Explicitly Staged Software Pipelining

Explicitly Staged Software Pipelining

By
Wolfgang Thaller

A Thesis
Submitted to the School of Graduate Studies
in Partial Fulfillment of the Requirements
for the Degree
Master of Science

McMaster University

MASTER OF SCIENCE(2006) McMaster University
COMPUTING AND SOFTWARE Hamilton, Ontario

TITLE: Explicitly Staged Software Pipelining

AUTHOR: Wolfgang Thaller

SUPERVISORS: Dr. Christopher Kumar Anand and Dr. Wolfram Kahl

NUMBER OF PAGES: xi, 59

Abstract

Software Pipelining is a method of instruction scheduling where loops are scheduled more efficiently by executing operations from more than one iteration of the loop in parallel. Finding an optimal software pipelined schedule is NP-complete, but many heuristic algorithms exist.

In iteration i, a software pipelined loop will execute, in parallel, "stage" 1 of iteration i, stage 2 of iteration $i-1$ and so on until stage k of iteration $i-k+1$.

We present a new approach to software pipelining based on using a heuristic algorithm to explicitly assign each operation to its stage before the actual scheduling.

This explicit assignment allows us to implement control flow mechanisms that are hard to implement with traditional methods of software pipelining, which do not give us direct control over what stages instructions are assigned to.

Acknowledgements

My sincere thanks go to my supervisors, Dr. Christopher Anand and Dr. Wolfram Kahl, for all the support and guidance they provided. Additional thanks go to Dr. Anand for a great canoeing and camping trip that helped me recover from thesis-induced exhaustion.

I would also like to thank my family and friends in Austria for supporting me from afar, and finally my friends here in Canada for making me feel at home here.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

With the advent of pipelined execution, instruction scheduling was born; two instructions that do not depend on each other can be executed in parallel, while dependences between instructions can force the processor to finish executing one instruction before starting to execute another. Among the many possible orderings of instructions that produce the same result, the goal is to find one that minimises (or comes close to minimising) the number of machine cycles required to execute the code.

Given a piece of straight-line code, that is code without any branches, there are two obvious lower bounds to the number of cycles required for an optimal schedule. One of them stems from the limited availability of CPU resources — the CPU can only execute a limited number of instructions at the same time.

The other lower bound is the sum of instruction latencies through the longest path in the data dependency graph.

In a loop, we can sometimes do better than just to rearrange the instructions within the loop body with respect to each other. If we can reach the resource-constrained lower bound, there is nothing we can do to improve the schedule any more, except, of course, for improvements in instruction selection or in the algorithm that is used, which falls outside the scope of this thesis.

If, however, our schedule is latency-constrained, there are some tricks that can be used to improve matters. The best known of these is loop unrolling or replication, where the loop body is replicated n times, such that one loop iteration in the final schedule does the work of n consecutive "logical iterations" in the original code. In addition to making the cost of the branch smaller in relation to the cost of the loop body, this frees up some opportunities to overlap instructions from different logical iterations.

However, replication is not actually necessary to allow overlapping of in-

structions from different logical iterations. Consider a loop with two operations that depend on each other, such as $(AB)^n$; this is equivalent to $A(BA)^{n-1}B$, with one important difference: in the transformed loop, the operations A and B operate on data from two different logical iterations and therefore may no longer depend on each other.

This work was developed as part of the COCONUT project, which prepares to produce a system that provides a coherent and consistent path from a mathematical specification of signal processing problems to verified *and* highly optimised machine code [ACK+04]. One of the hypotheses of this project is that more efficient implementations would be possible if the interface between the compiler front and back-ends were more flexible and extensible.

In this thesis, we present a new software pipelining scheduling method related to decomposed software pipelining (section 1.1.3). It consists of a heuristic algorithm to assign operations to different pipeline "stages" (chapter 4) and an accurate description of how a non-pipelined loop can be transformed to a pipelined loop once the staging is known (chapter 3).

In line with the idea of exposing flexible interfaces from the compiler backend, the representation of the input for the scheduler (chapter 2) allows us to specify inter-iteration dataflow in loops explicitly, even if that causes the loop to become unschedulable without software pipelining — in a software-pipelined loop, it is sometimes possible to use a value that is produced in a *future* iteration (see sections 2.5 and 3.5 for an example of this).

Taking advantage of our scheduling algorithm's properties, we introduce a limited form of control flow that can be scheduled especially well using our algorithm; a *multiloop* (chapter 6) is a loop that consists of a block of common code followed by a "switch" or "case" construct with an arbitrary number of different cases.

In chapter 5, we explore "Merge Scheduling", a novel approach to scheduling the output of our stage assignment heuristic, based on the idea of scheduling the stages separately and then "merging" them into one using a dynamic programming algorithm.

Finallly, we give experimental results for our algorithms for our target architecture of choice, the Cell Synergistic Processor Unit (see section 1.2 and [IBM06]).

The remainder of this chapter will give a very brief survey of other approaches to software pipelining and to the relevant aspects of the Cell architecture.

1.1 Software Pipelining

Software Pipelining [AJLA95] is the problem of finding a schedule for a loop while honoring both dependence and resource constraints, but without requiring that one iteration of the loop is finished before the next iteration starts.

The loop contains a set of operations op_i which are executed once in each iteration; we refer to the instance of operation op_i in iteration j as (op_i, j).

Resource constraints restrict which operations can be scheduled in the same cycle. Dependence constraints restrict the relative ordering of the operations and also may require a certain number of cycles (latency) to pass between two instructions.

There are three kinds of dependence constraints:

data dependence (read after write) Operation A calculates a value; operation B uses this value, so B has to be scheduled after A. Most instructions on most architectures take more than one machine cycle until the data is actually available (latency).

antidependence (write after read) Operation A uses a value that will be overwritten by operation B, so operation A has to be scheduled before operation B.

output dependence (write after write) Both operations A and B store their result in the same location, so the order of execution matters.

Dependences between two operations in the same iteration are called *loop independent*; dependences between different iterations of the loop are called *loop carried*.

In the *data dependency graph*, each node represents an operation op_i. Different instances (op_i, j) and (op_i, j) of that operation are represented by the same node. Edges in the dependency graph are labelled with the required latency (for data dependences) and with the difference in iterations (0 for loop independent dependences).

Let us consider the schedule for a completely unrolled loop; in conventional scheduling, all operations of iteration j will have been issued before the first operation of iteration $j + 1$ is issued. In a software-pipelined loop, this is not the case; iteration $j + 1$ will be *initiated* before iteration j completes.

We require the unrolled schedule to reach a repeating pattern after a short while (otherwise, the result would be code bloat proportional to the number of iterations). The number of cycles between the start of two consecutive (but overlapping) iterations is termed the *initiation interval*, or λ.

1.1.1 Kernel-based Methods

One approach to software pipelines is to completely unroll the loop, run a conventional scheduling algorithm on it and wait for a repeating pattern to emerge. The drawbacks to this approach are that a repeating pattern might not emerge under all circumstances, and that the resulting kernel might be longer than desired, *i.e.* contain multiple copies of each operation.

1.1.2 Modulo Scheduling

In modulo scheduling [RG81], the λ is chosen beforehand; then, instructions are scheduled into this limited space, backtracking as required. The problem has been shown to be NP-complete; various heuristics for cutting down the search space to a feasible size exist (*e.g.* [Lam88], [RGSL96]).

1.1.3 Decomposed Software Pipelining

Decomposed Software Pipelining [WEJS94; GS94; CDR98] approaches the software pipelining problem by decomposing it into two separate problems; first, the the problem is transformed into an acyclic scheduling problem by taking into account dependence constraints; second, resource constraints are taken into account by a classical scheduling algorithm, *e.g.* list scheduling.

1.1.4 Register Issues

If two instances of the same operation, (op_i, j) and $(op_i, j+1)$ write their output to the same location, this introduces loop carried antidependences from all consumers of (op_i, j) to $(op_i, j+1)$; the value has to be used before it is overwritten. This means that the longest lifetime the output of op_i can have is λ cycles.

Special hardware support for software pipelining has been proposed and implemented in the past [RYYT89; RST92]; in a *rotating register file*, several instances of a named register exists; a special register, the iteration control register (ICR) is used to select the instance to be used.

Unfortunately, rotating register files have not become commonplace; without rotating register files, we are faced with a choice: to accept the added antidependences and their consequences on the achievable λ, or to ignore them and simulate the effect of rotating register files in another way: *Modulo variable expansion* [Lam88] first replicates the loop body a few times; instances of

op_i from consecutive iterations have now become separate instructions in the schedule, and can be made to explicitly reference different registers.

1.2 The Cell Synergistic Processor Unit

Our current system targets IBM's and Sony's Cell Broadband Engine architecture [IBM06], more specifically, the "Synergistic Processor Units" of the Cell processor.

The Cell Broadband Engine Processor is a single-chip multiprocessor with nine processing units. One of these is the PowerPC-compatible "PowerPC Processor Unit" (PPU) which is intended to run the operating system and handle the more control-intensive tasks. The other processing units are eight identical "Synergistic Processing Units" (SPUs); these are processors optimised for high-speed computation tasks.

Each SPU has its own 256KB of memory, or local store, for instructions and data, and a large register file of 128 general purpose registers of 128 bits each. Data is transferred between the local stores and main memory using DMA transfers. The SPU instruction set is a SIMD (single instruction multiple data) instruction set; all instructions operate on 128-bit vectors.

Up to two instructions are issued *in-order* each cycle from two separate pipelines; every instruction can only execute in one of the pipelines, depending on its type.

This allows us to model the processor as having just two execution units; each instruction op requires a specific unit, $\mathsf{unit}(op)$. Two instructions op_1 and op_2 (whose dependence constraints have been satisfied) can be scheduled in the same cycle iff $\mathsf{unit}(op_1) \neq \mathsf{unit}(op_2)$.

Another noteworthy feature of the SPU architecture is the absence of condition registers; conditional branch instructions will simply compare a word in a general purpose register to zero. Of particular interest is also the "hint for branch" instruction `hbr` which can be used to explicitly inform the processor of the target a certain branch will jump to. When scheduled early enough in a loop, this eliminates all pipeline stalls due to branch misprediction, even when the branch in question is an indirect branch (branch to a computed address), as is required for a multiloop (see chapter 6).

Chapter 2

Representation

2.1 Codegraphs

In our system, a loop body before scheduling is represented by a "codegraph". This section summarises and adapts definitions from [KAC06] for our purposes.

A codegraph is a hypergraph with a sequence of input nodes and a sequence of output nodes. The hyperedges of the graph are labelled with machine instructions and their immediate arguments, *i.e.* any constants that are directly encoded in the opcode, but no source or target registers. Each hyperedge has zero or more ordered input tentacles and one or more ordered output tentacles, connected to nodes in the codegraph. Each node in the codegraph is labelled with a type.

Definition 2.1 *A* code graph $G = (\mathcal{N}, \mathcal{E}, \mathsf{In}, \mathsf{Out}, \mathsf{src}, \mathsf{trg}, \mathsf{nType}, \mathsf{eLab})$ *over an edge label set* ELab *and a set of types* NType *consists of*

- *a set* $\mathcal{N}$ *of* nodes *and a set* $\mathcal{E}$ *of* hyperedges *(or* edges*),*
- *two node sequences* $\mathsf{In}, \mathsf{Out} : \mathcal{N}^*$ *containing the* input nodes *and* output nodes *of the code graph,*
- *two functions* $\mathsf{src}, \mathsf{trg} : \mathcal{E} \to \mathcal{N}^*$ *assigning each hyperedge the sequence of its* source nodes *and* target nodes *respectively,*
- *a function* $\mathsf{nType} : \mathcal{N} \to \mathsf{NType}$ *assigning each node its* type*, and*
- *a function* $\mathsf{eLab} : \mathcal{E} \to \mathsf{ELab}$ *assigning each hyperedge its* edge label*, where the label has to be compatible with the numbers of source and target nodes of the edge, and with the types of those nodes.* □

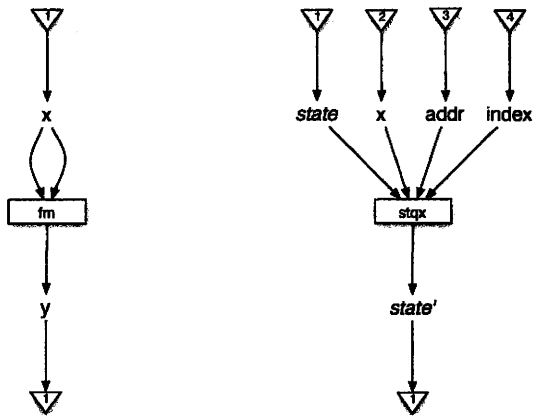

Figure 2.1: Simple codegraphs. Left: a codegraph that squares a vector of numbers using the SPU instruction `fm` (floating point multiply); right: a codegraph with the state-affecting store instruction `stqx`

Definition 2.2 *The function* consumers : $\mathcal{N} \to \mathbb{P}\,\mathcal{E}$ *maps a node to the set of all hyperedges in whose source list it appears (its* consumers*). Likewise, the function* producers : $\mathcal{N} \to \mathbb{P}\,\mathcal{E}$ *maps a node to the set of all hyperedges in whose target list it appears (its* producers*).* □

Definition 2.3 *A codegraph is called* executable *if*

- *It is acyclic.*
- *For every edge, an output node is reachable from at least one target node of the edge.*
- *Every node is either an input node or the target node of exactly one edge.* □

At the left of figure 2.1, you can see a very simple codegraph that squares one vector of four floating point values using the SPU instruction `fm`, floating point multiply. To make the figures easier to talk about, we will label each node in the codegraph with a unique name; remember, however, that nodes in the codegraph actually only have types, not names. The codegraph in the figure has two nodes, which we name x and y, and one edge, labeled with the SPU instruction. The sequence of input nodes contains only x, and y is the only

output node. The input and output sequences are visualised by the numbered triangles connected to the nodes in the figure. The triangles themselves are not nodes or edges of the graph.

