Explicitly Staged Software Pipelining

Christopher Kumar Anand ANANDC@MCMASTER.CA
Wolfram Kahl KAHL@CAS.MCMASTER.CA
Wolfgang Thaller WOLFGANG.THALLER@GMX.NET
McMaster University, Computing and Software, ITB-201
1280 Main Street West, Hamilton, ON L8S 4K1 Canada

Abstract

This paper describes an alternative to modulo scheduling for loops, in which the first step is to divide instructions into stages by solving a series of min-cut problems constructed from the code graph of the unscheduled loop body. Our algorithm is formulated and implemented in terms of the code graphs of our own "declarative assembly language" for CELL SPUs. We have measured an average 20% reduction in Iteration Interval between our algorithm and modulo scheduling via XLC.

1. Introduction

There are many techniques for software pipelining in use and under research, including unrolling, kernel recognition, modulo scheduling [1, 2, 3], and decomposed software pipelining [4, 5, 6]; see [7] for an overview.

Our approach differs from most approaches by defining and working with an enriched data dependency graph, called *loop specification*, in which relationships between multiple logical iterations are expressible. This simplifies some concepts (modulo variable renaming), and facilitates a different trajectory through the scheduling problem. Rather than assigning cycles, or partial orderings to instructions, we first assign them to stages. We call our algorithm *Explicitly-Staged Software Pipelining* (ExSSP) in contrast to modulo scheduling where stage assignment is not explicit.

The first fully implemented instance of our algorithm targets the Cell SPU [8]. We have tested the scheduler output for correctness, and verified that its self-reported performance agrees with the timing tool shipped with the CELL SDK 1.1. On a library of math functions, our self-reported timing is on average 20 percent faster than XLC.

2. Code Graphs

In our system, a loop body before scheduling is represented by a "code graph". This section summarises and adapts definitions from [9] for our purposes.

A code graph is a hypergraph with a sequence of input nodes and a sequence of output nodes. Each node in the code graph is labelled with a type. 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 (connected to nodes representing the arguments consumed by the instruction) and one or more ordered output tentacles (connected to nodes representing the results of the instruction).

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.* □

For the purposes of the current paper, we only consider code graphs that correspond to straight-line code; for motivation for more general code graph concepts and for more details see [10, 9].

Definition 2.2 *A code graph is called* executable *if it is acyclic and*

- *for every edge, an output node is reachable from at least one target node of the edge, and*
- *every node is either an input node, or the target node of exactly one edge.* □

At the left of figure 1, you can see a very simple code graph 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 code graph with a unique name; remember, however, that nodes in the code graph actually only have types, not names. The code graph in the figure has two nodes, which we name *x* and *y*, and one edge, labelled 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*.

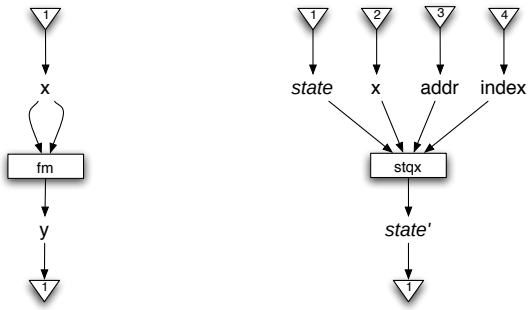

Figure 1: Simple code graphs. Left: a code graph that squares a vector of numbers using the SPU instruction `fm` (floating point multiply); right: a code graph with the state-affecting store instruction `stqx`

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 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 code graph. 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 currently not checked by our system.

To help describe transformations of code graphs, we use a theory based on category theory, more specifically on gs-monoidal categories; this is described in detail in [9]. For the purposes of this paper, it is sufficient to summarise the algebra defined by that theory without requiring any further understanding of category theory.

Every code graph 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 code graphs $G : A \to B$ and $H : B \to C$, the code graph denoted $A;B : A \to C$ is their sequential composition; G's output nodes are identified with H's input nodes.

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

Code graphs without any edges and where all nodes are inputs and/or outputs are called *wiring graphs*; the following wiring graphs, parameterised with a type sequence T, will be used later:

- $\mathbb{I}_T : T \to T$ is the identity code graph 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 code graph is both a right and left unit for sequential composition.

- $\nabla_T : T \to T \times T$ is a code graph without operations that "duplicates" its input; the list of output nodes equals the list of input nodes concatenated with itself.

