree first.

The final termination condition, which specifies how many steps must be taken, is not yet decided. As the cost of overflowed nodes is doubled in each step, it is easy to see that this cost

**Procedure 8** FlowSchedule(graph $G$, commodity set $K$)

Create initial routing for all commodities (defines flow $f$)
Overflow history $H[v] = 0\ \forall v \in G$
$step \leftarrow 1$
**while** $step < N_{step}$ **do**
  $C_{overflow} \leftarrow 2^{step}$
  **for each** node $v \in G$ **do**
    $l[v] \leftarrow \textsc{CalculateCost}(v, C_{overflow})$
  **end for**
  **repeat**
    $is_{changed} \leftarrow$ false
    $is_{correct} \leftarrow$ true
    $H'[v] = 0\ \forall v \in G$
    **for each** commodity $k \in K$ **do**
      $T_{old} \leftarrow$ the old tree of commodity $k$
      Remove the flow of commodity $k$ via $T_{old}$ from flow $f$
      Update the costs of the nodes in $T_{old}$
      $T_{new} \leftarrow \textsc{Steiner}(G, k)$
      Add the flow of commodity $k$ via $T_{new}$ to flow $f$
      Update the costs of the nodes in $T_{new}$
      **if** $T_{new}$ contains any overflowed nodes **then**
        $is_{correct} \leftarrow$ false
        $H'[v] = H'[v] + 1\ \forall \{v \in G | v \text{ is an overflowed node}\}$
      **end if**
      **if** $T_{new} \neq T_{old}$ **then**
        $is_{changed} \leftarrow$ true
      **end if**
    **end for**
    **if** $is_{correct} =$ true **then**
      **return** true
    **end if**
  **until** $is_{changed} =$ false
  $H[v] = H[v] + H'[v]\ \forall v \in G$
  $step \leftarrow step + 1$
**end while**
**return** false

**Procedure 9** CalculateCost(node $v$, cost $C_{overflow}$)

$C_{base} \leftarrow 0$
$C_{saturate} \leftarrow 1$
$C_{history} \leftarrow 1$
**if** $(f[v] \geq u[v])$ **then**
  $l[v] \leftarrow C_{overflow}$
**else if** $(f[v] = u[v] - 1)$ **then**
  $l[v] \leftarrow C_{saturate}$
**else**
  $l[v] \leftarrow C_{base}$
**end if**
$l[v] \leftarrow l[v] + H[v] * C_{history}$
**return** $l[v]$

will soon dominate the other cost factors, and increasing it any further will not help. Probably we should couple the end condition to the cost of overflowed nodes with respect to the cost of the longest non-overflowed commodity.

# 7 Selection of a unit for an operation

The selection of the best unit for an operation is done similarly for all proposed scheduling methods. This method is highly dependent on the topology of the processor. The approach described in this section is mainly intended for the ReMove architecture in which the units are aligned in a linear array. For other architectures other selection methods may give better results.

The UnitPreference function (Procedure 10) calculates the preference of an operation for a unit of a ReMove processor. It calls the function RecursiveUnitPreference (Procedure 11) which calculates a stochastic distribution of the units on which the operation may be scheduled. It looks at all operations that are less than a certain number of data flow edges away from the examined operation, and gives the highest chance to units close to units on which closely related operations are or will be scheduled. If a related operation is not yet scheduled, the function calls itself recursively to find the most likely location for that operation.

**Procedure 10** UnitPreference(DependenceGraph *ddg*, operation $o$, unit $u$, cycle $t$)

```
vector P ← RecursiveUnitPreference(ddg, o, C_depth)
return P[u]
```

The RecursiveUnitPreference function has a *depth* argument which specifies how many edges deep it must look. If the *depth* is 0, it does not take its predecessors and successors into account and simply returns $1/N$, where $N$ is the number of units that support the operation. If *depth* is greater than 0, the function calls itself for all predecessor and successor operation with a depth of one less.

Given the chances that an operation is scheduled on a certain unit for all successor and predecessor operations, the function calculates a cost for each possible unit $u$ for the current operation $o$. For each incoming and outgoing data flow edge it calculates a partial cost which is directly proportional to the expected distance of the units on which both operations (the operation itself, and its predecessor or successor) are scheduled, and inversely proportional to the expected distance in cycles in the schedule. The formula for the total cost is:

**Procedure 11** RecursiveUnitPreference(DependenceGraph *ddg*, operation *o*, depth *depth*)

```
create vector P | P[u] = 0 ∀ u ∈ all units
if IsScheduled(o) then
   P[Unit(o)] ← 1
else if depth = 0 then
   set U ← all units which support o
   P[u] ← 1/|U|   ∀ u ∈ U
else
   set U ← all units which support o
   C[u] ← 0   ∀ u ∈ U
   for each operation o_p ∈ pred(o) do
      P_p ← RecursiveUnitPreference(ddg, o_p, depth − 1)
      d_node ← EstimatedTime(o) - EstimatedTime(o_p)
      for each unit u ∈ all units that support o do
         for each unit u_p ∈ all units that support o_p do
            d_unit ← UnitDistance(u_p, u)
            C[u] ← C[u] + P_p[u_p] * d_unit/d_node
         end for
      end for
   end for
   for each operation o_s ∈succ(n) do
      P_s ← RecursiveUnitPreference(ddg, o_s, depth − 1)
      d_node ← EstSchedTime(o_s) - EstSchedTime(o)
      for each unit u ∈ all units that support o do
         for each unit u_s ∈ all units that support o_s do
            d_unit ← UnitDistance(u, u_s)
            C[u] ← C[u] + P_s[u_s] * d_unit/d_node
         end for
      end for
   end for
   sum ← Σ_{u∈U} 1/C[u]
   P[u] ← 1/(C[u] * sum)   ∀ u ∈ U
end if
return P
```

$$\begin{aligned} C(o,u) \quad = \quad & \textstyle\sum_{o_p \in pred(o)} \sum_{u_p \in units(o_p)} \frac{d_{unit}(u_p,u)}{d_{op}(o_p,o)} \cdot P(o_p,u_p) + \\ & \textstyle\sum_{o_s \in succ(o)} \sum_{u_s \in units(o_s)} \frac{d_{unit}(u,u_s)}{d_{op}(o,o_s)} \cdot P(o_s,u_s) \end{aligned}$$

In this formula $pred(o)$ is the set of predecessor operations and $succ(o)$ is the set of successor operations of operation $o$. The set $units(o)$ contains all units on which a operation $o$ can be scheduled. $d_{unit}(u_1,u_2)$ is the distance between the units $u_1$ and $u_2$ on the array of units in a ReMove processor. The distance between two operations is measured by $d_{op}(n_1,n_2)$, which gives the estimated number of cycles in the final schedule between the operations $o_1$ and $o_2$. The estimated scheduling times are taken from the EstimatedTime entry of each operation. The chance that a predecessor or successor operation $n$ will be scheduled on a unit $u$ is given by $P(o,u)$.

The costs are transformed into chances by taking the $1/C$ and normalizing their sum to 1. This only works if the cost never becomes zero. If care is taken that the unit distance is always larger than zero, this condition will always be satisfied.

# 8 Conclusion

We have proposed three different methods to schedule for architectures with partial connectivity. e are not aware of any earlier algorithm that provides a similarly complete solution to this problem, though for selected number of topologies algorithms have already been designed. The major problem encountered when scheduling for architectures is the possibility that deadlocks may arise if due to aggressive scheduling of operations the communication bandwidth becomes too low to be able to schedule all operand paths.

For cycle-based schedulers, a simple test exist to detect deadlocks because there is only one partially scheduled cycle. We built a complete algorithm around such deadlock test, though a cycle-based scheduler is probably not the best solution for the scheduling problem on partially connected architectures. In partially scheduled architectures the scheduling of the operand paths is not trivial, and it can easily be seen that these paths can be scheduled better if we at least know where they lead to. In a cycle based scheduler, we have to schedule parts of the operand paths before we know where the consumers of the operands will be scheduled, so we can only guess where the operand paths must lead to.

Scheduling of operand paths is simpler and more efficient in an operation-based scheduler in which we can postpone the scheduling of the operand paths until the consumers have to be scheduled. The detection of deadlocks is however more difficult because we have to deal with more than one partially scheduled cycle. By making use of a two-dimensional space-time representation of the scheduling algorithm, we were fortunately able to design a deadlock test for this more complex case. Together with a shortest path based operand path router, we were able to create an operation-based scheduling algorithm which we expect to be reasonably good, and nearly always better than the proposed cycle-based algorithm.

Flow scheduling is a technique which is designed to improve the operation-based scheduling. It integrates the routing of the operand paths and the deadlock test, and it allows previously scheduled paths to be rerouted if that is beneficial. Flow scheduling is inspired on a number of advanced flow graph algorithms, and is expected to work even better.

We are busy implementing the algorithms in a scheduler and hope to confirm our expectations soon.

# Bibliography


[1] A. Baveja and A. Srinivasan. Approximation Algorithms for Disjoint Paths and Related Routing and Packing Problems. *Mathematics of Operations Research*, 25(2):255–280, May 2000.