The set of possible types consists of one type for each type of register available in the target architecture — in the case of the Cell SPU, there is just one type of register —, and the type *state*.

The state type exists to account for the fact that some machine instructions cannot be modelled as functions from input values to output values; examples include load, store, and branch hint instructions; we do not support actual branch instructions in the code graph. A value of type state is a token that represents all the state that can be affected by an instruction, including, but not limited to, memory (in our case, the Cell SPU's local store), and the state of the branch processor (which is affected by the hint for branch instruction).

A state-affecting instruction, like the store instruction **stqx** (store quadword indexed) shown on the right of figure 2.1, is then modelled as taking an additional parameter of type state, and returning a modified state. The figures use slanted text to distinguish nodes of type state, like ***state*** and ***state'***, from nodes of the register type, like **addr**.

Sometimes, we want to work with independent aspects of state, *e.g.* with different, non-overlapping areas of memory, without imposing a sequential ordering on the instructions that work with different state. In this case, we will just use two separate state tokens in the codegraph. Use of two separate state tokens is taken as an assertion that the operations on the two separate tokens are independent from each other; this assertion is not checked by our system.

To help describe transformations of codegraphs, we use a theory based on category theory, more specifically on gs-monoidal categories; this is described in detail in [KAC06]; for the purposes of this thesis, it is sufficient to summarise the algebra defined by that theory without requiring any further understanding of category theory.

Every codegraph G has a signature which consists of the sequence of the types of its input nodes and the sequence of the types of its output nodes; it is written as $G : I \to O$. Type sequences can be concatenated using the associative operator $\times$; the empty type sequence, denoted by $\mathbb{1}$, is both a right and left unit for $\times$.

For two codegraphs $G : A \to B$ and $H : B \to C$, the codegraph denoted $A;B : A \to C$ is their sequential composition; G's output nodes are identified with H's input nodes.

Two codegraphs $G : A \to B$ and $H : C \to D$ can also be composed in parallel using the $\otimes$ operator, producing $G \otimes H : A \times C \to B \times D$.

For a type sequence T, we can construct several basic codegraphs that do not contain any edges:

- $\mathbb{I}_T : T \to T$ is the identity codegraph over that type sequence; it contains no edges, and each of its nodes is both an input node and an output node in the corresponding position, with the types taken from T. The identity codegraph is both a right and left unit for sequential composition.
- $\nabla_T : T \to T \times T$ is a codegraph without operations that "duplicates" its input; the list of output nodes equals the list of input nodes concatenated with itself.
- As a generalisation of the above, $\nabla_T^n : T \to T \times T^n$ is a codegraph without operations that replicates its input n times; the list of output nodes equals the list of input nodes concatenated with itself n times.
- $!_T : T \to \mathbb{1}$ contains distinct input nodes, and nothing else. Informally speaking, it ignores its input and produces no output.

2.2 Loop Specifications

To specify a loop, rather than just a loop body, we need to specify what values are communicated between different iterations of the loop.

Definition 2.4 *A* loop specification *is a tuple* $(G, \mathbf{d})$ *containing*

- *an executable codegraph* $G : F \times K \to F \times C$
- *a sequence of integers* $\mathbf{d}$*, one for each element of* F*.* □

The input of the codegraph consists of two parts; for the left part, with types F, there are corresponding outputs and an associated integer d. These inputs and outputs represent the values that are communicated between different iterations of the loop. In any iteration i, the value of each of these inputs is equal to the value of the corresponding output in iteration $i + d$.

Hence, the usual case of using an output from the previous iteration is represented by $d = -1$. A value of d of less than -1 means that the input for the codegraph should be an output from further back in the past; this can, for example, be achieved by storing the value in an array so that it won't be overwritten by the next iteration.

A value of $d = 0$ means that the input should be equal to the output from the same iteration; alternatively, we could just use a single interior node in place of the input node and the output node.

An input/output pair with an associated d greater than 0 means that the input for iteration i should be the output of some *future* iteration $i + d$. This is, of course, impossible *if* the loop is scheduled in the conventional way, without software pipelining. In the presence of software pipelining, scheduling loop specifications with positive d values sometimes becomes possible, as we will see in chapter 3. For an example of how d values other than -1 might be used in a loop specification, refer to section 2.5.

In addition to the inputs with corresponding outputs (F), the codegraph G also has another group of inputs, of types K. These inputs are called *constant inputs* and represent all values that are passed to the loop from the outside but are not changed by the loop; these include unchanging parameters for the loop and constants that cannot be part of the opcode of a machine language instruction (and therefore included in the label of a hyperedge in the codegraph). In other words, the constant inputs are those values that are required to be available in a register when the loop starts and throughout the execution of the loop.

Finally, G also has a group of outputs of types C, called the *control outputs*. These control outputs are used to control when the loop should terminate; their meaning is described in more detail in section 2.4.

2.3 Maximum Lifetime and Loopability

We impose one additional restriction on the scheduler's output: Every operation in the codegraph must appear exactly once in the scheduled loop body, and every instruction in the scheduled loop body must appear in the codegraph. We do not want the scheduler to insert additional instructions, like extra register-to-register moves.

As a consequence, a node in the codegraph will be assigned to at most one register for its entire lifetime. The lifetime of one node in one iteration also cannot overlap with the lifetime of the same node in another iteration (that is being executed at the same time due to software pipelining); we make it the duty of the scheduler to limit the lifetimmes to less than λ, so that modulo variable expansion is never required.

Forbidding long lifetimes can have a negative impact on the achieved λ. However, as the codegraph does not explicitly specify locations for temporary values, modulo variable expansion becomes nothing but a fancy term for loop

unrolling; we can therefore achieve the effects of longer lifetimes and modulo variable expansion by replicating the loop body a few times *before* running the software pipeliner.

This has the advantage that we can always avoid having to use modulo variable expansion, which is crucial for scheduling multiloops (see chapter 6).

Not all loop specifications can be scheduled as loops without adding extra instructions. First, we consider the situation without software pipelining, *i.e.* when all operations scheduled in the loop body must operate on data belonging to the same iteration.

If an input node in the codegraph is also an output node, then they must be an input and an output in corresponding positions in F. The value of the node then has to be the same for all iterations; the input-output pair can therefore be converted to a constant input by removing the node from the output sequence of the codegraph and moving its position in the input sequence to the right to make it part of K rather than of F.

The case where a node is both an input and the corresponding output is easy to avoid by making it a constant instead; in other cases it is easy to explicitly specify an appropriate register-to-register move instruction. We therefore require the input codegraph to have no input node that is also an output.

If a value is calculated by an instruction in iteration i, the same instruction in iteration $i+1$ will overwrite it with a new value. Therefore, to achieve $d < -1$, an additional instruction has to be added to move the value to some other location before it gets destroyed.

Positive values for d are impossible: Using results from iteration $i+d, d > 0$ in iteration i requires iteration $i+d$ to start before iteration i has finished (software pipelining or reordering of iterations).

Definition 2.5 *A loop specification is* loopable *if*

- *All* d *are* -1 *or* 0.
- *No input is also an output.* □

An input-output pair with $d = 0$ is equivalent to identifying the input node with the output node (and removing them from the codegraph's input and output sequences). If this transformation succeeds without introducing cycles into the codegraph, we get a strictly loopable loop specification:

Definition 2.6 *A loop specification is* strictly loopable *if*

- *All d are -1.*
- *No input is also an output.* □

2.4 Loop Termination

Informally speaking, the control outputs (C) determine whether the loop should continue after the current iteration. The control outputs are connected directly to the branch instruction at the end of the loop; depending on what kind branch instruction is used for the loop, they can have different meanings.

For the Cell SPU target architecture, we currently use the following variants:

- The loop branch is a **brnz** (branch if not zero word) instruction; the loop continues if the first component of the only control output (which is, as are all values on the SPU, a vector) is non-zero.
- The loop branch is a **bi** (branch indirect) instruction; the first component of the only control output should contain the address of either the first instruction of the loop or of the first instruction after the loop.
- The loop branch is a **bi** instruction, as above; additionally, a second control output is a state token that is generated by a **hbr** (hint for branch) instruction.

2.5 Example

As a toy example, let us specify and schedule a loop that takes an array of floating point numbers as input and writes the square of every element to a separate output array. We can start with the codegraph from Figure 2.1, but to get a loop, we need to add instructions to load values from the input array, store values to the output array, update a loop counter, and calculate a loop condition as an output of the codegraph. The actual branch instruction will not be part of our codegraph (after all, we do not need an instruction scheduling algorithm to tell us that the branch instruction should occur right after the end of the loop body).

Figure 2.2 shows a loop specification for this loop; the **d** vector for the loop specification is visualised by adding dashed, labelled arcs from the output to the corresponding input in the codegraph. The instruction `lqd 0` loads a vector from the SPU's local store; its inputs are a state token and the address

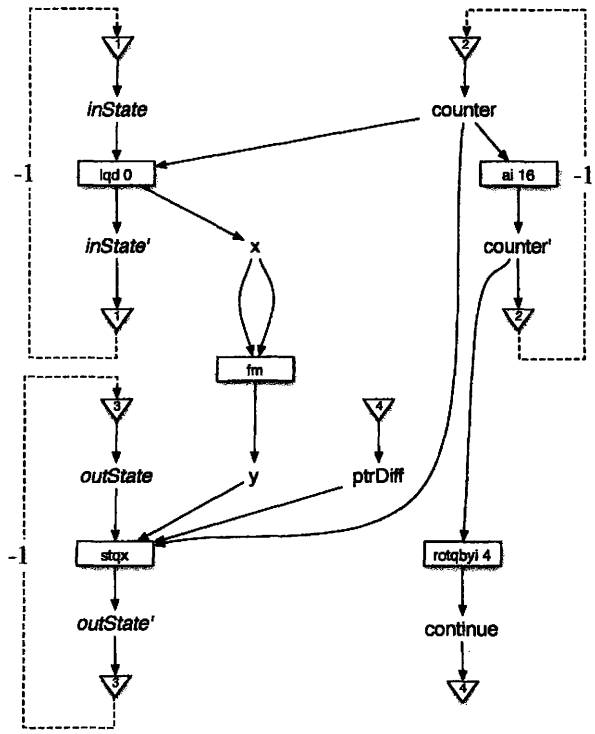

Figure 2.2: Loop that squares all elements in an array

of the vector to load, and its outputs are a state token and the vector that was loaded. For storing a vector, we use the `stqx` instruction; its inputs are a state token, the value to be stored, and two values which are added to yield the target address in the SPU's local store. We use separate state tokens for loading and storing, which are both passed on from one iteration to the next ($d = -1$).

The loop counter is incremented by 16 (the size in bytes of a vector value) in every iteration using the `ai 16` instruction and passed on to the next iteration ($d = -1$). We play a little SPU-specific game here: we take advantage of the fact that all values on the SPU are vectors, and the `ai` instruction separately affects all four 32-bit components of the register. We initialise the first component of the vector to the address of the input array (the `lqd` and `stqx` instructions always use the first component of their address arguments). The last component will contain $-16n$, where n is the number of

times the loop should be executed; it will reach zero after the nth iteration.

The `stqx` instruction stores the vector y at the address that results from adding the first components of the vectors ptrDiff and counter'. It consumes the state token *outState* and produces a new state token, *outState'*.

Finally, the instruction `rotqbyi 4` rotates the vector so that the last component is moved to the first component, where it is needed for the conditional branch.

The values counter and counter' will occupy the same register (otherwise, we would need to insert a register-to-register move instruction), so there is a data antidependence between the `stqx` and the `ai 16`. The `stqx` has to be scheduled before the `ai` because an input it needs (counter') is destroyed by `ai`.