3. Loop Specifications

In a code graph representing a loop body, variables modified by this loop body are represented both as inputs and outputs of that code graph, and the input instances feed from the output instances of the previous iteration.

In a staged loop body, the situation can be more complex, since inputs of one stage may be connected to outputs from different stages, which will then belong to different "logical" iterations of the unstaged loop body.

To properly formalise the relating issues, we need to specify what values are communicated between different iterations of the loop:

Definition 3.1 *A* loop specification *is a tuple* $(F, K, C, G, \mathbf{d})$*, consisting of type sequences* F*,* K*,* C*, an executable code graph* $G : F \times K \to F \times C$*, and a sequence of integers* $\mathbf{d}$*, called* iteration distances*, one for each element of* F*.*

In this set-up, the type sequence F identifies the state modified by the loop body, and forms part of both the code graph inputs and outputs. In addition, there is an iteration distance d associated with each of these "variables" (via the sequence $\mathbf{d}$), indicating that in any iteration i, the value of the input 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 code graph 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; semantically this is equivalent to using a single interior node in place of the input node and the output node, but the separated view is the better starting point for splitting a loop specification into stages.

In addition to the inputs with corresponding outputs (F), the code graph 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. 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 3.2.

3.1 Maximum Lifetime and Loopability

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

As a consequence, a node in the code graph 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 lifetimes to less than λ, so that modulo variable expansion [2] is never required.

Forbidding long lifetimes can have a negative impact on the achieved λ. However, as the code graph 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.

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 code graph 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 code graph 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 code graph 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 $i + d$ with $d > 0$ in iteration i requires iteration $i + d$ to start before iteration i has finished (software pipelining or reordering of iterations).

Definition 3.2 *A loop specification is* loopable *if no input node is also an output, and all d are -1 or 0.*

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

Definition 3.3 *A loop specification is* strictly loopable *if no input node is also an output, and all d are -1.*

3.2 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.

4. Theory of Staging

Software pipelining can be viewed as a transformation of the loop specification. Informally speaking, we split up the code graph 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 section, 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 $(F, K, C, G, \mathbf{d})$ to a loopable loop specification $(F', K, C, G', \mathbf{d}')$. We will describe below, in section 5, how to obtain a legal stage assignment for a loop specification.

