MultiLoop: Efficient Software Pipelining for Modern Hardware

Christopher Kumar Anand Wolfram Kahl

McMaster University, 1280 Main St. West, Hamilton, Ontario Canada L8S 4K1

May 8, 2007– *Draft for Comment* –

Abstract

This paper is motivated by trends in processor models of which the Cell BE is an exemplar, and by the need to reliably apply multi-level code optimizations in safety-critical code to achieve high performance and small code size.

A MultiLoop is a loop specification construct designed to expose in a structured way details of instruction scheduling needed for performance-enhancing transformations. We show by example how it may be used to make better use of underlying hardware features, including software branch prediction and SIMD instructions. In each case, the use of MultiLoop transformations allows us to take full advantage of software branch prediction to completely eliminate branch misses in our scheduled code, and reduce the cost of loop overhead by using SIMD vector instructions.

Given the novelty of our representation, it is important to demonstrate feasibility (of zero branch misses) and evaluate performance (of transformations) on a wide set of representative examples from numerical computation. We include simple loops, nested loops, sequentially-composed loops, and loops containing control flow. In some cases we obtain significant benefits: *halving* execution time, and *halving* code size. As many details as possible are provided for other compiler writers wishing to adopt our innovative transformations, including instruction selection for SIMD-aware control flow.

1 Introduction

In the context of the move to multiple light-weight with increasingly-powerful SIMD support, we have reexamined the way control flow and particularly loops are specified, and present here a more general construction called a MultiLoop. The idea is to provide a vocabulary for high-level programmers to provide hints to instruction scheduling.

We analyse four representative examples of numerical calculations relative to the Cell Broadband Engine's Synergistic Processing Units (SPUs) [8], and find that the types of changes in instruction selection and scheduling enabled by the more general specification make a large difference in terms of execution efficiency and code size. Since MultiLoops contains normal loops as a special case, adoption of the MultiLoop only adds implementation possibilities. The MultiLoop is aimed at medium-level code transformations which imply constraints on instruction schedules, but not at low-level scheduling itself.

Yet we have found significant potential for increased performance by rethinking standard patterns of code generation. Specifically, we address the following features which are not standardized across architectures:

1. software branch prediction,
2. SIMD parallelism,
3. two-way dispatch executing "arithmetic" and "data movement" in parallel.

In our examples, all branches can be correctly hinted so that branch misses do not occur, and loop overhead can be largely shifted from the bottlenecked arithmetic pipeline to the non-arithmetic pipeline.

SPU Pipelines To simplify the discussion for readers who are not familiar with the SPU ISA and programming model, we refer to the two execution pipelines as "arithmetic" and "non-arithmetic", because this is a good approximation of the division of instructions handled by the two pipelines, and this division is very natural across processor families. Specific measurable claims are always made with respect to the division of execution as defined in the programming model, see [2] for details of that model. In the literature, the arithmetic and non-arithmetic SPU pipelines are called the even and odd pipelines in the literature, as a result of alignment requirements for maximum instruction dispatch.

Pipelined Loops Since the point of defining new control flow structures is to achieve higher levels of performance, we recall the best conventional approach to efficient loop implementation: software pipelined loops. Figure 1 shows a simple loop structure with three blocks such that the middle block depends on outputs of the top block, and the bottom block depends only on outputs of the first block. The left is

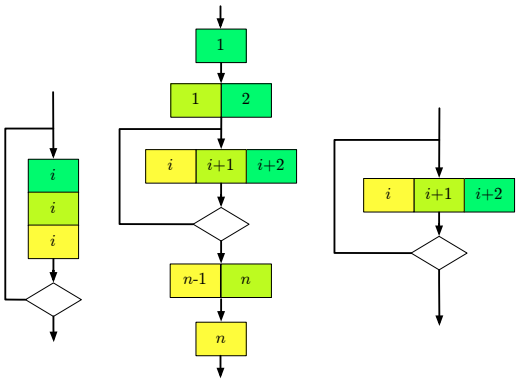

Figure 1: *Left:* a simple loop as it would appear in a high-level language, with three stages executed in sequence. *Middle:* a software-pipelined version of the loop, with stages are executed in parallel, prologue and epilogue. *Right:* without prologue and epilogue additional stages.

the usual view in a high-level language, and other two are software-pipelined versions. Horizontal juxtaposition of blocks means that the blocks are scheduled in parallel (interleaved on a processor with finite superscalar resources); the labels are the logical iteration numbers, which we define to be the iteration number of the loop in the standard (left) presentation.

In the version including loop prologue and epilogue (middle), the loop is equivalent to the original in the sense that the same instructions are executed (although in different orders) on the same data.

Without loop prologue and epilogue (right), we could get the same performance with one third the code, but we do not get the same execution. Using MultiLoop generators, we can meet loser specifications without significant additional cost.

Contribution: In this paper we present a class of *MultiLoop* control flow patterns, show how several typical numerical tasks can be formulated as MultiLoops, and introduce two MultiLoop transformation schemes that produce efficiently schedulable code. Both transformations generalize software pipelining as presented above.

Four examples demonstrate the applicability of these transformations, and give some hint as to the applications which will benefit the most. In particular, the Map example shows significant improvement and we present the loop overhead in enough detail for other compiler writers to implement as part of instruction selection.

Overview: In the next section, we review the aspects if the SPU instruction set relevant for this paper. Then we define and discuss the MultiLoop control structure and its transformations in Sect. 3, present examples in Sect. 4, and discuss related work in Sect. 5.