We can now estimate a lower bound for the number of cycles this loop will take without software pipelining; we need to add up latencies, plus one cycle for the antidependence between `stqd` and `ai`:

$$6\ (\texttt{lqx}) + 6\ (\texttt{fm}) + 1\ (\text{antidependence}) + 2\ (\texttt{ai}) + 4\ (\texttt{rotqbyi}) = 19$$

Nineteen cycles is nothing to be proud of for a loop with just five instructions. The throughput can be increased by unrolling the loop or by applying software pipelining.

To make software pipelining this loop easier, we can also consider deriving the loop condition and the store address from the counter values for a different iteration, as shown in Figure 2.3. The value counter' is now mentioned twice in the output list of the codegraph; one occurence still corresponds to the counter input with $d = -1$, that is, counter in iteration i is the value computed by the `ai` instruction in iteration $i-1$. The other counter' output corresponds to a new input counter'', which, in iteration i, is the value computed by the `ai` instruction in iteration $i + x$.

This loop specification is *not* semantically equivalent to the earlier one; we need to adjust the ptrDiff and the initial value of counter according to the value of x.

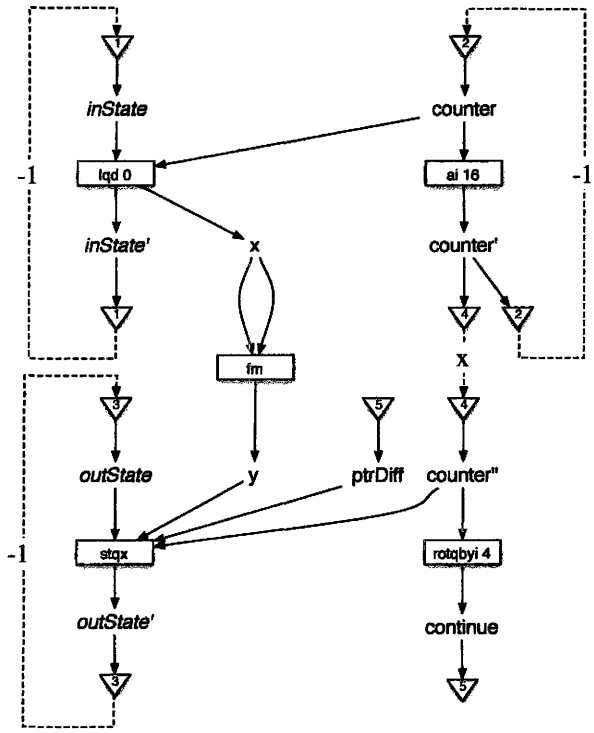

Figure 2.3: Improved Loop for Software Pipelining — use $x > 0$ to facilitate software pipelining

Chapter 3

Theory of Staging

Software pipelining can be viewed as a transformation of the loop specification. Informally speaking, we split up the codegraph into sequential parts, which we call stages, and compose them again in parallel to get a new loop body such that adding appropriate prologue and epilogue code to the loop yields a loop that is equivalent to the original loop.

For the purposes of this chapter, we assume a *legal* assignment to stages to be given; a stage assignment is legal when the transformation outlined in the next section can transform the loop specification $(G, \mathbf{d})$ to a loopable loop specification $(G', \mathbf{d}')$. Chapter 4 describes an algorithm for calculating a legal stage assignment for a loop specification.

Consider a loop that we can split into three stages, G_1, G_2 and G_3. In figure 3.1, $G_{i,j}$ denotes stage G_i operating on data for logical iteration j. If we schedule the loop without software pipelining (figure 3.1, left), the loop independent data dependences between the stages will prevent us from exploiting much parallelism. With software pipelining (figure 3.1, right), these dependences are between different iterations of the pipelined loop and therefore do not restrict parallelism between the stages.

Figure 3.2 shows how different values of d can be realised in a pipelined loop. When we assign operations to stages (chapter 4), we will need to assign stages in such a way that the resulting assignment is *legal*:

Definition 3.1 *A stage assignment with k stages for a given loop specification is considerd* legal *iff the following conditions are fulfilled:*

- *Every operation in the code graph is assigned to exactly one stage in the range* $1 \ldots k$.

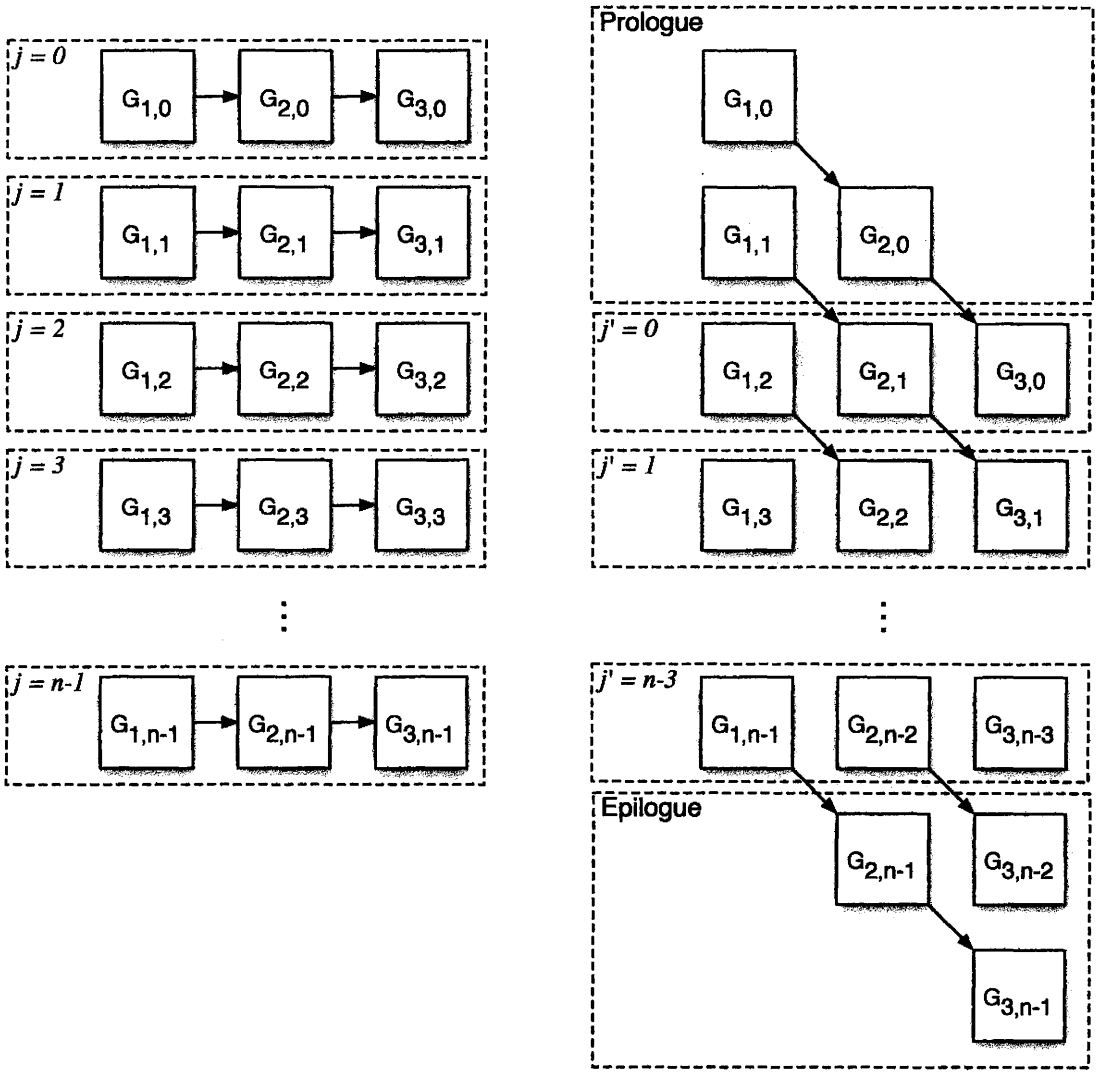

Figure 3.1: Software pipelining with three stages; left: loop independent data dependences in a non-pipelined loop restrict parallelism; right: the same dependences are now dependences between different iterations of the pipelined loop.

- *A value produced by an operation in stage i can be consumed by operations in stage i and/or in stage $i+1$.*

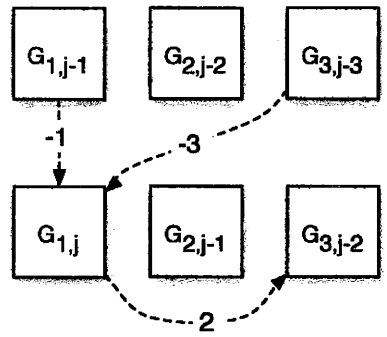

Figure 3.2: Inter-iteration dependences in a pipelined loop

- *For an input/output pair where the output value is produced by an operation in stage i, the input value can be consumed by operations in stages $i + d$ and/or in stage $i + d + 1$.* □

It is tempting to simply split up the codegraph G into k sequentially composed subgraphs, the stages G_i:

$$G = G_1 ; G_2 ; \ldots ; G_k$$

and then to recompose them in parallel:

$$G' = P ; (G_1 \otimes G_2 \otimes \ldots \otimes G_k),$$

where P is a permutation to make the argument order of the codegraph match up. We would then pick an appropriate $\mathbf{d}'$ to make a new loop specification.

This approach, however, gets us no closer to actually scheduling a software-pipelined loop, as we are potentially violating both conditions for being loopable. For one, we have no way of guaranteeing that $-1 \leq d' \leq 0$. Also, output values that are computed in one stage have to be passed through the later stages in the sequential composition, and inputs have to be passed through the stages before the one where they are consumed.

3.1 Pipelining Transformation

We want to do the sequential decomposition in a way that connects inputs consumed and outputs produced in a stage G_i directly to the inputs and

outputs of G as a whole, without routing them "through" the other stages.

The individual stages will be

$$G_i : M_{i-1} \times I_i \times K_i \to O_i \times C_i \times M_i$$

The I_i inputs of each stage are inputs from F, the K_i are constants from K, and M_{i-1} are outputs from the previous stage. Likewise, the O_i and C_k are connected to the output of G, while the M_i are fed into the next stage.

Due to the obvious lack of a stage before the first stage or a stage after the last stage, $M_0 = M_k = \mathbb{1}$. Furthermore, we want all control outputs to be generated in the same stage c, so we define $C_i = \mathbb{1}$ for all $i \neq c$, and $C_c = C$. The "control stage number" c becomes an additional parameter for the software pipelining and influences the meaning of the control outputs; see section 3.3 for details.

Because every output node has exactly one producer in G, every element of F will appear in the output list O_i of exactly one stage G_i; inputs, however, can appear in the I_i lists more than once.

We will also have to add a codegraph without any operations at the top to rearrange the I_i and K_i to match the order of $F \times K$, and to do the same at the bottom to make the order of the O_i match up with $F \times C$:

$$G = P \fatsemi (G_1 \otimes \mathbb{I}_{I_2 \times \ldots \times I_k}) \fatsemi (\mathbb{I}_{O_1} \otimes G_2 \otimes \mathbb{I}_{I_3 \times \ldots \times I_k}) \fatsemi \ldots \fatsemi (\mathbb{I}_{O_1 \times \ldots \times O_{k-1}} \otimes G_k) \fatsemi Q,$$

The codegraph $P : F \times K \to (I_1 \times K_1) \times \ldots \times (I_k \times K_k)$ has no hyperedges; it will route each input to the place or places (if any) where it appears in the input lists of the individual stages; constant inputs can appear on P's output list any number to times, other inputs up to twice. The codegraph $Q : O_1 \times \ldots \times O_k \to F \times C$ is a permutation, *i.e.* a codegraph without operations whose nodes each appear exactly once on its input sequence and exactly once on its output sequence. Figure 3.3 illustrates how three stages would fit together.

If the stages G_i have been chosen appropriately, we can construct a new, loopable, software pipelined loop specification $(G', \mathbf{d}')$ where the stages are executed in parallel, but for different "logical iterations", *i.e.* for different iterations of the original loop specification.

We have stated before that an input value from F can be used in more than one stage.

In every iteration of a software pipelined loop, the value of an input in F for two different iterations; if at the top of the pipelined loop, the value is available for some logical iteration j, then at some later point in the schedule,

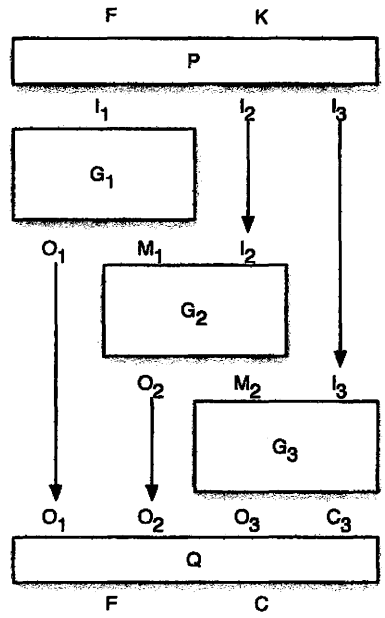

Figure 3.3: Splitting a codegraph into three stages

there will be an instruction that produces the value for iteration $j+1$, replacing the value for iteration j.