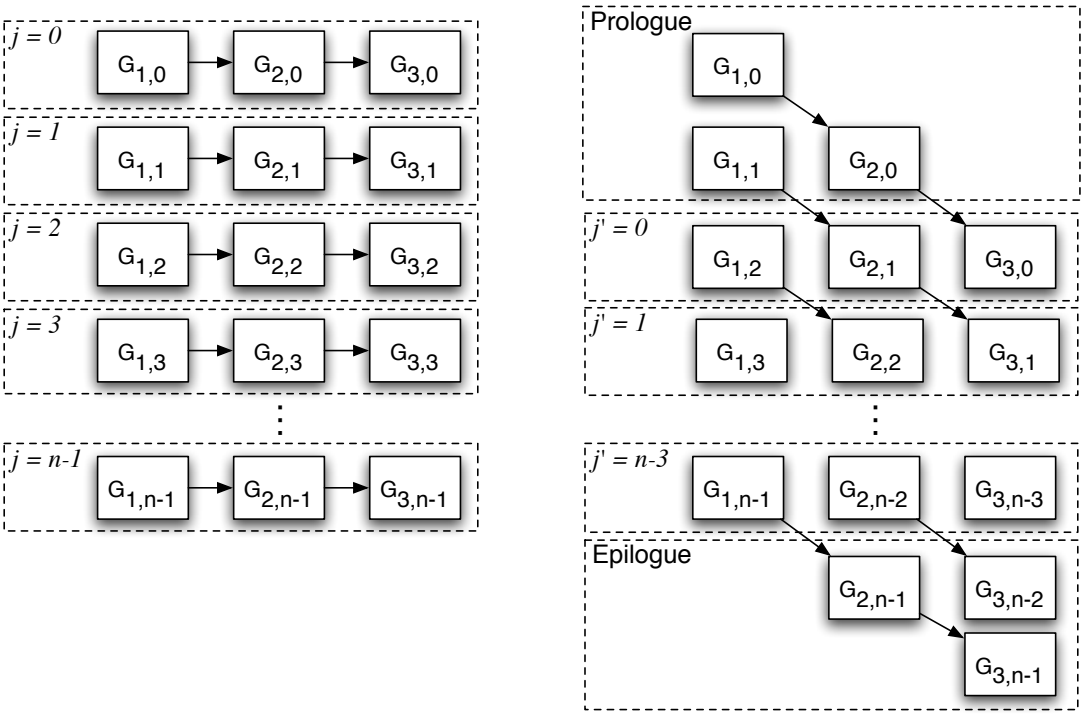

Figure 2: Software pipelining with three stages; left: loop independent data dependences in a non-pipelined loop restrict parallelism; right: the same dependences are now dependences between different iterations of the pipelined loop.

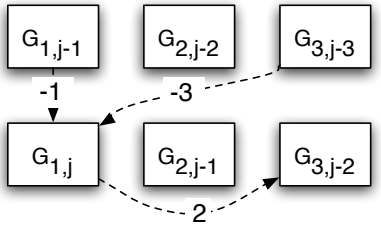

Figure 3: Inter-iteration dependences in a pipelined loop

Consider a loop that we can split into three stages, G_1, G_2 and G_3. In Fig. 2, $G_{i,j}$ denotes stage G_i operating on data for logical iteration j. If we schedule the loop without software pipelining (Fig. 2, left), the loop-independent data dependences between the stages will prevent us from exploiting much parallelism. With software pipelining (Fig. 2, right), these dependences are between different iterations of the pipelined loop and therefore do not restrict parallelism between the stages.

Figure 3 shows how different values of d can be realised in a pipelined loop. When we assign operations to stages (section 5), we will need to assign stages in such a way that the resulting assignment is *legal*:

Definition 4.1 *A* legal stage assignment with k stages *for a loop specification associates a stage number in* $\{1, \ldots, k\}$ *with each operation in the code graph such that*

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

It is tempting to simply split up the code graph G into k sequentially composed subgraphs, called stages, with $G = G_1 ; G_2 ; \ldots ; G_k$, and then to re-compose them in parallel, to yield $P ; (G_1 \otimes G_2 \otimes \ldots \otimes G_k)$, where P is a permutation to make the argument order of the code graph match up.

This approach, however, gets us no closer to actually scheduling a software-pipelined loop, as we are potentially violating both conditions for being loopable. 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.

4.1 Pipelining Transformation

We take a more specialised approach to sequential decomposition and connect 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.

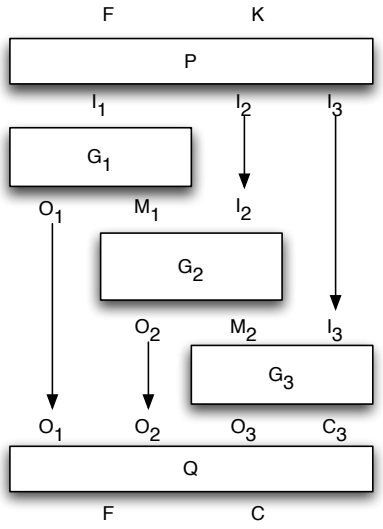

Figure 4: Splitting a code graph into three stages

For each stage $G_i : M_{i-1} \times I_i \times K_i \to O_i \times C_i \times M_i$, the I_i inputs 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 — obviously we have to set $M_0 = M_k = \mathbb{1}$.

Furthermore, we add the "control stage number" c as additional parameter c to the software pipelining and demand that all control outputs are generated in stage c, so we define $C_i = \mathbb{1}$ for all $i \neq c$, and $C_c = C$. This influences the meaning of the control outputs; see Sect. 4.2 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 wiring graph $P : F \times K \to (I_1 \times K_1) \times \ldots \times (I_k \times K_k)$ at the top to rearrange the I_i and K_i to match the order of $F \times K$, and a permutation wiring graph $Q : O_1 \times \ldots \times O_k \to F \times C$ at the bottom to make the order of the O_i match up with $F \times C$:

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

Constant inputs can appear on P's output list any number to times, other inputs up to twice. Figure 4 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 $(F', K, C, G', \mathbf{d}')$ where the stages are executed in parallel, but for different "logical iterations", *i.e.*, for different iterations of the original loop specification.