2 SIMD and the SPU ISA

Single Instruction Multiple Data (SIMD) instructions operate on more than one data element in parallel. They were introduced into commodity processors to accelerate multimedia synthesis and processing. These *instruction set architectures* (ISAs) are diverse in structure and implementation, with

VMX/Altivec [5] on Power and SSE [12] on x87 being the best-known.

Data Flow The SPU ISA [6] uses 128-bit operands, in common with VMX and SSE. It contains a rich set of operations formed by dividing the 128-bit operands into 8-, 16-, 32- or 64-bit quantities and performing the usual scalar operations independently on each. See Fig. 2, for an example 32-bit add instruction operating on four elements. This results in a

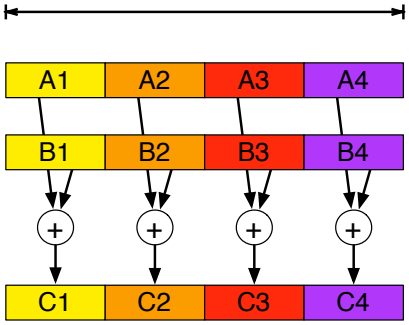

Figure 2: **fma**, a 32-bit add operating on two 128-bit wide operands.

useful level of parallelism, but introduces alignment issues in data. As a result, all SIMD instruction sets have some instructions to rearrange data. Two approaches are possible, a large set of instructions with specific functions (e.g. unpacking pixel data into vectors by component), or a small set of software-controllable instructions. All ISAs follow a middle path, with genericity increasing from SSE to VMX to SPU ISA. The instructions of most use in synthesizing loop overhead are the byte permute instruction **shufb** (analogous to VMX's *vperm*), shown in Fig. 3, and quadword rotate instructions, including **rotqbii**, which rotates the whole 128-bit vector up to 7 bits (immediate argument) left; this is the preferred register move instruction.

As shown in Fig. 3, SPU byte permutation can be used to move 32-bit components from one slot to another one (useful for transposing single-precision floating point matrices, for example), or duplicate bytes. It can rotate bytes through cycles, which can be used to count through loop induction variables when the loop sizes are known at compile time. Byte and bit

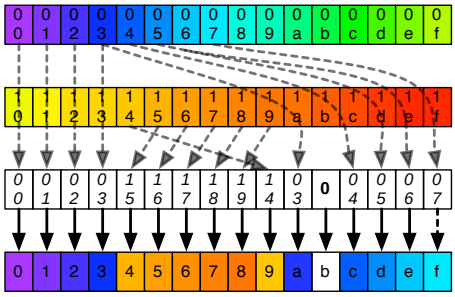

Figure 3: **shufb** byte permutation taking two source operands (coloured) and an operand of byte indices.

rotate instructions take both immediate counts and counts from operand registers.

Storage Model As we move to higher levels of parallelism on a chip, which Cell is pioneering, the overhead associated with a shared memory space (page tables, cache coherency, *etc.*) will increase superlinearly. To avoid this, the Cell processor's SPU compute engines have their own local stores (LSs), and use DMA to transfer data to and from system memory. This has the positive effect that access to local memory can be as predictable as accesses which hit in L1 cache on a normal processor. Unfortunately, it means that all code required for a particular task must be copied to the LS.

This puts a premium on achieving small code sizes, and changes the assumptions we can make about loops. Since input and output data on such light-weight engines will have to be transfered to and from local storage in digestible chunks, many more loop lengths will be knowable at compile time — especially if code is compiled just-in-time before distribution to multiple light-weight cores. The SPU's Local Stores are direct-mapped with wrap-around addressing, and no memory protection. As a result loads cannot raise segmentation faults, which makes it easier to replace clean-up code, prologues and epilogues with extra loop iterations, thus reducing code size.

Control Flow To produce efficient loop implementations at the single SPU level, we exploit branch hinting, which does not effect the results of execution, only timing. The **hbr** in-

struction takes as an immediate argument the location of a branch, and (as a register argument) the branch target.

Branch miss penalties have grown with increasing hardware pipelining within microprocessors. Hardware branch prediction requires additional circuits not required by the architected instructions. It is a natural feature to omit from compact processing elements designed for heterogeneous parallelism on a chip systems, like Cell.

3 MultiLoop Code Graphs

For hand-coded assembly language kernels, experts use certain patterns of efficient control flow. It is a meta-goal of the Coconut project to capture such patterns and abstract them into either code graph transformations at the level of our intermediate language, or into translators from higher-level structures into assembly language.

MultiLoops make it easy to express such patterns. They are the target of transformations from operations normally represented by nested or sequentially composed loops, and allow the folding of multiple branches into single branches. On the SPU, only a single branch hint is active at a time, so a successful strategy for software predicting branches can only work if branches are sparse.

Intermediate Language All intermediate structures take the form of labelled hypergraphs and nested hypergraphs. We do not use anything similar to a register transfer language.

Straight-line computation is expressed in *code graphs* [7]; these are data-flow hypergraphs with nodes corresponding to register values or generalised state component values, and with hyper-edges labelled by assembly instructions; these hyper-edges can have multiple inputs and outputs according to the arguments they use and the results they produce (see Fig. 13 in Sect. 4.2 for an example).

Our code graphs are strictly more expressive than data dependency graphs used in compilation algorithms based on Single Static Assignment (SSA), because state information is explicitly represented as nodes separate from data.

These data-flow-level code graphs are used as edge labels for non-branching edges in nested hypergraphs where the outer level represents control flow:

Definition 3.1 A *control-flow arrangement (CFA)* is a directed hypergraph with two kinds of hyper-edges, distinguished by their labels:

- A *code graph edge*, or *stage edge*, is labelled with a code graph, and has exactly one input node and one output node, labelled according to the input and output types of the code graph.
- A *control flow edge* is labelled with a control-flow assembly instructions that cannot occur in the data-flow level. □

Besides control-flow edges, another aspect of control flow is the existence of *control-flow join nodes*, i.e., nodes that are output nodes of more than one edge.