In the pipelined loop, stage i will operate on logical iteration $j+1$ while stage $i+1$ is still working on logical iteration j. Stage i can access the value of the input for logical iteration $j+1$ if the consuming instruction is scheduled below the producer; in the same iteration of the pipelined loop, stage $i+1$ can access the value for logical iteration j if the consuming instruction is scheduled above the producer. No other stages have access to a value of the input for the logical iteration they are working on.

Therefore, every input value from F can be used by at most two consecutive stages; the corresponding output value appears exactly once in the O_i list of exactly one stage, but it must appear twice in the output list of G'. One appearance will be associated with a $d' = -1$ entry in $\mathbf{d}'$, the other appearance with $d' = 0$.

Unused inputs are permissible in codegraphs, so we can simply do this for all outputs, even if they are used only once. We define

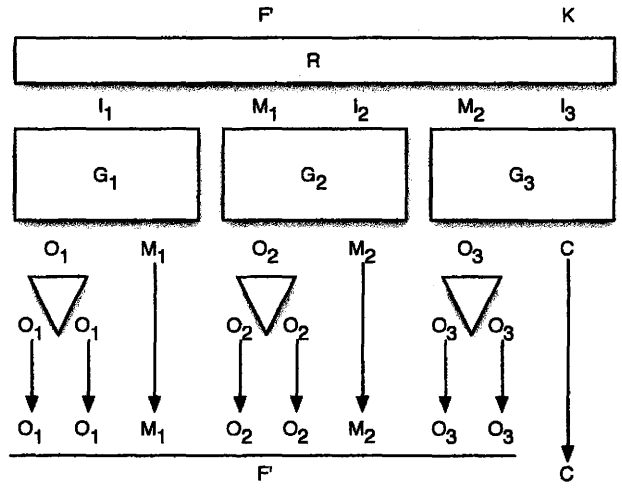

Figure 3.4: Putting the stages together in parallel

$$G_i' = G_i \wr (\nabla_{O_i} \otimes \mathbb{I}_{C_i} \otimes \mathbb{I}_{M_i})$$

The kernel $G' : F' \times K \to F' \times C$ of the software-pipelined loop will then be

$$G' = R \wr (G_1' \otimes \ldots \otimes G_k') \wr S,$$

where $F' = O_1 \times O_1 \times M_1 \times \ldots \times O_k \times O_k$, and $S : O_1 \times O_1 \times C_1 \times M_1 \times \ldots \times O_k \times O_k \times C_k \to F' \times C$ and $R : F' \to I_1 \times K_1 \times M_1 \times I_2 \times K_2 \times \ldots \times M_{k-1} \times I_k \times K_k$ is a codegraph without operations.

S is constructed such that it moves the C outputs to the end of the output list; if the control stage is the last stage ($c = k$), then $S = \mathbb{I}_{F' \times C}$. Figure 3.4) shows how these parts fit together for $k = c = 3$.

The entries in $\mathbf{d}'$ will be 0 for each first instance of O_i outputs, and -1 for the second instance of O_i outputs and for the M_i. We will choose R such that:

- Every M_i input is mapped to the corresponding M_i output; the corresponding d' value is -1.

- If P maps an input from K to an element of K_i, then R will map that same input from K to the same element of K_i (which will, in general,

not be in the same position in the output of R as it is in the output of P).

- The first occurence of each O_i input is mapped to the corresponding output in I_{i+d}, if such a corresponding output exists, where d is the value associated with O_i in the original loop specification.

- The second occurence of each O_i input is mapped to the corresponding output in I_{i+d+1}, if such a corresponding output exists.

3.2 Prologue and Epilogue

The software-pipelined loop specification is not a drop-in replacement for the original loop; after all, it requires intermediate results (M_i) as its input, which have to be supplied to the first iteration of the modified loop body. We do this by prefixing the modified loop with a prologue, a block of non-looped code that initialises the pipeline, as we see in the pipelined loop shown on the right of figure 3.1.

The first time the modified loop body executes, it will execute G_k for logical iteration 0, $G_k - 1$ for logical iteration 1, and G_1 for logical iteration $k-1$. The prefix code, therefore, has to execute $G_1 \ldots G_{k-1}$ for logical iteration 0, $G_1 \ldots G_{k-2}$ for logical iteration 1, and so on.

Likewise, after the last iteration of the software pipelined loop body, some logical iterations have been started but not finished. If the software pipelined loop body has been executed $n - k + 2$ times, the latest logical iteration that has been completed will be $n-k-3$; logical iterations $n-k+2$ through n will have been initiated but not completed. The epilogue code needs to execute G_k for iteration $n-k+2$, G_{k-1} and G_k for iteration $n-k+3$, and so on, and finally G_2 through G_k for iteration n.

Every iteration of the loop uses output values from earlier iterations as inputs, so when the loop starts at logical iteration 0, values have to be supplied from the outside, as there are no iterations with negative indices. If an input is associated with a d value of less than -1, multiple "initial" values for the input have to be supplied, corresponding to the outputs of the non-existant iterations d through -1. Inputs with nonnegative values for d, on the other hand, do not need any initial values at all.

Figure 3.5 shows how the loop gets initial values that are conceptually the outputs of negative iterations; only the values originating from stage 3 of iteration -3 and from stage 2 of iteration -1 are shown in the figure. Not

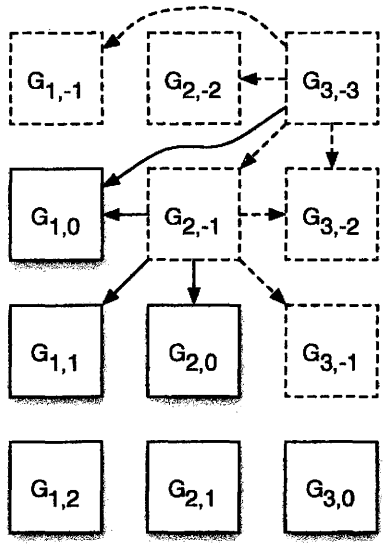

Figure 3.5: Prologue of loop with three stages. Initial values from $G_{3,-3}$ and $G_{2,-1}$ are shown as arrows.

all the outputs from those stages are required for loop initialisation; some are only required by other negative iterations (dashed arrows).

To construct a codegraph for the loop prologue we first define $\mathsf{init}(x)$ to be the primitive codegraph consisting of one hyperedge labeled with $\mathsf{init}x$, with no inputs and exactly one output. Using that, we define

$$\begin{aligned} G_{i,j} &= G_i && \text{if } j \geq 0 \\ G_{i,j} &= !_{M_{i-1} \times I_i \times K_i}(\mathsf{init}(i,j,1) \times \ldots \times \mathsf{init}(i,j,|I_i \times M_i \times C_i|)) && \text{if } j < 0 \end{aligned}$$

Here, i is the stage number and j is the logical iteration. For negative j, we do not want to generate any code; instead of the codegraph G_i for the stage, we construct a codegraph with the same input and output types that ignores all its inputs, and whose outputs are the results of hyperedges specially marked with the stage, iteration and position of the output.

We then continue with

$$G'_{i,j} = G_{i,j}\,;(\nabla_{O_i} \times \mathbb{I}_{C_i} \times \mathbb{I}_{M_i}),$$

so that we can define one slice of the prologue

$$Pr_j = R\,;(G'_{1,j} \otimes \ldots \otimes G'_{k,j-k+1})\,;S.$$

We resolve all $d' = -1$ by identifying the corresponding input and output nodes, and removing them from the input and output lists. Let the results of this transformation be $Pr'_j : F'' \times K \to F'' \times C$. We ignore the control outputs from the prologue:

$$Pr''_j = Pr'_j ; (\mathbb{I}_{F''} \otimes !_C)$$

Next, we compose $Pr''_{-1} \ldots Pr''_{k-2}$ sequentiallly, supplying a copy of the constant inputs to each:

$$Pr' = (\mathbb{I}_{F''} \otimes \nabla^k_K) ; (Pr''_{-1} \otimes \mathbb{I}_{K(k-1)}) ; (Pr''_0 \otimes \mathbb{I}_{K(k-2)}) ; \ldots ; Pr''_{k-2}$$

Then, we remove all unused nodes and edges from which no output is reachable from Pr', except for the constant inputs; this makes the codegraph executable again. Finally, we remove all init-labelled hyperedges and make their target nodes inputs of the prologue (the labels of the init hyperedges now define the meaning of those inputs).

Construction of the epilogue is symmetric to construction of the prologue.

3.3 Loop Termination Revisited

In a software pipelined loop, we have little choice but to interpret the control outputs based on the pipelined loop body; therefore, their meaning changes depending on the value of c, *i.e.* depending one which stage they are produced in.

If the control outputs evaluate to a value meaning "do not continue" in stage c of iteration n, then, the last iteration of the pipelined loop executes $G_{1,n+c-1}, \ldots, G_{c,n}, \ldots, G_{k,n+c-k}$. Including the loop epilogue, the last iteration to be executed will be iteration $n + c - 1$.

If we are trying to save on code size, we can do without an epilogue altogether in many cases; we might need to provide some extra space for partial results in any memory areas that the loop stores its results in. Without a prologue, if iteration n causes loop termination, the last iteration to be completed is iteration $n + c - k$; additionally, iterations $n + c - 1$ through $n + c - k + 1$ have been initiated, but not finished. It will depend on the particular stage assignment chosen for this loop whether any results for those partial iterations will be stored to memory or not.

3.4 Correctness

This thesis does not concern itself with a formal proof that the pipelining transformation and the prologue/epilogue construction yield a transformed loop that is equivalent to the original.

However, we can observe that every stage appears is executed the same number of times in the original loop and in the pipelined loop with prologue and epilogue. Also, the codegraph R (which governs how the inputs of the pipelined loop body are arranged) is constructed at the end of section 3.1 to always match up the corresponding inputs and outputs, and taking the appropriate d values into account. The transformation makes some assumptions about the assignment of operations to stages, which are all covered by the definition of a legal stage assignment given in the introduction to this chapter.

3.5 Example

Let us now return to our earlier example and have a look at how we can stage the loop specification from figure 2.3 on page 16; this is the variant where we can choose a parameter x that will tell the store instruction and the loop condition to use the loop counter from a different iteration. We will use this loop specification instead of the more straightforward one from figure 2.2 because it is much more amenable to software pipelining.

If we schedule the `stqx` two stages or more after the `lqd`, then the counter value will already have been updated in the meantime; the value for the current iteration will not be available any more. In its place, however, there is the counter value of a later iteration ready for use in the same register. Setting x to a value greater than -1 will therefore allow the load and store instructions to be farther apart.

With $x = 1$, the `stqx` and `rotqbyi` instructions that use counter" can be two or three stages after the `ai` instruction that produces them. If we decide on a total of three stages, this means that the `ai` instruction will be in the first stage, and the `stqx` and `rotqbyi` instructions will be in the last stage.

Figure 3.6 shows how the codegraph can be split up into three sequentially composed stages. In the example at hand, P ends up being an identity codegraph because the inputs used by stage 1 happen to be listed before the inputs used by stage 3 in the input sequence of the original codegraph.

The loop can then be pipelined by recomposing the stages in parallel, as shown in figure 3.7. The vertical positions of the operations indicate where

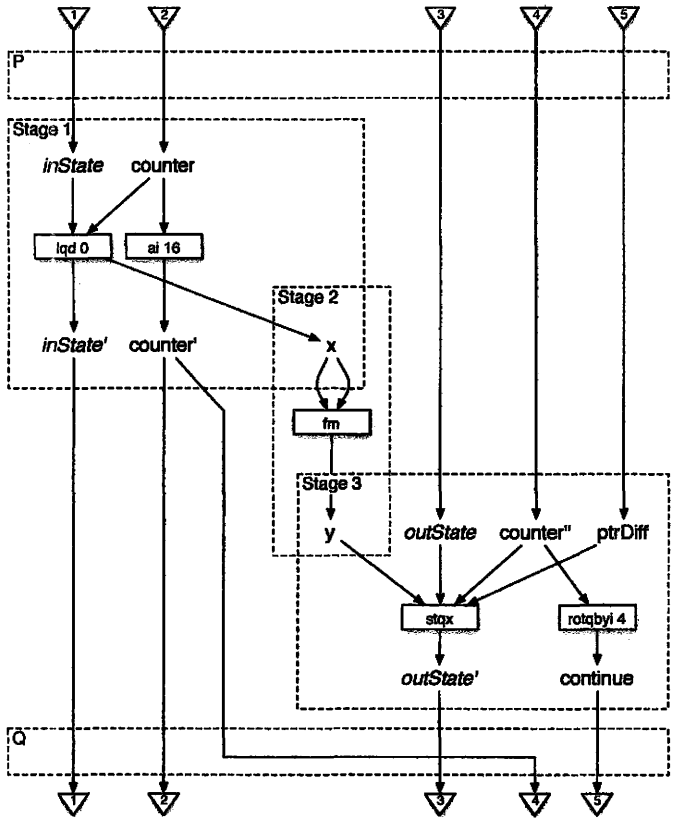

Figure 3.6: Splitting up the loop into three stages. $\mathbf{d} = (-1, -1, -1, 1)$

they occur in the final schedule. None of the inputs with $d' = 0$ (shown slightly above and to the left of the inputs with $d' = -1$) are actually used in this example; the nodes **a**, **b**, **c** and **d** are not used by any operations. Transforming the loop specification to a strictly loopable loop specification will therefore simply remove them and their corresponding outputs.