We have mentioned that an input value from F can be used in more than one stage; since each such input is updated during each iteration of a software-pipelined loop, it can be used by at most two consecutive stages (for more details see [11]. If an input is used

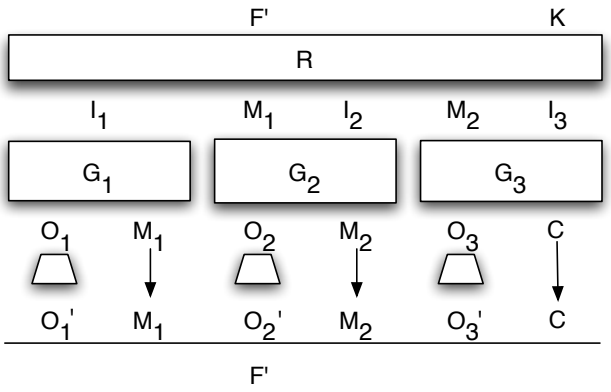

Figure 5: Putting the stages together in parallel

by two stages, then 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 $d' = 0$.

These additional input/output pairs will contribute to change F to F' and $\mathbf{d}$ to $\mathbf{d}'$; the corresponding wiring graph $X_i : O_i \to O'_i$ is used to adapt the stage graphs:

$$G'_i := G_i ; (X_i \otimes \mathbb{I}_{C_i} \otimes \mathbb{I}_{M_i})$$

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

$$G' := R ; (G'_1 \otimes \cdots \otimes G'_k) ; S,$$

where $F' := O'_1 \times M_1 \times \cdots \times O'_k$, and $S : O'_1 \times C_1 \times M_1 \times \cdots \times O'_k \times C_k \to F' \times C$ and $R : F' \to I_1 \times K_1 \times M_1 \times I_2 \times K_2 \times ... \times M_{k-1} \times I_k \times K_k$ are wiring graphs.

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 5 shows how these parts fit together for $k = c = 3$.

The entries in $\mathbf{d}'$ will be 0 for each first instance of duplicated O_i outputs, and -1 for all other O_i and M_i outputs. The wiring graph R distributes K constants to their K_i uses, propagates M_i inputs, also propagates non-duplicated O_i inputs (adapting for d if necessary); it wires the first occurence of each duplicated O_i input to the corresponding output in I_{i+d}, and the second occurence to the corresponding output in I_{i+d+1}.

4.2 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 on 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.

5. Algorithm for Stage Splitting

For splitting a given loop specification into k stages we use a heuristics that, in addition to finding a legal stage assignment, also strives to achieve the following goals:

1. The latency along the longest path through any stage should be at most the initiation interval we expect to achieve.

2. Not too many registers should be alive across stages.

   All registers that are alive across 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. Not too many forward dependences should 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.

Our heuristic 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 first use these values to decide approximately where to cut the code graph into stages; based on the dependences between operations, the selection of candidate cuts is further restricted; from the remaining candidates we finally 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 5.2) 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 code graph with all operations known to be in stage $i + 1$ or greater removed (see section 5.1)

   (b) Try to mark all operations with height $> h_{\mathrm{tot}}/i$ as being in stage $i - 1$ or less

(c) Try to mark all operations with $\mathsf{depth} > (i-1)d_{\mathrm{tot}}/i$ as being in stage i

(d) Let A be the set of all operations 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 5.3)

(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.

5.1 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 5.1 *Given a code graph G, then, for each node n and each edge e,*

- $\mathsf{depth}_{\mathrm{L}}(n)$ *is the depth of the producer edge of n plus the producer's latency, or 0 if n is an input;*
- $\mathsf{height}_{\mathrm{L}}(n)$ *is the maximum of the heights of all consumer edges of n incremented by their respective latencies, or 0 if n is an output;*
- $\mathsf{depth}_{\mathrm{L}}(e)$ *is the maximum of the depths of all sources of e;*
- $\mathsf{height}_{\mathrm{L}}(e)$ *is the maximum of the heights of all targets of e.* □

For an operation (a hyperedge) e in the code graph, the function $\mathsf{depth}_{\mathrm{L}}(e)$ 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, $\mathsf{height}_{\mathrm{L}}(e)$ 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 5.2 *Given a code graph G, then, for each edge e,*

- $\mathsf{height}_{\mathrm{U}}(e)$ *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 $\mathsf{height}_{\mathrm{U}}(e)$ becomes maximal.*
- $\mathsf{depth}_{\mathrm{U}}(e)$ *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 $\mathsf{depth}_{\mathrm{U}}(e)$ becomes maximal.* □