Certain properties of CFAs are important:

Definition 3.2 A control-flow arrangement is called

- *deterministic* iff each node has only one outgoing edge;
- *concurrent* iff some control flow edge has more than one input node (e.g., synchronisation primitives);
- *sequential* iff all control edges have exactly one input node;
- *tight* iff each node immediately adjacent to at least two code graph edges is a control-flow join node. □

Control flow edges with more than one output node are also called *branching edges*.

For this paper, we are only interested in a restricted kind of CFA; we chose the name since the original motivation was that of a generalised deterministic non-concurrent loop specification that essentially can be seen as implementing a "loop with multiple bodies":

Definition 3.3 A *MultiLoop* is a deterministic sequential CFA. □

For example, Fig. 4 (Left) shows a MultiLoop for the loop pipelining shown in the introduction, in Fig. 1. This CFA is not tight since it contains the sequence S1, S2, S3 of code graph edges (the *stages*) without intervening control

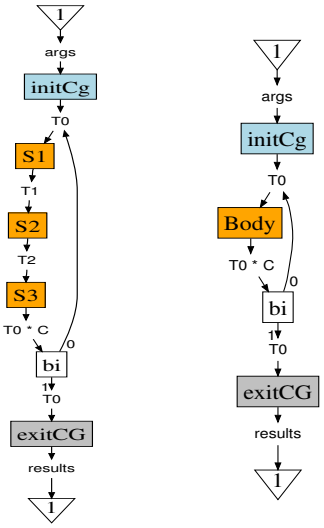

Figure 4: *Left:* MultiLoop for three-staged loop pipelining. *Right:* Tight MultiLoop of simple loop.

flow joins; this sequence is semantically equivalent to a single code graph node containing the sequential composition Body = S1 ; S2 ; S3 of the three stage graphs. Such a non-tight CFA can be *tightened* by replacing such sequences with the results of the corresponding sequential composition; in this case, the corresponding tight MultiLoop has a single Body edge inside the loop, shown in Fig. 4 (Right). In both cases, the node below the initCG code graph edge is a control-flow join node, and the bi hyperedge is a branching edge.

From a scheduling point of view, the goal of intermediate code generation is to obtain a tight MultiLoop where typically there is a branch after every code graph edge, each code graph can be densely scheduled, and each code graph is sufficiently deep to hint its branch in time.

However, just tightening a non-tight MultiLoop does not normally achieve this goal.

Some function bodies are designed or generated directly as non-tight MultiLoops, for example when a common pattern underlies the generated code. Other function bodies are generated first as tight MultiLoops, and then "arranged" into MultiLoops with longer stage sequences, for example using the heuristic for loop pipelining stage separation described in [17]. There are also cases where a function body is first generated as a non-tight MultiLoop, and then re-arranged into a different non-tight MultiLoop, with additional constraints imposed by the original staging. An example for this will be explained in Sect. 4.2; we show the original non-tight version in Fig. 5 to the left, and to the right the re-arranged version where midCG has been split as sequential composition midCG $= m_1; m_2$, and the two halves have been integrated into the adjacent stages, yielding Top1CG = top1CG; m_1 and Bot12CG $= m_2$; bot12CG *etc.*.

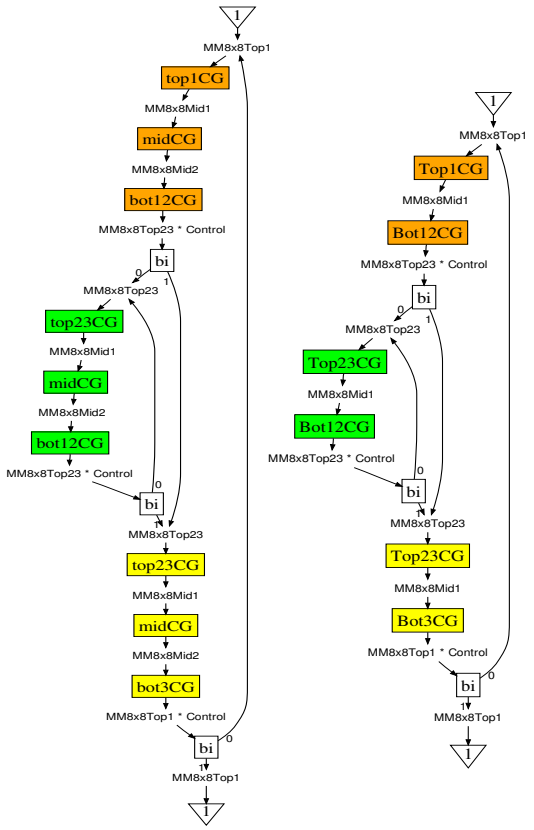

Figure 5: *Left:* Source MultiLoop for Matrix Multiplication *Right:* Re-arranged version.

The scheduling objectives are then achieved via *MultiLoop transformations*, and in Sect. 4 we show example applications for the two high-level transformations described in the following.

MultiLoop Pipelining: A loop body software-pipelined into n stages processes data from n consecutive logical iterations in parallel. This approach can also be used for more complex MultiLoops. While in the case of simple loops, as in Fig. 4 (Left), those n data sets are always processed by the same set of the n stages of the loop body, in a more general MultiLoop, different combinations of n stage graphs will be needed to process data from n consecutive logical iterations.