The scheduled loop body takes just 7 cycles to execute; this is just one cycle longer than the latency of the multiply and load instructions, and is an optimal schedule for this code.

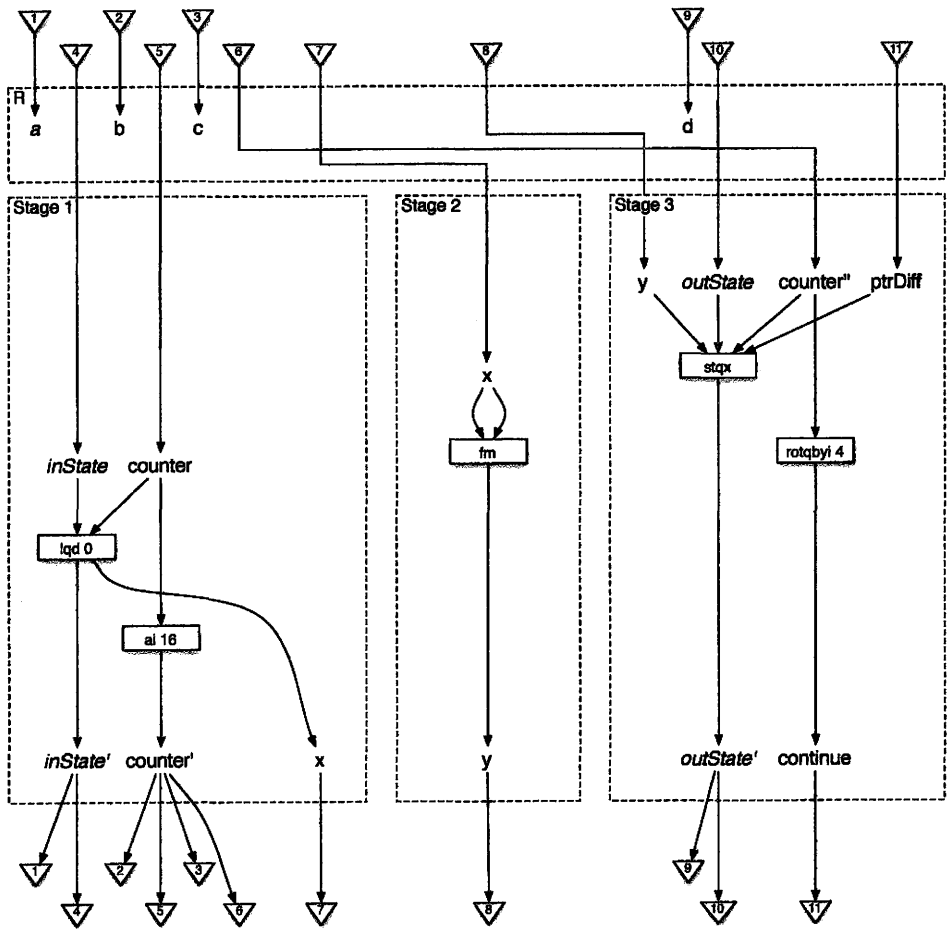

Figure 3.7: Pipelined loop, with three stages.
$\mathbf{d}' = (0, 0, 0, -1, -1, -1, -1, -1, 0, -1)$

Chapter 4

Heuristic Staging

We will now present a heuristic algorithm for splitting up a given loop specification into k stages. This means assigning a stage number in the range $1..k$ to each hyperedge in the given codegraph.

In addition to having to find a legal stage assignment, we want to *avoid* partitioning the codegraph in such a way that:

1. The latency along the longest path through one of the stages is greater than the initiation interval we expect to achieve.

2. Too many registers are alive accross stages.

   All registers that are alive accross stages will be live at the top of the final scheduled loop and will therefore conflict with each other. On the other hand, higher register requirements inside a stage can be mitigated by good decisions in the later steps.

3. Too many forward dependences between stages arise between stages.

   An inter-stage forward dependence corresponds to an input/output pair in the pipelined loop specification for which the associated $d' = 0$. This is generally undesirable, because it reduces parallelism between the involved stages; it is especially undesirable when we use the merge scheduling algorithm (chapter 5) which cannot handle forward dependences well.

The heuristic method we are going to use is based on using two functions, depth and height, that map every operation in the code graph to a nonnegative "depth" and "height" value.

We will use the depth and height values to decide approximately where to cut the codegraph into stages. More restrictions are placed on where to split

based on the dependences between the operations; finally, from the cuts that have not been eliminated earlier, we pick the one that minimises the number of values communicated from one stage to the next, *i.e.* we minimise register requirements at the top of the loop.

The algorithm proceeds as follows:

1. Initialise a constraints graph S (see section 4.2) that contains our current knowledge about the stage assignment.
2. For $i := k$ down to 2
   (a) Calculate **height**, **depth**, h_{tot}, and d_{tot} based on the codegraph with all operations known to be in stage $i + 1$ or greater removed (see section 4.1)
   (b) Try to mark all operations with **height** $> h_{tot}/i$ as being in stage $i - 1$ or less
   (c) Try to mark all operations with **depth** $> (i - 1)d_{tot}/i$ as being in stage i
   (d) Let A be the set of all opoerations known (according to S) to be in stage $i - 1$ or less, and let B be the set of all operations known to be in stage i or greater
   (e) Construct G_{cut} (see section 4.3)
   (f) Run a minimum cut algorithm on G_{cut}
   (g) Mark all operations in G_{cut} that are above the minimum cut as being in stage $i - 1$ or less
   (h) Mark all operations in G_{cut} that are below the minimum cut as being in stage i or greater
3. Extract final stage assignment from S.

4.1 Height and Depth

There are several choices for the **depth** and **height** functions. The simplest is to calculate depth and height based on the latencies of instructions:

Definition 4.1 *Given a codegraph G,*

- *For all* $n \in \mathcal{N}$, $\mathsf{depth}_L(n)$ *is the depth of the producer of* n *plus the producer's latency, or 0.*
- $\mathsf{height}_L(n)$ *is the maximum of the heights of all consumers of* n *plus their respective latencies, or 0.*
- *For all* $e \in \mathcal{E}$, $\mathsf{depth}_L(e)$ *is the maximum of the depths of all sources of* e.
- $\mathsf{height}_L(e)$ *is the maximum of the heights of all targets of* e. □

For an operation (a hyperedge) e in the codegraph, the function $\mathsf{depth}_L(e)$ gives a lower bound on the number of cycles that pass between the initiation of an iteration and when the instruction is issued for that iteration in a software pipelined schedule. Likewise, $\mathsf{height}_L(e)$ gives a lower bound for the number of cycles that pass between the completion of the instruction and the completion of the last instruction in the iteration.

Definition 4.2 *Given a codegraph* G,

- *For all* $e \in \mathcal{E}$, $\mathsf{height}_U(e)$ *is the number of hyperedges* e' *that are reachable in* G *from* e *and for which* $\mathsf{unit}(e') = u$ *when* u *is chosen such that* $\mathsf{height}_U(e)$ *becomes maximal.*
- $\mathsf{depth}_U(e)$ *is the number of hyperedges* e' *that are reachable in reverse direction from* e *and for which* $\mathsf{unit}(e') = u$ *when* u *is chosen such that* $\mathsf{depth}_U(e)$ *becomes maximal.* □

The lower bounds provided by $\mathsf{height}_U(e)$ and $\mathsf{depth}_U(e)$ stem from the fact that only one instruction per unit can be executed in one machine cycle; all edges reachable from e have to be executed after it, and all nodes reachable in reverse direction have to be executed before it.

To get the "best of both worlds", we therefore define

Definition 4.3 *For all* $e \in \mathcal{E}$, $\mathsf{height}_{LU}(e) = max(\mathsf{height}_L(e), \mathsf{height}_U(e))$ *and* $\mathsf{depth}_{LU}(e) = max(\mathsf{depth}_L(e), \mathsf{depth}_U(e))$. □

Definition 4.4 *Furthermore, we define a* total height h_{tot} *and a* total depth d_{tot}*:*

- $h_{tot} = max_{n \in \mathcal{N}} \mathsf{height}(n)$
- $d_{tot} = max_{n \in \mathcal{N}} \mathsf{depth}(n)$ □

4.2 Stage Constraints

Let s be the function that maps each operation to its stage number. During the execution of the algorithm, a directed graph S will represent our current knowledge about this function. Its nodes are the operations in G, plus an additional node z. We define $s(z) = 0$. An edge (x, y, d) (an edge from x to y, labeled with the integer d) is taken to denote the constraint

$$s(x) + d \geq s(y).$$

Due to the transitivity of $\geq$, the shortest path $SP(x, y)$ between two nodes can be used to derive new constraints.

$$\forall x, y : s(x) + SP(x, y) \geq s(y)$$

Specifically, we get:

$$\forall x : -SP(x, z) \leq s(x) \leq SP(z, x)$$

Initially, we populate the graph with edges representing the constraints we already know:

- Edges to and from z to enforce $\forall x : 0 \leq s(x) < k$
- For every edge (x, y, d) in the dependency graph, edges $(x, y, d+1)$ and $(y, x, -d)$
- Edges forcing all producers of control outputs to be in the last stage.

Then, we add some more constraints that are likely to yield a better stage assignment:

- Edges forcing the operations with the greatest height to be in the first stage.

  This is essential for making the stage assignment yield good results.

- For each non-constant input, edges forcing all its consumers to be in the same stage.

  This constraint serves to subtly discourage, but not entirely forbid inter-stage forward dependences ($d' = 0$ in the pipelined loop specification).

These constraints are not logically necessary, and as such can cause negative-weight cycles in S if they contradict other constraints. Therefore, if adding an edge for one of these constraints would create such a cycle, the constraint is simply ignored.

The algorithm then decides the boundaries between stages one by one, starting at the last stage. The number of the stage below the boundary to be decided shall be denoted by i.

For each boundary, the height and depth functions are used to classify each operation into three categories: above, below and undecided. The primary aim here is to exclude stage assignments that violate the second condition stated above, *i.e.* assignments where one of the stages, when scheduled individually, is longer than we would like our final loop body to be.

We calculate the height and depth functions based on a subgraph of the codegraph G; all nodes that are (according to S) known to be in stage $i+1$ or below are excluded. Thus, the total height and depth correspond to all the instructions from the first stage to stage i, inclusively.

Rather than pre-determining a target λ, we just strive to split up the codegraph into stages of roughly equal size, as measured by the height and depth functions. As the height function is intended to give a lower bound of the distance in cycles between an instruction and the "bottom" of the codegraph, we force all operations with a height greater than h_{tot}/i to be in stage $i-1$ or above; likewise, all instructions with depth greater than $(i-1)d_{tot}/i$ to be in stage i.

The graph S is augmented to reflect this new information by adding the appropriate edges to and from z. If adding such an edge would create a cycle with negative weight in S, then it is skipped. This means that previous decisions, or their consequences, contradict the new "recommendation" that was derived from the height and depth functions. Precedence has to be given to the earlier decisions.

We can now extract the set of all operations whose position with respect to the stage boundary under consideration is not yet known from the graph S. The exact boundary is then determined in a way that minimises the number of registers required (as described below). The results of that decision is then recorded in S again, and the process repeated for the next lower-numbered boundary, until all stages are decided.

4.3 Stage Separation

Now let us turn to the problem of determining where to draw the exact boundary between two stages. We have already narrowed down our choice so that condition 1 will always be fulfilled.

The goal now is to find a partitioning that fulfills the other two conditions, *i.e.* one that minimises register use without violating any of the constraints and without generating too many forward dependences. Of course, we must still make sure that our choices don't violate any other constraints defined in the previous section.

We can do this by transforming the remaining codegraph (after removing the nodes that are already known to be either above or below the boundary) to an instance of the minimum cut problem; the minimum cut will be the stage boundary that uses the smallest number of registers at the top of the loop.