The lower bounds provided by $\mathsf{height}_{\mathrm{U}}(e)$ and $\mathsf{depth}_{\mathrm{U}}(e)$ 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 5.3 *Given a code graph G, then, for each edge e,*

- $\mathsf{height}_{\mathrm{LU}}(e) := \mathit{max}(\mathsf{height}_{\mathrm{L}}(e), \mathsf{height}_{\mathrm{U}}(e))$,
- $\mathsf{depth}_{\mathrm{LU}}(e) := \mathit{max}(\mathsf{depth}_{\mathrm{L}}(e), \mathsf{depth}_{\mathrm{U}}(e))$.

□

Furthermore, we define a *total height* $h_{\mathrm{tot}} := \mathit{max}_{n \in \mathcal{N}}\, \mathsf{height}(n)$ and a *total depth* $d_{\mathrm{tot}} := \mathit{max}_{n \in \mathcal{N}}\, \mathsf{depth}(n)$.

5.2 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, labelled with the integer d) is taken to denote the constraint

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

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 code graph 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 code graph 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 code graph, 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 $(i - 1)d_{\mathrm{tot}}/i$ 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.

5.3 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 fulfils 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 code graph (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

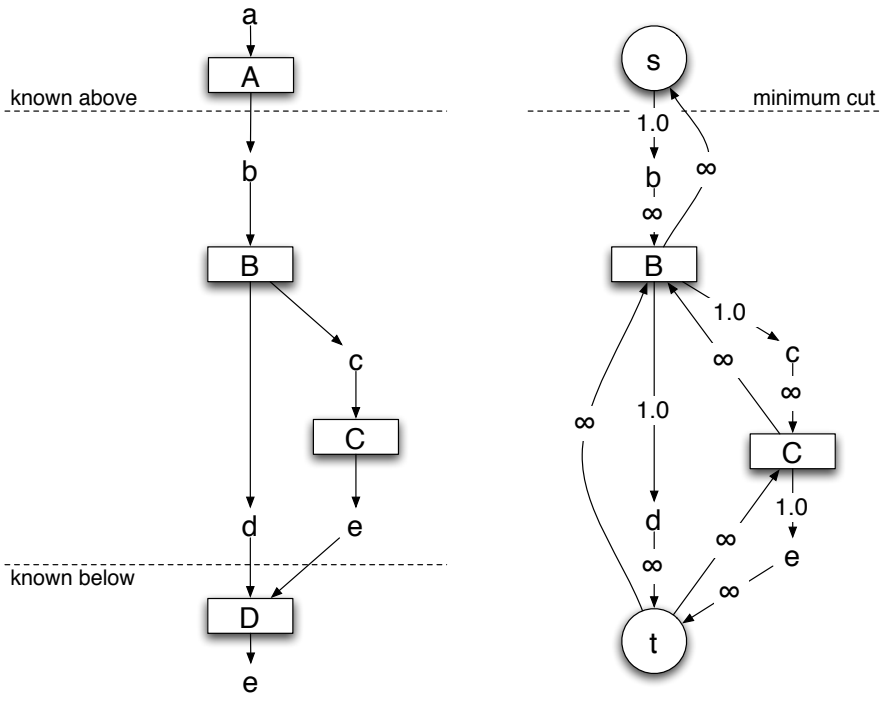

Figure 6: Transforming parts of a code graph (left) to a minimum cut problem (right). Edges `A` and `D` are known to be above and below the stage boundary beforehand; using the minimum cut we decide to put `B` and `C` below the cut, because that requires the least number of registers to be live at the top of the stage.

1. a minimum cut yields two stages with no illegal dependences, and
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 5.4 *Given a code graph G, we define the graph* G'_{cut} *as containing*

- *a "source" node s,*
- *a "sink" node t,*
- *for every operation in the code graph, an "operation" node,*
- *for every node in the code graph, 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.* □

Definition 5.5 *Based on G'_{cut}, a set A of operations (hyperedges in the code graph) 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 the graph G_{cut} by*

- *deleting all operation nodes in $A \cup B$,*
- *deleting all value nodes that are not connected to a consumer or producer 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 6 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.

As an example, Fig. 7 shows the stage assignment produced by our algorithm for the best result for the non-unrolled cosine function listed in Table 1.