Although for arbitrary code graphs, the code size could multiply by a factor exponential in n, if this is done for a highly regular MultiLoop, with a small set of equal stage graphs occurring with the same periodicity in different sequences, then performing this pipelining transformation followed by a minimization step can lead to tight MultiLoops with no or small code size increase.

The examples presented in sections 4.1 and 4.4 rely on the MultiLoop pipelining transformation.

MultiLoop Re-Sequencing: The initial and final parts of many loop bodies are hard to schedule densely, since there are frequently too many dependencies in these regions. Since not all of these dependencies carry across the loop boundary, however, moving the boundaries improves efficiency.

As a MultiLoop transformation, Re-Sequencing combines sequences of stage edges with intervening branch nodes, producing a separate sequence for each way such intervening branches can take. Conceptually, this moves all branches to the beginning of the sequence, choosing "prophetically" the right sequence. Therefore, an appropriate transformation of the branch logic is necessary; this is particularly easy in counted loops with counts known at compile time. For the rearranged MultiLoop from Fig. 5 (Right) we show the immediate (machine-generated) result of this re-sequencing transformation in Fig. 6. The one-element sequenced code graph edges only serve to connect start and end nodes of the MultiLoop; each of the five two-element sequenced code graph edges results from a different way of re-sequencing the branches with one stage before the branch and one after.

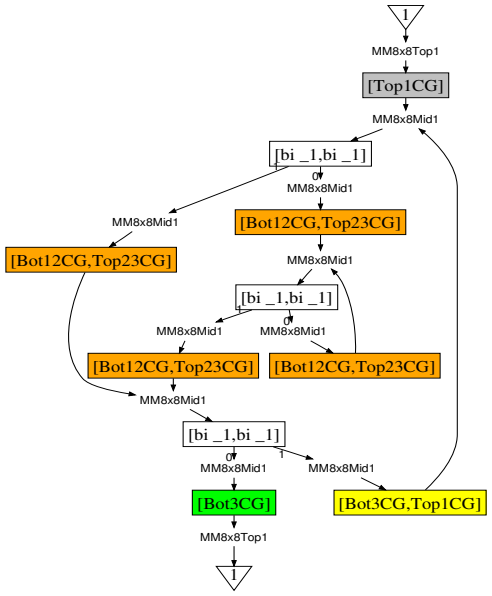

Figure 6: Raw Re-Sequenced MultiLoop for Matrix Multiplication

Because of the regularity of the input MultiLoop Fig. 5 (Right), only two different two-sequence code graph edges remain after standard control flow minimisation together with a slight rearrangement of the branch logic (see Fig. 7).

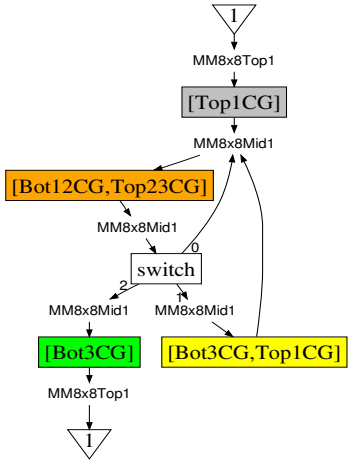

Figure 7: Minimised Re-Sequenced MultiLoop for Matrix Multiplication

In Sect. 4.2 we explain how the "prologue" and "epilogue" edges can be eliminated in this

case; similar approaches can be made to work in other cases, so that for highly regular MultiLoops, the re-sequencing transformation also results in little or no code size increase.

The example presented inSect. 4.3 also relies on the MultiLoop re-sequencing transformation.

Programmer Interface The MultiLoop allows very general control flow. It is a tool for formal specification and verification of control-flow transformations, and we have also found it very useful in reasoning informally about proposed performance-enhancing transformations. It is not intended as an everyday programming language, and we envision a short list of interfaces for generating MultiLoops with specific structures. All of the examples in this paper can be understood in terms of loops with conditional execution, which can be presented as loops with versioned loop bodies. The programmer separately specifies declarative components, specifies their composition into multiple loop bodies, and puts constraints on staging as necessary (see Fig. 8).

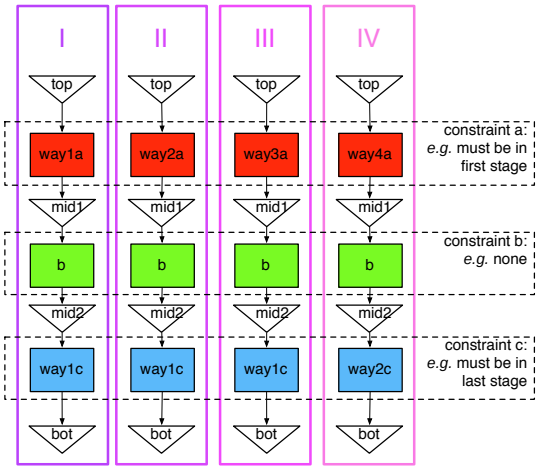

Figure 8: Four-way MultiLoop bodies.

Well-structured conditional execution can be refactored into this form. The final component in each loop version must contain a distinguished control output which contains the branch address for the next loop version.

In this way, application specialists can optimize the loop overhead for common high-level patterns, including transformations which depend on the staging. Four such examples are given in Sect. 4. In a conventional tool chain, this would not be possible because control-flow transformations are performed on the intermediate language(s) without exposing important pieces to the high-level language programmer.