We want to define a graph G_{cut} such that:

1. A minimum cut yields two stages with no illegal dependences.

2. The size of the minimum cut equals the number of values that have to be kept in registers from one stage to the next.

Definition 4.5 *Given a codegraph G, we first define a graph G'_{cut} containing*

- *A "source" node s*
- *A "sink" node t*
- *For every operation in the codegraph, an "operation" node*
- *For every node in the codegraph, a "value" node*
- *An edge with infinite weight from each consumer of a value to the producer*
- *An edge with weight* 1.0 *from each producer to the corresponding value node*
- *An edge with infinite weight from each value node to each of its consumers.* □

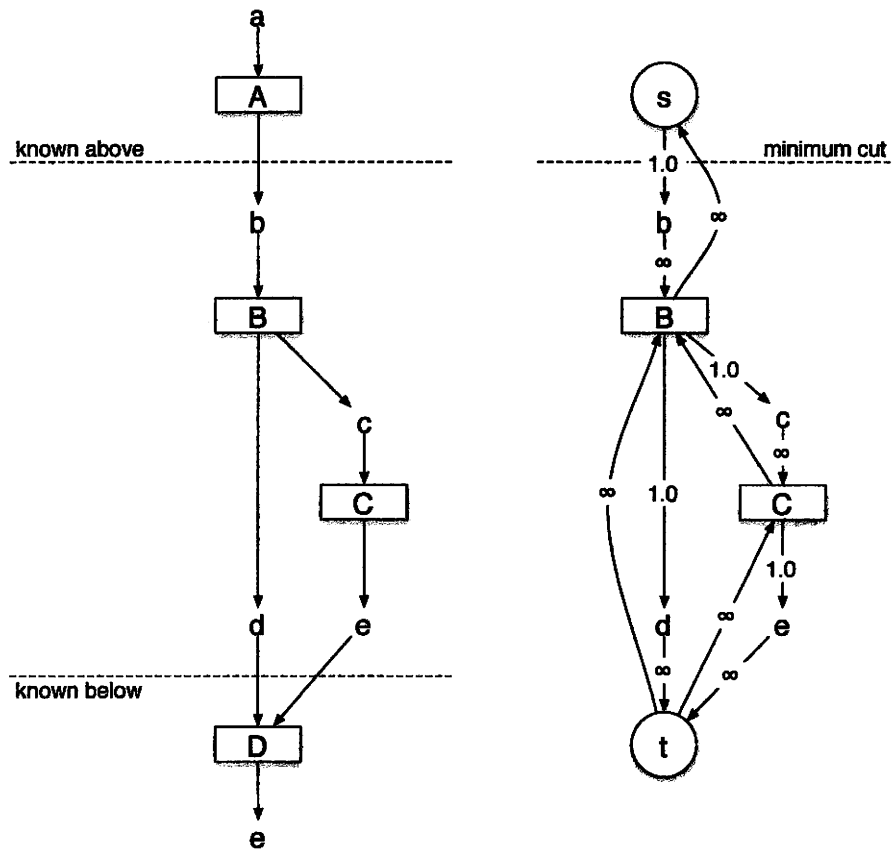

Figure 4.1: Transforming parts of a codegraph (left) to a minimum cut problem (right). Edges `A` and `D` are known to be above and below the stage boundary beforehand; using the minimum cut we decide to put `B` and `C` below the cut, because that requires the least number of registers to be live at the top of the stage.

Definition 4.6 *Based on G'_{cut}, a set A of operations (hyperedges in the codegraph) known to be above the boundary, i.e. in stage $i-1$ or less, and a set B of operations known to be below the boundary, i.e. in stage i or greater, we define a graph G_{cut} by*

- *Deleting all operation nodes in $A \cup B$*
- *Deleting all value nodes that are not connnected to a consumer or pro-*

ducer not in $A \cup B$

- *Deleting all edges that connect two deleted nodes*
- *Replacing all mention of nodes in A in the remaining edges with s*
- *Replacing all mention of nodes in B in the remaining edges with t* □

Figure 4.1 illustrates this transformation on a simple example.

If at least one of the consumers of a value is above the cut, then the producer must also be above the cut. This is easily achieved by having an edge with infinite weight point from each of the consumers to the producer; any cut where the producer is below the cut but one of the consumers is above would therefore have infinite weight, and therefore cannot be the minimum cut.

A value is live at the top of the loop if the operation that produces it is above the cut and at least one of the consumers is below the cut. For every value, we want an edge with weight 1.0 that crosses the cut if and only if the value is live at the top of the loop. We only want one such edge per value in order to avoid counting any live value twice. Therefore, this edge with weight 1.0 connects the producer node to the value node; edges with infinite weight are used to connect the value node to each of its consumers. Note that there is no edge going from the consumer to the value, so that it is possible for a value node to be below the cut while the operation node that consumes the value is above the cut.

We express the constraint that we want all consumers of an input to be in the same stage at this level by adding a cycle of edges with infinite weight between all successors of each input.

If a value has a producer or consumer whose stage is already known and a producer or consumer whose stage is not yet known, the transformation is done as above, but t is substituted for any operation that is known to be in stage i or greater, and s is substituted for any operation that is known to be in stage $i - 1$ or less.

Chapter 5

Merge Scheduling

Merge scheduling is based on the idea of "merging" two or more scheduled pieces of code to be executed in parallel. The individual stages we just created have relatively few inter-stage dependences, which works in our favour.

Analogous to the use of the term "merging" in the well-known merge sort algorithm, we do not rearrange the instructions in any one input schedule with respect to other instructions from the same input schedule. Merging can be done optimally in $O(n^{k+1})$ cycles, where k is the number of schedules to be merged and n is the number of instructions involved.

This naturally leads to the following plan for software-pipelining loops:

1. Heuristically assign instructions to stages
2. Schedule the individual stages using a conventional scheduling algorithm
3. "Merge" the individual linear schedules.

5.1 Separate Scheduling

After the stages have been decided, we need to come up with separate schedules for each of the stages. For this, we can essentially use any scheduling algorithm for straight-line code, as long as we enforce a few additional restrictions imposed by loop carried dependences.

After staging, loop carried and inter-stage dependences can be classified in two groups depending on the value of their associated component in $\mathbf{d}'$, and based on whether the producing and the consuming instruction are in the same stage or not.

- $d' = 0$, same stage

  This can only be caused by an input/output pair with $d = 0$ in the original loop specification. We can eliminate this either before or after staging by identifying the input and the output node in the codegraph and removing them from the input and output sequences.

- $d' = 0$, different stages

  An inter-stage forward data dependence constraint is passed on to the merging algorithm; no special treatment is required when scheduling the individual stages.

- $d' = -1$, same stage

  The producer will overwrite the value used by the consumer — this is a data antidependence which needs to be taken into account when generating the schedule for the stage.

- $d' = -1$, different stages

  The inter-stage antidependence is passed on to the merging algorithm.

Also, in the $d' = -1$ case, there is a forward dependence between the producer in one iteration of the pipelined loop and the consumer in the following iteration. First, let us observe that this can be ignored completely; violating these dependences does not affect correctness, but it will introduce a pipeline stall once for every iteration of the loop.

Adding the appropriate number of empty cycles to the end of each of the individual schedules is a definite win if not all schedules require the same amount of padding; the merging algorithm will automatically try to schedule the empty cycles in parallel with instructions from other stages.

Deciding the schedule for one stage amounts to arbitrarily adding edges to the dependency graph for operations that do not depend on each other but happen to appear in a certain order in the schedule. When scheduling a stage, decisions that have already been made while scheduling other stages have to be respected.

Assume we have two stages; 1 contains operations op_1 and op_2, and stage 2 contains op_3 and op_4. Further assume that, due to inter-stage dependences, op_3 has to appear before op_1 and op_2 has to appear before op_4 in the final schedule (solid arrows in figure 5.1). While scheduling stage 1, we decide that op_1 will appear before op_2 in the schedule (dashed arrow in the figure). If we schedule stage 2 without taking that decision into account, we might

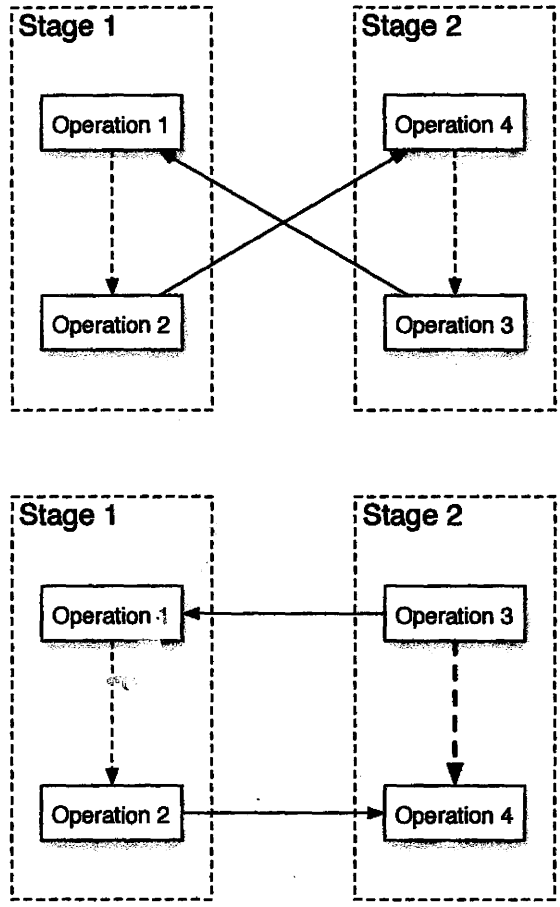

Figure 5.1: Avoiding dependence cycles when scheduling seperately

decide to schedule op_4 before op_3, thereby creating a cycle in the dependency graph which prevents merge scheduling from succeeding (figure 5.1, top).

Therefore, when scheduling stage 2, we have to take the fact into account that after scheduling stage 1, there is a path from op_3 to op_4 in the dependence graph, and therefore op_3 has to be scheduled before op_4, as shown at the bottom of figure 5.1.

To summarise, the following steps have to be performed to build the per-stage schedules:

1. Generate the pipelined loop specification $(G', \mathbf{d}')$ (see chapter 3).

2. Let the dependency graph D_0 be a graph consisting of:
   - For every hyperedge in G', a node.
   - For every node in G', edges from the producer to each consumer, labelled with the producer's latency.
   - For every input/output pair with $d' = -1$, edges from each consumer to the producer (antidependence), labelled with 0.
   - For every input/output pair with $d' = 0$, edges from the producer to each consumer (forward dependence), labelled with the producer's latency.
3. For each stage i from 1 to k,
   (a) Build a dependence graph for the operations in the stage by taking the subgraph of the transitive closure of D_{i-1} consisting of only the operations in stage i.
   (b) Run the list scheduling algorithm (or another straight-line scheduling algorithm) on this graph.
   (c) Let D_i be the graph that results from adding edges (x, y) with label 0 to D_{i-1} for every pair of nodes x and y where x has been scheduled before y.
4. For each stage i from 1 to k,
   (a) For each input/output pair in G' with an associated $d' = -1$ where the producing operation is in stage i, check the latency from the producer to the consumers.
   (b) Add the minimum number of empty cycles at the end of the schedule for stage i required to satisfy the latency constraints.

5.2 Merging

At this point, we have k non-software-pipelined schedules for the individual stages, and a set of dependences between those stages.

The dependences are now neatly divided into two groups:

- data antidependences

  These indicate that a consuming instruction in one stage has to be scheduled before a producing instruction in another stage, which will overwrite the data with a new value; this kind of dependence arises very often.

- forward dependences

  A consuming instruction in one stage requires data produced by a producing instruction from another stage; the producer has to be scheduled after the consumer in the resulting schedule, while taking account of latency. Forward dependences are rare because our staging algorithm tries to avoid them in most situations.

5.2.1 Merging without dependences

To explore the basics of merging schedules, let us disregard all inter-stage dependences for a while.

The input consists of schedules S_i for each the k stages; we denote the length in cycles of each schedule as n_i. Every cycle contains zero or more instructions which can be issued in that cycle. All pipeline stalls are explicitly represented by empty cycles in the schedule.

The instructions in each cycle j use a subset $U_{i,j}$ of the functional units U of the processor.

In the process of merging, two instructions from the same stage are never reordered; more specifically,

- If two instructions are in the same cycle of one stage schedule, they will be in the same cycle in the final schedule.

- If two instructions are in cycles c_1 and c_2, $c_2 - c_1 = d > 0$ in one stage, then they will be in cycles c_1' and c_2' in the merged schedule, such that $c_1' \geq c_1$ and $c_2' - c_1' \geq d$.

At each cycle of the merged schedule, the merging algorithm can schedule a combination of the first unscheduled cycles from each of the stage schedules.

After picking which instructions to issue in the first cycle, the remaining schedule can be found by recursively solving the subproblem of merging the k schedules with the already-scheduled instructions removed, so that $n_i - 1 \leq n_i' \leq n_i$. All the subproblems that arise this way are obviously independent from each other, and there are only $\prod_{i=1}^{k} n_i$ such subproblems; the problem can therefore be solved using the technique known as dynamic programming, where subproblems of increasing size are calculated by a loop, storing the results that might still be needed in an array.

This leads us to the following algorithm: We define $A[x_1, \ldots, x_k]$ to be the minimum number of cycles required to merge the first x_i cycles of each of the k stages. Trivially, we define $A[0, \ldots, 0] = 0$.