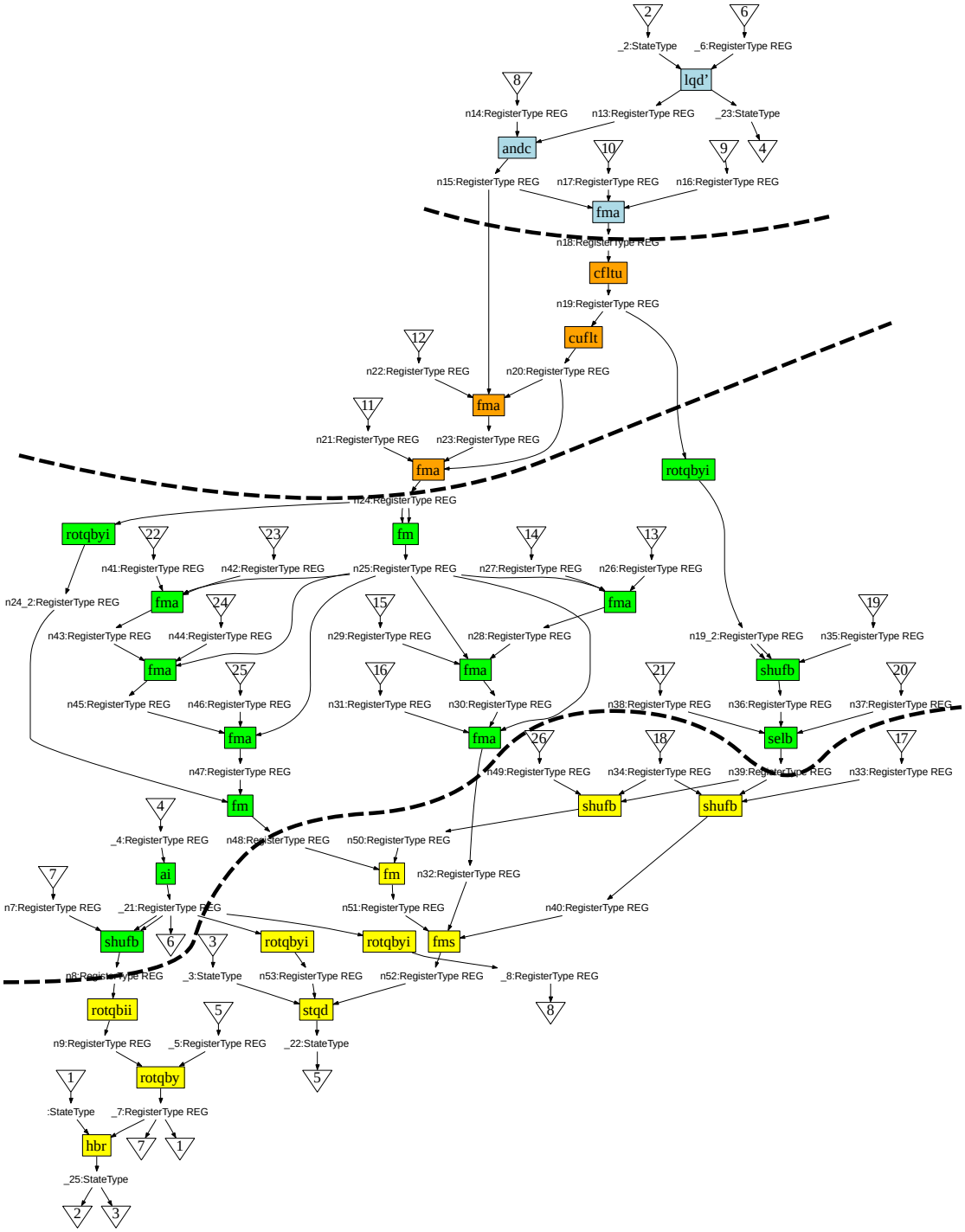

Figure 7: Stage assignment for cos with $u = 1$, $s = 4$, $m = 2$

6. Numerical Results

We used code graphs defined using COCONUT declarative assembly language [10] for the Cell SPU to test the performance of a prototype scheduler based on the algorithms described in this paper. We present the results for a library of basic mathematical functions, both against resource-based lower bounds and to modulo scheduling in `xlc`.

Each of the basic mathematical functions operates on SIMD vectors, and is iterated over input and output arrays of such.