4 Examples

In many application domains, execution efficiency is largely determined by loop scheduling. Loops can be classified by

- whether the number and pattern of iterations is known at compile time or run time;
- whether the number and pattern of iterations is data dependent (*i.e.*, whether it can be calculated from values in registers at the start of the loop, or depends on values loaded through the course of the loop, or calculated iteratively);
- whether the loop is nested or not nested;
- whether the loop is isolated, or sequentially composed with similar loops.

The examples cover these cases, but not in all combinations. Data dependent loops are more complicated in general, and only the resampling example covers this case.

4.1 Mapping over Arrays

Mapping a function over a structure (list, set, tree) is a common pattern in all programs. With common memory organizations, maps over arrays (contiguously stored lists) are the most efficient. Some hardware vendors provide optimized libraries of mathematical or graphics functions mapped over arrays.

We have found that in this case, a MultiLoop specification can shift most of the loop overhead from the arithmetic pipeline to the non-arithmetic pipeline. For mathematical functions, computation is almost always limited by arithmetic in the arithmetic pipeline, so this results in a measurable speedup. For a standard math library, the resulting reduction in the resource lower bound using this formulation will drop by two to ten percent, based on the implementations reported in [17, Tables 7.1 and 7.2].

Since this is our smallest example, we will go through all of the loop overhead, showing how a single arithmetic instruction is sufficient to move two pointers, update a counter and calculate the branch address for the loop. We will show it for the case of mapping a function, $f :: RegVal \rightarrow RegVal$, with one input and one output. There is one unused word in our control quadword, so this would work just as well for three-pointer maps (one input and two outputs, or two inputs and one output).[1]

In Fig. 9, we show the four instructions of loop overhead for such loops. The control quadword (1) contains the input and output pointer and the counter (and one unused 32-bit word); a single add instruction (2) updates all three active elements by the increments in the register constant (3). Single-word add is an arithmetic instruction and slants to the right. All other instructions are non-arithmetic and slant to the left. All non-immediate load/store

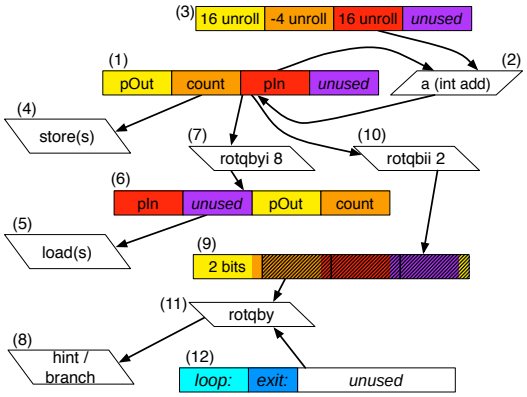

Figure 9: Overhead for a map loop.

instructions take the address from the first word in a register quadword, so the stores (4) use the control word as an argument. The loads (5) need their argument (6) rotated (7) by two words, *i.e.*, 8 bytes. The branch address (8) used by the branch and the hint instructions is calculated via a byte rotation count (9) which is calculated by shifting (10) the two high-order bits from the count word into the low-order bits of the first word. Since loads and stores and 16-byte aligned, the four low-order bits of *pOut* will always be zero, so the only significant bits of the rotation count (9) are the two bits shifted in from the count word. If the count is initialized to the number of elements to be mapped, then the first count times through the loop, those two bits are zero, but on iteration (count + 1), the count becomes negative and the high bits are set. So the rotation count is either zero (initially) or three (on the ultimate iteration). The count is used to rotate (11) a quadword (12) composed of the addresses of the top of the loop and the loop exit. If the function ends with the loop doing the mapping, and no non-volatile registers are used in the body, then the exit address for the loop can be the address in the link register. In this case the function return is hinted and the total number of branches is equal to the number of values mapped divided by the unroll factor.

This example is interesting for two reasons: one, it is a very efficient implementation of a common loop pattern; and two, in the natural software-pipelining of this loop the loads will be in the first stage, and the stores the last stage, so the add instruction (2) in some sense belongs to multiple stages and the control word (1) contains values from different logical iterations. We view the transformation of the loop overhead required to software pipeline this loop as being on the same level as loop unrolling, and implement the two transformations together in the corresponding code generator. The output of the generator includes iteration distances between outputs of one stage and inputs of the next stage, and stage constraints on load and store operations.

4.2 Dense Matrix Multiplication

Matrix matrix multiplication and multiply-addition are important basic operations in linear algebra. Imperatively, they are encoded using nested loops. To achieve efficiency, they are always unrolled by blocking calculations into rectangular submatrices. When the blocks have the same structure, there is only one calculational kernel, and differences in loop levels and iteration manifest in varying pointer cal-

[1] Less obvious is that that the same idea works for up to seven-pointer cases, if the pointer arithmetic is performed using 16-bit integers. If the ranges permit, the 16-bit integers can be used as-is, but in general it is necessary to put addresses modulo 16 in the control quadword, and shift the result by four bits before extracting pointers.

culations. We have transformed such calculations into branch-free form, including the calculation of branch addresses in time to hint **all** *branches. We have analyzed the cost-benefit for the dense matrix case (no a priori information about zero matrix entries), and found that the expected improvement on a Cell SPU with perfect scheduling is small. This is because such basic linear algebra has a low computation to data size ratio. Significant benefits would accrue as this ratio rises, e.g., if matrix elements are computed on the fly, or unpacked from complicated sparse representations.*

We analyze the case in which single precision floating point dense matrices are stored in row-major order (corresponding to a statically-declared two-dimensional C array). SIMD parallelism enforces the grouping of row elements into quadwords containing four floats. In practice, matrix sizes larger than the SPU Local Stores are multiplied. We will assume that such matrices are already blocked contiguously into *big* blocks with dimensions divisible by four. We then further block them (non-contiguously) into *small* blocks. In Fig. 10, the *small* blocks are 8×2 quadwords, which is 8×8 floats, and are shown as contiguous squares, with quadwords separated by solid lines. The *small* blocks must divide evenly into the *big* blocks.