To determine the value of $A[x_1, \ldots, x_k]$, we need to examine all combinations $C \in \mathbb{P}\,1, \ldots, k$ of stages such that the instructions S_{i,x_i} for $i \in C$ do not cause a resource conflict, *i.e.* $\forall\, i, j \in C, i \neq j : U_{i,x_i} \cap U_{j,x_j} = \varnothing$.

Let $b_i = 1$ if $i \in C$ and $b_i = 0$ otherwise. We pick C such that $A[x_1 - b_1, \ldots, x_k - b_k]$ is minimal; then, $A[x_1, \ldots, x_k] = 1 + A[x_1 - b_1, \ldots, x_k - b_k]$. We also define an array L and define $L[x_1, \ldots, x_n] = C$; at the end, the schedule can be constructed from L by going backwards starting at $L[n_1, \ldots, n_k]$.

One of the main concerns with software pipelining is the increased register pressure. Therefore, it is worthwhile to extend the algorithm to choose the schedule with the minimum number of simultaneously live values among all the possible schedules with minimum number of cycles.

To do so, we need every instruction in the non-software-pipelined schedules to be annotated with the difference $D_{i,j}$ between the number of values that are live beginning at that instructions and the number of values that die at that instruction.

For this algorithm, we define R_{max} such that $R_{max}[j, x_1, \ldots, x_k]$ is the minimum number of simultaneously live values required when merging the first x_i cycles of each of the k states into a merged schedule with a length of j cycles; we further define $R[j, x_1, \ldots, x_k]$ to be the number of simultaneously live values *after* cycle j in that optimal merged schedule. As initial values, $R[0, 0, \ldots, 0] = R_{max}[0, 0, \ldots, 0] = r_0$, the number of live values at the top of the loop, and $R[0, x_i, \ldots, x_k] = R_{max}[0, x_i, \ldots, x_k] = \infty$ (if at least one $x_k \neq 0$).

We again examine all combinations C that do not cause resource conflicts; this time, we pick C such that $R_{max}[j - 1, x_1 - b_1, \ldots, x_k - b_k]$ is minimal, and define $R[j, x_1, \ldots, x_k] = R[j - 1, x_1 - b_1, \ldots, x_k - b_k] + \sum_{i \in C} D_{i,x_i}$, and $R_{max}[j, x_1, \ldots, x_k] = max(R_{max}[j - 1, x_1 - b_1, \ldots, x_k - b_k], R[j - 1, x_1 - b_1, \ldots, x_k - b_k])$.

5.2.2 Merging with antidependences only

The dynamic programming algorithm can easily be extended to accomodate antidependences. An antidependence tells us that a certain instruction in one stage must be scheduled before another instruction in another stage, because the latter instruction overwrites and destroys an input for the former.

When considering whether to schedule the instructions at position x_i from stage i in a cycle j, we know exactly how many instructions from each of the stages have already been scheduled before cycle j. If the dependence is violated, the instructions from that stage are simply not eligible to be scheduled at this point.

5.2.3 Handling forward dependences

A forward dependence between two instructions means that one instruction from one stage must be scheduled *at least a certain number of cycles* after another instruction from another stage, because the latter calculates data used by the former.

Unfortunately, handling latencies at this point violates one of the fundamental requirements of the dynamic programming approach. For dynamic programming to work, all the subproblems have to be independent from one another; in the presence of latencies greater than one, we cannot tell whether a particular schedule for the cycles up to j is optimal without knowing what code follows.

We can approximate the solution in the presence of forward dependences by counting only the cycles within one stage. To satisfy a forward dependence that requires the instructions from cycle j_1 of stage a to be scheduled at least l cycles after the instructions from cycle j_2 of stage b, we instead require the instructions from cycle $j_2 + l$ of stage b to be scheduled before the instructions from cycle j_1 of stage a.

Chapter 6

Adding Control Flow — The Multiloop

Control flow in software-pipelined loops has been the subject of much research. In the context of software pipelining, it is always necessary to impose some kind of limitation on control flow, or to accept the fact that the presence of control flow might severly limit the efficacy of software pipelining.

In many cases, it is beneficial to avoid control flow altogether and instead use some kind of conditional "select" instructions to select one result after calculating both alternatives unconditionally.

One possible approach to implementing limited forms of control flow is to have several versions of the entire software-pipelined loop body and to jump to the appropriate version of the loop body after every iteration.

This approach becomes very interesting when we have direct control over which operation is scheduled in which stage. Given a loop of the form

```
for i=0..n
    A(i);
    B(i);
    c = C(i);
    case c of
        1 -> D1(i);
        2 -> D2(i);
        ...
        n -> Dn(i);
    end case
end for
```

we can restrict all the D_n operations to the last stage; we can then software-

pipeline the loop, resulting in the following structure:

```
A(0);
A(1); B(0);
A(2); B(1); c = C(0);
for i=0..n-3
    case c of
        1 -> A(i+3); B(i+2); c = C(i+1); D1(i);
        ...
        n -> A(i+3); B(i+2); c = C(i+1); Dn(i);
    end case
end for
```

In fact, the branch instruction at the end of the loop will be a computed branch that will jump to the appropriate version of the loop body for the next iteration. A branch hint instruction scheduled earlier can make this jump almost free of cost.

Let us now extend the definition of a loop specification from chapter 3 for multiloops:

Definition 6.1 *A* multiloop specification *with n cases is a tuple $(G, n, \mathbf{d})$ containing*

- *a codegraph $G : F \times K \to F^n \times C$*
- *a positive integer n*
- *a sequence of integers $\mathbf{d}$, one for each element of F.* □

The control outputs C are interpreted to specify which of the n blocks of F outputs are to be used as the output of this iteration; only instructions that the selected set of outputs or the control outputs depend on are to be executed for this iteration of the loop.

Definition 6.2 *We define* control dependent *nodes and edges in a multiloop specification inductively:*

- *A node that occurs in the output sequence of G, but not at exactly the same position(s) for all cases is control dependent.*
- *If a node is control dependent, then its producer is control dependent.*

- *If a node is control dependent, then all its consumers are control dependent.*
- *If an edge is control dependent, then all its targets are control dependent.* □

Control dependent edges are those operations which have to be different for different cases of the multiloop, *after register allocation*; if one edge in G delivers its result to the ith output of one case, and to the jth ($i \neq j$) output of another case, then it is control dependent, because the assembly language instructions in the final schedule will have to differ by their target register.

When a multiloop is software pipelined, all control dependent edges have to be in the last stage. For every case, a separate version of the entire loop body is scheduled. The individual versions contain the same instructions for stages 1 to $k-1$, they only differ in the last stage.

The edges that generate the control outputs C have to be in the second-to-last stage, so that the branch instruction at the end of the loop can select which version of the final stage (and therefore, which version of the scheduled loop body) to execute.

In the context of multiloops, the decision to disallow the scheduler from duplicating any instructions from section 2.3 pays off. For a simple loop, modulo variable expansion is always an option; in a multiloop, our scheme for handling control flow would break down if we allowed the loop body to be replicated.

Chapter 7

Experimental Results

We have implemented a prototype scheduler based on the algorithms described in this thesis; concurrently, several efforts have been underway in our group to develop code and other tools target the Cell SPU and the COCONUT declarative assembly language [ACK+04], a textual representation of codegraphs. In the following, we give some results on how our algorithms perform on some of that code.

7.1 Simple Loops

We evaluate the performance of the algorithm for regular loops on a library of basic mathematical functions for the Cell SPU that has been developed at our work group.

Each of the basic mathematical functions takes one or two arguments and returns one or two results; the majority take one argument and return one result. They operate on vectors of four floating point values in parallel. We have run the scheduler on loops that sequentially read input values from an input area in memory and store the output values to a different memory area.

We can apply two different preprocessing steps to the code before scheduling it; we can unroll the loop by replicating the loop body a u times in parallel (the parallel instances of the loop body will use common loop counting instructions).

We can also apply a preprocessing step that splits up the m nodes with the greatest lifetimes (as estimated *before* scheduling using the height_{LU} function from section 4.1) by inserting register-to-register move instructions (we use the `rotqbyi 0` instruction, rotate quad word by zero bytes).

Tables 7.1 and 7.2 show the result of applying the heuristic staging

	$u=1$					$u=2$				
	s	m	b	n	r	s	m	b	n	r
acos	6	2	28	43	38	6	4	55	59	58
acosh	5	2	23	32	37	5	4	45	45	49
asin	6	2	28	54	39	4	4	55	77	44
asinh	5	2	21	35	36	5	4	41	42	46
atan2	2	2	24	103	32	2	4	47	103	43
atanh	7	2	20	21	40	3	0	39	42	41
cbrt	3	2	21	26	39	3	4	41	44	41
cos	4	2	26	29	41	4	4	51	51	53
cosh	6	2	17	28	36	6	4	33	34	44
div	2	2	13	24	13	2	4	17	20	17
exp	4	0	13	23	29	4	4	23	25	36
log10	5	2	15	23	35	6	4	25	25	48
log	5	2	15	23	35	6	4	25	25	48
pow	5	2	23	41	45	4	4	45	46	58
qdrt	5	2	13	25	17	4	4	25	40	23
rcbrt	5	2	24	32	40	5	4	47	49	56
rec	4	2	11	15	13	3	4	14	19	14
rqdrt	4	2	12	21	17	4	4	19	26	24
rsqrt	4	2	11	19	15	2	4	14	23	18
sin	4	2	26	30	43	3	4	51	51	50
sincos	5	2	29	32	50	3	0	57	57	51
sinh	6	2	17	28	36	6	4	33	34	44
sqrt	4	2	11	23	15	3	4	14	20	18
tan	5	2	33	40	45	2	4	65	65	47
tanh	8	2	19	25	37	6	4	37	37	46

Table 7.1: Results of scheduling math functions in simple loops for unrolling factors 1 (no unrolling) and 2

algorithm followed by a simple list-scheduling algorithm to schedule the transformed loop body; for each function and each value of the unrolling factor u, the table shows the number of stages s and the number of inserted moves m that yielded the shortest schedule; for this shortest schedule, it shows the

	$u = 3$					$u = 4$				
	s	m	b	n	r	s	m	b	n	r
acos	5	0	82	87	59	3	0	109	112	69
acosh	3	6	67	69	51	2	0	89	91	56
asin	3	6	82	91	61	3	8	109	110	76
asinh	3	6	61	61	51	3	8	81	81	62
atan2	2	6	70	105	52	2	8	93	107	60
atanh	3	0	58	59	51	3	8	77	77	61
cbrt	2	6	61	61	51	2	8	81	81	60
cos	3	6	76	79	58	3	8	101	102	70
cosh	6	6	49	49	56	4	0	65	65	61
div	2	6	21	23	21	2	0	23	24	22
exp	4	6	34	34	44	4	8	45	45	52
log10	3	6	37	37	42	3	8	49	49	48
log	3	6	37	37	42	3	8	49	49	48
pow	3	6	67	67	63	3	8	89	90	74
qdrt	4	6	37	50	30	2	8	49	59	34
rcbrt	4	6	70	70	65	4	8	93	93	78
rec	2	6	17	19	17	2	8	20	21	20
rqdrt	3	6	28	37	28	3	8	37	44	34
rsqrt	2	6	17	23	22	2	0	21	24	25
sin	3	0	76	76	59	3	8	101	101	72
sincos	3	6	85	85	63	2	8	113	113	66
sinh	6	6	49	49	56	4	0	65	65	61
sqrt	3	6	19	37	22	2	8	25	37	26
tan	2	0	97	97	57	2	8	129	129	63
tanh	5	6	55	55	54	6	8	73	73	74

Table 7.2: Results of scheduling math functions in simple loops for unrolling factors 3 and 4

theoretical lower bound b for the schedule length calculated from the number of instructions for each of the Cell SPU's two execution units, the achieved schedule length (in cycles) n and the number of registers r used by the schedule.