[2] M. Charikar, C. Chekuri, T.Y. Cheung, Z. Dai, A. Goel, S. Guha, and M. Li. Approximation Algorithms for Directed Steiner Problems. *Journal of Algorithms*, 33:73–91, 1999.

[3] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms.* The MIT Press, 1990.

[4] U. Feige. A Threshold of $\ln n$ for Approximating Set-Cover. In *Proc. 28th Symp. on Theory of Computing*, pages 314–318, 1996.

[5] M. Franklin and G. S. Sohi. Register Traffic Analysis for Streamlining Inter-operation Communication in Fine-Grain Parallel Processors. In *25th Symp. on Microarchitecture*, pages 236–245, 1992.

[6] A.V. Goldberg, A.D. Oldham, S.A. Plotkin, and C. Stein. An Implementation of a Combinatorial Approximation Algorithm for Minimum-Cost Multicommodity Flow. In *Proc. 6th Conf. on Integer Programming and Combinatorial Optimization*, pages 338–352, June 1998.

[7] A.V. Goldberg and R.E. Tarjan. Solving Minimum-Cost Flow Problems by Successive Approximation. *Mathematics of Operations Research*, 15(3):430–466, 1990.

[8] M.D. Grigoriadis and L.G. Khachiyan. Approximate Minimum-Cost Multicommodity Flows in $\tilde{O}(\epsilon^{-2}knm)$ Time. *Mathematical Programming*, 75(3):477–482, December 1996.

[9] V. Guruswami, S. Khanna, R. Rajaraman, F.B. Shepherd, and M. Yannakakis. Near-Optimal Hardness Results and Approximation Algorithms for Edge-Disjoint Paths and Related Problems. In *ACM Symposium on Theory of Computing*, pages 19–28, 1999.

[10] J. Hoogerbrugge. *Code Generation for Transport Triggered Architectures.* PhD thesis, Delft University of Technology, February 1996.

[11] J. Hoogerbrugge and H. Corporaal. Register File Port Requirements of Transport Triggered Architectures. In *27th Symp. on Microarchitecture*, 1994.

[12] D. Karger and S.A. Plotkin. Adding Multiple Cost Constraints to Combinatorial Optimization Problems with Applications to Multicommodity Flows. In *Proc. 27th Symp. on Theory of Computing*, pages 18–25, May 1995.

[13] R. M. Karp. Reducibility among Combinatorial Problems. In R.E. Miller and J.W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[14] P. Klein, S.A. Plotkin, C. Stein, and É. Tardos. Faster Approximation Algorithms for the Unit Capacity Concurrent Flow Problem with Applications to Routing and Finding Sparse Cuts. *SIAM Journal on Computing*, 23(3):466–487, June 1994.

[15] J.M. Kleinberg. *Approximation Algorithms for Disjoint Path Problems.* PhD thesis, Massachusetts Institute of Technology, May 1996.

[16] S.G. Kolliopoulos and C. Stein. Approximating Disjoint-Path Problems Using Greedy Algorithms and Packing Integer Programs. In *IPCO*, pages 153–168, 1998.

[17] T. Leighton, F. Makedon, S. Plotkin, C. Stein, É. Tardos, and S. Tragoudas. Fast Approximation Algorithms for Multicommodity Flow Problems. *Journal of Computer and System Sciences*, 50(2):228–243, April 1995.

[18] S.A. Plotkin, D.B. Shmoys, and É. Tardos. Fast Approximation Algorithms for Fractional Packing and Covering Problems. *Mathematics of Operations Research*, 20(2):257–301, May 1995.

[19] S. Roos, R. Lamberts, and H. Corporaal. Remove: A Computer Architecture Designed for Modern VLSI Technology. In *Proc. 5th ASCI Conf.*, pages 421–428, June 1999.

[20] F. Shahrokhi and D.W. Matula. The Maximum Concurrent Flow Problem. *Journal of the ACM*, 37(2):318–334, April 1990.

[21] A. Srinivasan. Improved Approximations for Edge-Disjoint Paths, Unsplittable Flow and Related Routing Problems. In *Proc. 38th Symp. Foundations of Computer Science*, pages 416–425, 1997.

[22] C. Stein. *Approximation Algorithms for Multicommodity Flow and Shop Scheduling Problems.* PhD thesis, Massachusetts Institute of Technology, September 1992.

[23] P.M. Vaidya. Speeding up Linear Programming Using Fast Matrix Multiplication. In *Proc. 30th Symp. on Foundations of Computer Science*, pages 332–337, 1989.

[24] A. Zelikovsky. A Series of Approximation Algorithms for the Acyclic Directed Steiner Tree Problem. *Algorithmica*, 18:99–110, 1997.