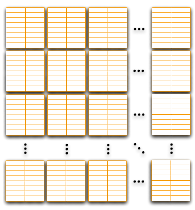

Figure 10: Block structure for square matrices, showing division into SIMD vectors.

We choose a block size large enough to ensure:

1. SIMD vector boundaries are on block boundaries (including results of transposes);
2. there are sufficient independent parallel **fma** operations to saturate pipelined execution units during the multiply;
3. the cost of non-arithmetic load, store, and transpose operations is smaller than that of the arithmetic operations (the arithmetic cost grows cubicly and the load/store/transpose cost quadratically in the width of square small blocks).

The minimum square block size with these properties is 8×8, in which case 128 **fma**s are needed. We will use these numbers in the following analysis. It follows that a lower bound on the scheduled cost per iteration will be the number of **fma** instructions plus any arithmetic instructions used in the loop overhead.

Iterating over blocks we form sums of products $C_{ij} = \sum_k A_{ik} B_{kj}$ of blocks. In an imperative language this would be

```
for i
  for j
    C[i,j] ← 0
    for k
      C[i,j] ← C[i,j] + A[i,k] * B[k,j]
    branch
  store C[i,j]
  branch
branch
```

On a processor with static hinting, the inner branch would miss once per iteration of the middle loop, and the middle branch would miss once per outer iteration, and the outer iteration would miss once. The relative cost of the misses decreases with the number of blocks in the multiplication. For an 8×8 array of 8×8 dense blocks it is about two percent of the theoretical mimimum execution time on an SPU.

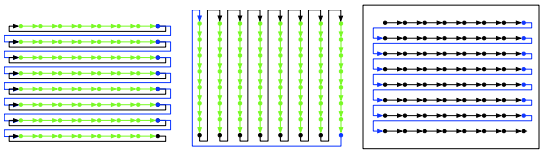

Figure 11: *Left:* Pointer movement through A blocks, showing movement in green associated with the inner-most loop, in black associated with the middle loop, and in blue associated with the outermost loop. *Middle:* Pointer movement through B blocks, with the same colours. *Right:* Pointer movement through C blocks, with the same colours.

By applying the re-sequencing transformation to a MultiLoop formulation, we can eliminate branch miss penalties and reduce overhead

to a single integer add, **a**, that increments three pointers. For short and fixed-length loops, we can use permute and rotate instructions to generate branch addresses on the fly in the non-arithmetic pipeline.

We reformulate the triple-nested loop as a MultiLoop with three versions, with three components each. The three versions correspond to the first iteration of the inner loop, the middle iterations and the last iterations. The middle component (midCG in Fig. 5) is unchanging and performs the multiply, as shown in Fig. 12.[2]

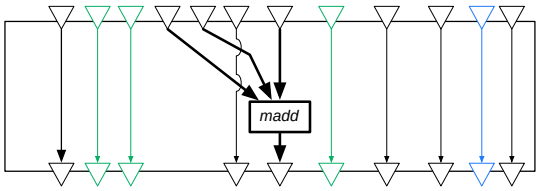

Figure 12: The middle code graph multiplies.

Figure 13 shows two cases (top1CG and top23CG in Fig. 5) of the load and increment portion. The first case is for one version of the loop which loads or zeros *C* depending on whether a multiply or multiply-add is being performed. The second case only loads the next block of *A* and *B*.

The last components, bot12CG and bot3CG in Fig. 5, shown in detail in Fig. 14, contain the branch hint and, in the second case, which is used in the third version of the loop body, the store of the finished *C* block.

Separate code graph components for the loop overhead allow for even smaller loop overhead code bodies. The address calculation, and hence the control-flow, handled by *iterInner* and *iterOuter* and the nodes they connect forms a disjoint subgraph. Effectively, the inner loop of the imperative description is encoded by *iterInner* and the two outer loops by *iterOuter*. Similary, the pointer movement is handled by different components for the inner-loop, *wayInner*, and the two outer loops, *wayOuter*. The number and type of inputs and outputs are not constant, but the outputs of a loop version match the inputs of a loop version which can legally follow it.

If we apply the re-sequencing transformation to the MultiLoop formulation with two stages per loop body as shown in Fig. 5 (Right), then there are only two scheduled versions of the re-sequenced loop body required, as shown in Fig. 7. This makes the code size for a scheduled MultiLoop roughly the same as a software-pipelined version of the imperative code, in which the code size for the inner loop would be double the size of the loop body.

[2]In the following code graph drawings, inputs and outputs are represented by triangles. Nodes for the state of the local store (green) and branch hint (blue) put state explicitly into the code graphs.

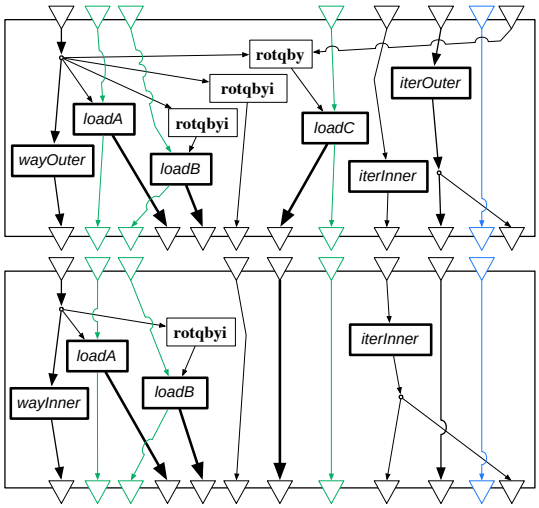

Figure 13: Top code graphs, with *C* load (*top*) for the first loop version and *C* accumulation (*bottom*) for the last two.