We can apply two different preprocessing steps to the code before scheduling it; we can unroll the loop by replicating the loop body 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 5.1) by inserting register-to-register move instructions (we use the `rotqbyi 0` instruction, rotate quad word by zero bytes).

Table 1 shows the result of using ExSSP on the math functions; for 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 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 expected, we we are more likely to find a schedule close to the minimum bound at higher unrolling factors, although some optimal schedules occur even with $u = 1$. For $u = 4$, almost all functions schedule within a few cycles of the theoretical lower bound b.

Fig. 8 shows an overall 20 percent II reduction, when using ExSSP instead of `xlcs` implementation of modulo scheduling. This includes all functions implemented in the vector math library except the two-result sine/cosine pair function, which could not be scheduled by `xlc` using modulo scheduling.

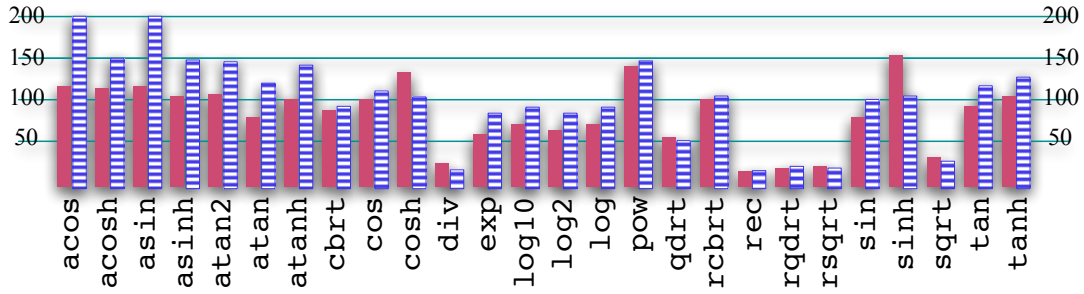

Figure 8: Iteration Interval (II) for Explicitly Staged Loops (maroon) versus `xlc` modulo-scheduled loops. Loops unrolled to a factor of four for ExSSP, and four or more times by `xlc`, but scaled to an unroll factor of four.

	$u=1$					$u=2$					$u=3$					$u=4$				
	s	m	b	n	r	s	m	b	n	r	s	m	b	n	r	s	m	b	n	r
acos	6	2	28	43	38	6	4	55	59	58	5	0	82	87	59	3	0	109	112	69
acosh	5	2	23	32	37	5	4	45	45	49	3	6	67	69	51	2	0	89	91	56
asin	6	2	28	54	39	4	4	55	77	44	3	6	82	91	61	3	8	109	110	76
asinh	5	2	21	35	36	5	4	41	42	46	3	6	61	61	51	3	8	81	81	62
atan2	2	2	24	103	32	2	4	47	103	43	2	6	70	105	52	2	8	93	107	60
atanh	7	2	20	21	40	3	0	39	42	41	3	0	58	59	51	3	8	77	77	61
cbrt	3	2	21	26	39	3	4	41	44	41	2	6	61	61	51	2	8	81	81	60
cos	4	2	26	29	41	4	4	51	51	53	3	6	76	79	58	3	8	101	102	70
cosh	6	2	17	28	36	6	4	33	34	44	6	6	49	49	56	4	0	65	65	61
div	2	2	13	24	13	2	4	17	20	17	2	6	21	23	21	2	0	23	24	22
exp	4	0	13	23	29	4	4	23	25	36	4	6	34	34	44	4	8	45	45	52
log10	5	2	15	23	35	6	4	25	25	48	3	6	37	37	42	3	8	49	49	48
log	5	2	15	23	35	6	4	25	25	48	3	6	37	37	42	3	8	49	49	48
pow	5	2	23	41	45	4	4	45	46	58	3	6	67	67	63	3	8	89	90	74
qdrt	5	2	13	25	17	4	4	25	40	23	4	6	37	50	30	2	8	49	59	34
rcbrt	5	2	24	32	40	5	4	47	49	56	4	6	70	70	65	4	8	93	93	78
rec	4	2	11	15	13	3	4	14	19	14	2	6	17	19	17	2	8	20	21	20
rqdrt	4	2	12	21	17	4	4	19	26	24	3	6	28	37	28	3	8	37	44	34
rsqrt	4	2	11	19	15	2	4	14	23	18	2	6	17	23	22	2	0	21	24	25
sin	4	2	26	30	43	3	4	51	51	50	3	0	76	76	59	3	8	101	101	72
sincos	5	2	29	32	50	3	0	57	57	51	3	6	85	85	63	2	8	113	113	66
sinh	6	2	17	28	36	6	4	33	34	44	6	6	49	49	56	4	0	65	65	61
sqrt	4	2	11	23	15	3	4	14	20	18	3	6	19	37	22	2	8	25	37	26
tan	5	2	33	40	45	2	4	65	65	47	2	0	97	97	57	2	8	129	129	63
tanh	8	2	19	25	37	6	4	37	37	46	5	6	55	55	54	6	8	73	73	74