As was to be expected, we can see that we are more likely to find a

	$u=1$						$u=2$					
	s	m	b	n	n_m	Δ	s	m	b	n	n_m	Δ
acos	3	2	28	54	54	0	3	4	55	63	64	1
acosh	3	2	23	44	44	0	3	4	45	50	51	1
asin	3	2	28	77	78	1	3	4	55	82	93	11
asinh	3	2	21	51	54	3	3	4	41	50	54	4
atanh	3	0	20	41	42	1	3	0	39	42	44	2
cbrt	3	2	21	26	31	5	3	4	41	44	44	0
cos	2	2	26	39	41	2	3	0	51	52	55	3
cosh	3	2	17	43	45	2	3	4	33	43	48	5
div	3	2	13	24	32	8	2	4	17	20	29	9
exp	2	2	15	31	31	0	3	4	23	31	39	8
log10	3	2	15	29	27	-2	3	4	25	26	30	4
log	3	2	15	29	27	-2	3	4	25	26	30	4
pow	3	2	23	48	53	5	3	4	45	51	57	6
qdrt	3	2	13	34	36	2	3	4	25	43	43	0
rcbrt	3	2	24	39	38	-1	3	4	47	55	55	0
rec	3	2	11	18	25	7	3	4	14	19	32	13
rqdrt	3	2	12	36	38	2	3	4	19	37	36	-1
rsqrt	3	2	11	21	22	1	2	4	14	23	24	1
sin	3	2	26	36	50	14	3	4	51	51	57	6
sincos	3	2	29	38	47	9	3	0	57	57	63	6
sinh	3	2	17	43	45	2	3	4	33	43	48	5
sqrt	3	2	11	24	26	2	3	4	14	20	28	8
tan	3	2	33	51	53	2	2	4	65	65	69	4
tanh	3	2	19	52	52	0	3	4	37	54	54	0

Table 7.3: Merge Scheduling vs. List Scheduling, unrolling factors 1 and 2

schedule close to the minimum bound at higher unrolling factors; we have good schedules for $u = 1$ for a few of the functions (atanh, cos, sin, sincos); at $u = 2$, we have reason to be happy with most of the schedules, and for $u = 4$, almost all functions schedule within a few cycles of the theoretical lower bound b.

Merge scheduling, while having polynomial complexity in the size of its input, is exponential in the number of stages; For the math functions

	$u = 3$						$u = 4$					
	s	m	b	n	n_m	Δ	s	m	b	n	n_m	Δ
acos	2	6	82	90	87	-3	3	0	109	112	116	4
acosh	3	6	67	69	69	0	2	0	89	91	94	3
asin	3	6	82	91	100	9	3	8	109	110	119	9
asinh	3	6	61	61	65	4	3	8	81	81	85	4
atanh	3	0	58	59	59	0	3	8	77	77	77	0
cbrt	2	6	61	61	64	3	2	8	81	81	83	2
cos	3	6	76	79	81	2	3	8	101	102	106	4
cosh	3	0	49	57	53	-4	2	8	65	66	69	3
div	3	6	21	23	29	6	2	0	23	24	34	10
exp	3	6	34	40	43	3	3	8	45	48	47	-1
log10	3	6	37	37	43	6	3	8	49	49	55	6
log	3	6	37	37	43	6	3	8	49	49	55	6
pow	3	6	67	67	71	4	3	8	89	90	90	0
qdrt	3	6	37	53	50	-3	2	8	49	59	65	6
rcbrt	2	6	70	76	74	-2	2	8	93	95	95	0
rec	2	6	17	19	21	2	3	8	20	21	33	12
rqdrt	3	6	28	37	48	11	3	8	37	44	53	9
rsqrt	2	6	17	23	24	1	2	0	21	24	27	3
sin	3	0	76	76	80	4	3	8	101	101	104	3
sincos	3	6	85	85	88	3	3	8	113	113	115	2
sinh	3	0	49	57	53	-4	2	8	65	66	69	3
sqrt	3	6	19	37	39	2	2	8	25	37	40	3
tan	2	0	97	97	100	3	3	0	129	129	129	0
tanh	3	6	55	59	58	-1	3	0	73	75	76	1

Table 7.4: Merge Scheduling vs. List Scheduling, unrolling factors 3 and 4

(especially for $u > 1$), this means that the execution time required for merge scheduling becomes impractical when using more than three stages.

Tables 7.3 and 7.4 show how merge scheduling stacks up against plain list scheduling after the pipelining transformation. This time, we limit the maximum number of stages to use to three, in order to be able to compare both algorithms at equal values of s; keep in mind, however, that by using list

scheduling directly, we can use higher values for s and thereby achieve better schedules in many cases.

In addition to the columns we have seen in the earlier tables, these tables show the schedule length n_m achieved by merge scheduling and the difference $\Delta = n_m - n$ between the schedule lengths achieved by merge scheduling and list scheduling. The tables do not show register usage.

The difference Δ varies from -4 to 14; in most cases, the simpler algorithm, list scheduling, is also the better one. In those cases where merge scheduling delivers a better schedule, it does so by only a few cycles; it is conceivable that minor adjustments to the priority function used in list scheduling might allow list scheduling to catch up there, too.

If we take into account the high time complexity of merge scheduling, we unfortunately have to conclude that merge scheduling is not suitable for practical use.

7.2 Multiloop

We have developed one real-world algorithm that exploits the strengths of the multiloop to the fullest. The algorithm was initially developed by Dr. Anand and not only serves as a test case for the multiloop scheduler, but also served as the original inspiration for the concept of a multiloop.

As part of a non-uniform fourier transform, we need to resample non-uniform samples in three-dimensional space to a regular grid; more specifically, we have samples along a trajectory through three-dimensional space, and we need to convolve these samples with a gaussian kernel. It boils down to having to add something to the entries near the position of the sample in the three-dimensional grid, for every sample on the trajectory.

The distance between two successive samples is small, so that many of the same grid positions will be affected by two consecutive samples; in fact, it is guaranteed to be less than the size of a grid cell per dimension. We therefore have a window of interest that moves by at most one grid cell in every dimension for each sample in the input.

At 128 registers, the Cell SPU's register file is large enough to accomodate a decently-sized window of $4 \times 4 \times 4$ complex values, stored in 32 vector registers.

The loop will, at each step, add the kernel to the current window (which is in registers) and then, depending on the trajectory, move the window by storing "old" values, shifting all values in the window, and loading "new" values. The $3^3 = 27$ different possible directions to move the window (including

the case of not moving at all) are each implemented as a case of the multiloop.

Things are further complicated by the fact that the Cell SPU only supports vector-aligned loads and stores; we therefore need to round up the size of the register window in the unaligned case; we keep the "unused" parts of the unaligned window in 4×4 additional registers. Adding the convolution kernel to the register window works the same for both aligned and unaligned windows; the extra registers are not modified. Moving the window, however, needs to be done differently for the aligned and for the unaligned case. This doubles the total number of different cases to 54.

Tables 7.5 and 7.6 show the results of scheduling the multiloop; for each of the 54 cases, the table shows the lower bound b, the length n of the schedule achieved by heuristic staging followed by list scheduling, the number r of registers used, the length n_m of the schedule achieved by heuristic staging followed by merge scheduling, and the difference $n_m - n$ between the results of the two scheduling strategies.

Merge scheduling is again outperformed on most cases, and it even happens to deliver one of its worst results on case 53, which will be executed most often, as it is the case where the register window does not move.

On the other hand, we see that by software pipelining the multiloop using our heuristic staging algorithm, we can achieve close-to-optimal schedules for most of the 54 cases of our code; we are confident that this level of performance cannot be reached without using a software-pipelined multiloop.

#	b	n	r	n_m	$n_m - n$
0	151	151	110	151	0
1	151	156	92	159	3
2	151	152	110	151	-1
3	151	155	92	159	4
4	87	98	87	98	0
5	143	143	110	143	0
6	151	151	110	151	0
7	151	156	91	159	3
8	151	154	110	151	-3
9	151	156	92	159	3
10	87	104	88	103	-1
11	143	145	110	143	-2
12	139	141	108	139	-2
13	139	144	92	150	6
14	139	139	108	139	0
15	139	144	91	150	6
16	81	91	85	100	9
17	107	117	104	117	0
18	151	151	110	151	0
19	151	156	92	159	3
20	151	151	110	151	0
21	151	156	92	159	3
22	87	103	88	103	0
23	143	143	110	143	0
24	151	151	109	151	0
25	151	155	91	159	4
26	151	151	109	151	0

Table 7.5: Scheduling results for the 3D resampling multiloop, cases 0 through 26

#	b	n	r	n_m	$n_m - n$
27	151	155	92	159	4
28	87	101	87	102	1
29	143	143	110	143	0
30	139	139	107	139	0
31	139	144	90	150	6
32	139	139	107	139	0
33	139	144	91	150	6
34	81	91	83	100	9
35	107	117	102	117	0
36	139	139	109	139	0
37	139	144	92	151	7
38	139	139	108	139	0
39	139	144	91	151	7
40	80	97	88	101	4
41	107	115	110	115	0
42	139	139	108	139	0
43	139	144	92	151	7
44	139	139	107	139	0
45	139	144	92	151	7
46	80	99	87	99	0
47	107	120	108	114	-6
48	123	125	111	130	5
49	123	127	93	141	14
50	123	123	111	130	7
51	123	127	93	141	14
52	80	86	86	99	13
53	79	86	86	99	13

Table 7.6: Scheduling results for the 3D resampling multiloop, cases 27 through 53

Chapter 8

Conclusions & Outlook

We have developed a new method of software pipelining that gives explicit control about which stages operations are assigned to.

We have defined a way of describing loops that allows specifying arbitrary inter-iteration dataflow, allowing the user or earlier stages in the compilation process to take better advantage of software pipelining. We have formalised software pipelining as a transformation between such loop specifications.

Controlling stage assignment explicitly and avoiding the need for modulo variable expansion after scheduling allowed us to handle a special but very useful case of control flow, the *multiloop* very efficiently. Even though several design decisions were made specifically to support the multiloop, the algorithm also performs well on a selection of simple loops.

As a future research direction, it should be investigated whether the heuristics of decomposed software pipelining [WEJS94; GS94; CDR98], especially the circuit-retiming based approach given in [CDR98], can be adapted to our framework. The algorithm of [CDR98] has a known efficiency bound, but it does not currently have any provisions for pre-assigning stages (required for the multiloop). Also, it does not limit the lifetime of values to λ, but instead relies on modulo variable expansion to handle these long lifetimes, which again precludes its use for scheduling multiloops.

In [KAC06], the concept of *joins* is discussed; the idea is to allow a node in the codegraph to have multiple producers; the scheduler is then expected to choose one alternative, based on which leads to the better code. This would allow us to specify a loop like the one from figure 2.3 (section 2.5) without having to decide on a value for the iteration distance x before scheduling; instead, the loop specification would contain a join that tells the scheduler to choose among several inputs with different d values.

Bibliography

[ACK+04] Christopher Kumar Anand, Jacques Carette, Wolfram Kahl, Cale Gibbard, and Ryan Lortie. Declarative assembler. SQRL Report 20, Software Quality Research Laboratory, McMaster University, October 2004. available from `http://sqrl.mcmaster.ca/sqrl_reports.html`.

[AJLA95] Vicki H. Allan, Reese B. Jones, Randall M. Lee, and Stephen J. Allan. Software pipelining. *ACM Computing Surveys*, 27(3):367–432, 1995.

[CDR98] P.-Y. Calland, A. Darte, and Y. Robert. Circuit retiming applied to decomposed software pipelining. *IEEE Transactions on Parallel and Distributed Systems*, 9(1):24–35, 1998.

[GS94] Franco Gasperoni and Uwe Schwiegelshohn. Generating close to optimum loop schedules on parallel processors. *Parallel Processing Letters*, 4:391–403, 1994.

[IBM06] IBM Corp, Sony Inc. and Toshiba Corp. *Cell Broadband Engine Programming Handbook*, 2006.

[KAC06] Wolfram Kahl, Christopher Kumar Anand, and Jacques Carette. Control-flow semantics for assembly-level data-flow graphs. In Wendy McCaull, Michael Winter, and Ivo Düntsch, editors, *8th Intl. Seminar on Relational Methods in Computer Science, RelMiCS 8, Feb. 2005*, volume 3929 of *LNCS*, pages 147–160. Springer-Verlag, 2006.

[Lam88] M. Lam. Software pipelining: an effective scheduling technique for VLIW machines. *SIGPLAN Not.*, 23(7):318–328, 1988.

[RG81] B. R. Rau and C. D. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance

scientific computing. In *MICRO 14: Proceedings of the 14th annual workshop on Microprogramming*, pages 183–198, Piscataway, NJ, USA, 1981. IEEE Press.

[RGSL96] John Ruttenberg, G. R. Gao, A. Stoutchinin, and W. Lichtenstein. Software pipelining showdown: optimal vs. heuristic methods in a production compiler. In *PLDI '96: Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation*, pages 1–11, New York, NY, USA, 1996. ACM Press.

[RST92] B. Ramakrishna Rau, Michael S. Schlansker, and P. P. Tirumalai. Code generation schema for modulo scheduled loops. *SIGMICRO Newsl.*, 23(1-2):158–169, 1992.

[RYYT89] B.R. Rau, D.W.L. Yen, W. Yen, and R.A. Towle. The Cydra 5 departmental supercomputer: design philosophies, decisions and trade-offs. In *System Sciences, 1989. Vol.I: Architecture Track, Proceedings of the Twenty-Second Annual Hawaii International Conference on*, volume 1, pages 202–213, Kailua-Kona, HI, USA, 1989.

[WEJS94] Jian Wang, Christine Eisenbeis, Martin Jourdan, and Bogong Su. Decomposed software pipelining: a new perspective and a new approach. *Int. J. Parallel Program.*, 22(3):351–373, 1994.