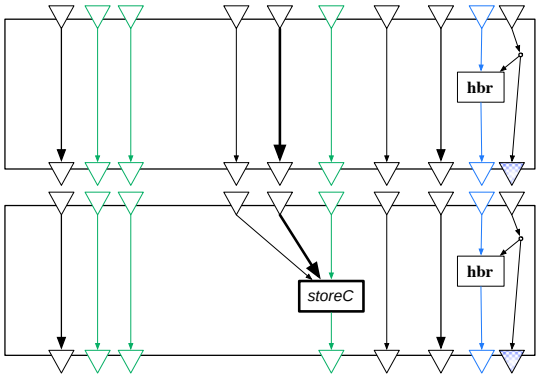

Figure 14: Multiply bottom code graphs, with *C* accumulation (*top*) for the first two loop variations and *C* store (*bottom*) for the last.

Loop Induction via Permutation Groups. To reduce the triple loop nest to two code MultiLoop bodies and synthesize the required pointer movement in Fig. 10 using shuffles and rotations, we use two key observations: (1) that top1CG occurs in logical iterations congruent to $0 \bmod 8$ and bot3CG occurs in logical iterations $7 \bmod 8$ which are paired together; (2) the pointer movement is periodic.

The action of any finite group can be simulated in this way, although we have only encountered Abelian groups up to the present. Groups with power-of-two sizes up to 32 can be simulated using word, half-word, byte or nibble rotation. Groups with other orders must be simulated using byte permutation via **shufb**. Both approaches can be used to directly cycle through offsets and addresses, or to indirectly cycle through indices into such, and then use rotation or permutation with register arguments (**rotqby**, **rotqbi** or **shufb** + key synthesis) to look up the actual offsets and addresses. Indirect permutation is especially effective when the list being permuted has a great deal of overlap. By increasing the order of indirection, *e.g.*, permuting lists of permute constants, the action of direct products of groups can be so constructed.

We can do without a prologue and epilogue, and only need one MultiLoop to do both matrix multiplication and multiplication-addition because we exposed load and store addresses as inputs and outputs, and put constraints on their staging. Loads on the SPU are safe (they cannot cause segmentation faults), so extra loads "past the end of the loop" can be ignored. Stores are never safe, but we know in which logical iterations extra stores will occur. In Fig. 14, the address for the stores is exposed as an input. Normally, this address is produced by a **rotqbyi** in the first iteration the address is generated in the init block, so we can set the initial store address to the location of a scratch memory area. Only a non-contiguous scratch area the size of a small block is required.

To produce both multiplication and multiplication-addition, we always include a loadC which takes its address from a **rotqby** controlled by an exposed register input, which we set to rotate either to the address which will be used in the next storeC or to load from a constant scratch memory area stocked with zeros.

For this example, the performance advantage provided by the MultiLoop will be relatively small, because the loop body itself needs to be so large to avoid being performance-limited by load/store activity. The encapsulation of the loop overhead in the MultiLoop will work for any sparse structure with identical blocks. For varying blocks (which is more common) a more general set of loop overheads will be required. This is the advantage of the MultiLoop for blocked matrix multiplication: efficient loop overheads for new block sparse structures can be supplied by the developer and do not have to be encoded in the compiler.

4.3 Partial Fourier Transform

MultiLoop formulations of multi-dimensional separable transforms lead to significant code reduction. In the simplest cases, conventional specifications lead to sd times more instructions, where s is the number of pipeline stages used in a schedule and d is the dimensionality. In the complicated case we study it is a factor of two.

Partial Fourier Transforms are Fourier transforms composed with coordinate projections or injections, *e.g.* they take a vector of 256 complex values, perform a Fourier transform, then copy out the central 192 elements. In the other direction we pad the input with zeros. Similar operations occur in wavelet and other fast transforms.

The MultiLoop and a re-sequencing transformation can be profitably applied when these operations are applied in multiple dimensions. Most fast transforms are, by definition, separable, which means that they can be applied independently in each dimension. In imperative code, this is expressed as a sequential composition of nested loops. We consider the three-dimensional case.

Such transforms are most efficiently computed in one direction. On scalar machines that is usually the direction in which addresses change the slowest. To use SIMD parallelism, another direction should be used, otherwise a lot of shuffling among registers is necessary.

To avoid having to transform in the expensive direction, we transpose before stores:

for *dimension* **in** *dimensions*
  for *quad-column* **in** columns(*dimension*)
    load *quad-column*
    transform *quad-column*
    transpose *quad-column*
    store *quad-column* as row

(loading enough columns to completely fill all loaded quadwords, *e.g.*, 4 floats).

For three dimensions, this looks like

$$xyz \rightarrow Yzx \rightarrow ZxY \rightarrow XYZ \qquad (1)$$

Index label order indicates array ordering, and case indicates whether a transform has been performed in the direction labelled by that index. Each is a loop of column transforms with code graph

$$L \rightarrow F \rightarrow S, \qquad (2)$$

where L is a complete set of loads with necessary data reforming for one row, F performs the transform, and S is a complete set of stores, with necessary data reforming for one row. We permute the indices cyclically (via a transpose) so that every dimension is transformed in the "middle" dimension, because SIMD parallelism is free in the "column" directions.