Table 1: Results of scheduling math functions in simple loops.

7. Conclusion and Outlook

As a slight extension of the code graphs presented in [9], we defined loop specifications as the main conceptual vehicle to guide the code transformations necessary for software pipelining. The constraints on stage composition arising from first investigating the pipelining transformation on the level of composition of code graphs and loop specifications provided useful guidance for the heuristic we employ for stage splitting.

The prototype implementation of the resulting Explicitly-Staged Software Pipelining algorithm produces very efficient software-pipelined loop bodies, which makes our approach worth pursuing further.

We plan to investigate whether the heuristics of decomposed software pipelining described in [4, 5, 6], especially the circuit-retiming based approach given in [6], can be adapted to our framework. The algorithm of [6] has a known efficiency bound, but it does not currently have any provisions for pre-assigning stages.

In [9], the concept of *joins* is discussed; the idea is to allow a node in the code graph to have multiple producers; the scheduler is then expected to choose one alternative, based on which leads to the better code. This would also allow the stage decomposition heuristic to take advantage of joins, which may provide choices to produce values based on inputs with different iteration distances d. While resolution of stage-internal joins would still be left to the final instruction scheduling pass. the larger gains are expected from joins where one of the alternatives is chosen during stage assignment; this is in fact a generalisation of our current method to insert moves to eliminate long lifetimes of values.

Acknowledgements

The authors acknowledge William Hua for his help with the implementation, and IBM for support in testing and timing the math functions used for benchmarking.

References

[1] 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*, (Piscataway, NJ, USA), pp. 183–198, IEEE Press, 1981.

[2] M. Lam, "Software pipelining: an effective scheduling technique for VLIW machines," *SIGPLAN Not.*, vol. 23, no. 7, pp. 318–328, 1988.

[3] J. 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*, (New York, NY, USA), pp. 1–11, ACM Press, 1996.

[4] J. Wang, C. Eisenbeis, M. Jourdan, and B. Su, "Decomposed software pipelining: a new perspective and a new approach," *Int. J. Parallel Program.*, vol. 22, no. 3, pp. 351–373, 1994.

[5] F. Gasperoni and U. Schwiegelshohn, "Generating close to optimum loop schedules on parallel processors.," *Parallel Processing Letters*, vol. 4, pp. 391–403, 1994.

[6] P.-Y. Calland, A. Darte, and Y. Robert, "Circuit retiming applied to decomposed software pipelining," *IEEE Transactions on Parallel and Distributed Systems*, vol. 9, no. 1, pp. 24–35, 1998.

[7] V. H. Allan, R. B. Jones, R. M. Lee, and S. J. Allan, "Software pipelining," *ACM Computing Surveys*, vol. 27, no. 3, pp. 367–432, 1995.

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

[9] W. Kahl, C. K. Anand, and J. Carette, "Control-flow semantics for assembly-level data-flow graphs," in *8th Intl. Seminar on Relational Methods in Computer Science, RelMiCS 8, Feb. 2005* (W. McCaull, M. Winter, and I. Düntsch, eds.), vol. 3929 of *LNCS*, pp. 147–160, Springer-Verlag, 2006.

[10] C. K. Anand, J. Carette, W. Kahl, C. Gibbard, and R. Lortie, "Declarative assembler," SQRL Report 20, Software Quality Research Laboratory, McMaster University, Oct. 2004. available from http://sqrl.mcmaster.ca/sqrl_reports.html.

[11] W. Thaller, "Explicitly staged software pipelining," Master's thesis, McMaster University, 2006.
http://www.cas.mcmaster.ca/~anand/papers/ThallerMScExSSP.pdf.