For *Partial* FFTs, the size of the dimensions before and after the one-dimensional transforms are not necessarily the same. This affects the load and store subgraphs by changing the striding pattern both within the column and the enumeration of columns, which changes register constants or immediate constants, depending on the type of indexing. Immediate-indexed forms reduce register pressure, and are more complicated for the MultiLoop, so we consider that case.

There are two different versions of L and of S depending on whether the first index direction has been transformed or not. We now specialize to the three-dimensional case: xyz has one load pattern and Yzx and ZxY have another; if more than one column has to be transformed in parallel then Yzx and ZxY have one store pattern and XYZ has a second. Load and store patterns can also differ if other data reformatting is performed, *e.g.* interleaved complex

$$..., r_j, i_j, r_{j+1}, i_{j+1}, r_{j+2}, i_{j+2}, r_{j+3}, i_{j+3}, ...$$

into planar or quadword-interleaved format

$$..., r_j, r_{j+1}, r_{j+2}, r_{j+3}, i_j, i_{j+1}, i_{j+2}, i_{j+3}, ...$$

The three sequentially-composed loops become a three-way MultiLoop with three components

$$\left((L_1, F, S_{1|2})|(L_{2|3}, F, S_{1|2})|(L_{2|3}, F, S_3)\right).$$

For all but very small transforms, two stages are sufficient to hide instruction latency in a software-pipelined loop, so the computation graph F is only split once, and the scheduled loop bodies will execute in this order:

(load,store) direction	(load,store) version	number of executions
$(1, *)$	$(T_1, S_{1\|2})$	n^2
$(2, 1)$	$(T_{2\|3}, S_{1\|2})$	2
$(2, 2)$	$(T_{2\|3}, S_{1\|2})$	$nm - 2$
$(3, 2)$	$(T_{2\|3}, S_3)$	2
$(3, 3)$	$(T_{2\|3}, S_3)$	m^2
$(*, 3)$	$(T_{2\|3}, S_3)$	m^2

for an n^3 to m^3 transformation.

In this case, the re-sequenced MultiLoop allows us to schedule three versions of the loop body, whereas three sequential software-pipelined loops would require three prologues and three epilogues, effectively doubling the code size. Even if we used a prologue and epilogue in the MultiLoop case, the code size is still reduced by $(d + 1)/(2d)$, where d is the number of dimensions.

4.4 Resampling in NUFT

This is the most specialized code we have encoded in a MultiLoop. It processes a one-dimensional array of velocities and data, and paints a three-dimensional array with the data. The MultiLoop has 54 different cases with branching dependent on the cumulative values of the velocity data, so simple unrolling would be impossible. Using a pipelined MultiLoop representation cuts the resource bound on execution time in half, over alternative representations, at the cost of increased code size. For the target applications (Nonuniform Fourier Transform)[3]*, this is a desireable trade.*

[3] This computation, together with the previous Partial Fourier Transform can be used to perform a nonuniform Fourier Transform, *e.g.*, used to reconstruct Magnetic Resonance Images from irregularly-sampled data, but the computational issues are more general.

A time series of positions and data are read in from one-dimensional arrays and rendered to a three-dimensional array of complex values. One can think of it as a variation of rasterization of anti-aliased lines in an array of pixels. In this case the pixels have complex floating-point values. Such tasks do not fit the simple SIMD pattern of parallelism. Single input data elements produce results in multiple neighbouring array elements. It is not possible to align the output with quadword boundaries. The only universal method of handling unaligned data access is to load or store all of the quadwords which overlap the required data and use permute or rotate instructions to align the data within quadwords.

In the simplest imperative implementation, the array data are read, modified and written as required. The load and store instructions would then dominate the computation and double the execution time (assuming perfect scheduling). A more complicated implementation which kept a cache of array values in registers and stored and loaded values as required would require multiple if statements nested six-deep or a switch, with so many missed branches, it would be much less efficient than the MultiLoop version. Since the conditions depend on the contents of the data cumulatively, unrolling cannot be used to hide the latency of the condition calculations. For computations following this pattern (rasterizing curves in higher dimensions), load/store activity can be significantly reduced by caching the active hypercube in registers, if the register file is large enough.

In our example, input data influences a 4^3 neighbourhood of array values, and the array values are complex, single precision floating point numbers, stored in interleaved real/imaginary row-major order. Each quadword contains two such values, so 32 quadwords are required for an aligned cache. Since complex floats are eight bytes wide, we are either aligned or unaligned (with the natural cache boundary in the middle of an aligned quadword). In the aligned case, rows cover three quadwords instead of two, so the number of registers for an unaligned cache is 48 if we store the entire quadword. On a 128 register machine, like the Cell SPU, there is room for such a buffer.

We assume that the active section of the array changes by at most one element along each of the axes. There are, therefore, 27 different possible movements. Since the cache can be aligned or not aligned on quadword boundaries, each direction of movement requires two different sets of load/store activity.[4] The amount of load/store activity depends on the movement, *e.g.*, with eight or twelve loads and stores, respectively in the aligned and unaligned cases.

Compare this with the number of loads and stores required for a single loop body in which all conditionals are synthesized using selection and permutation. In this case, 48 quadwords must be loaded before the computation and 48 quadwords stored after the computation. In addition, a shuffle map must be constructed on the fly to rotate values in the unaligned case, and this must be applied to each of the 48 quadwords between the computation and the loads and stores, for a total of 192 non-arithmetic instructions. Since the scheduled II for the most common (stationary) cases is 86 cycles, [17], and averaged over expected execution traces would not be more than 100 cycles, not using the MultiLoop would cut performance in half.

5 Related Work

Although our goal in this paper is to introduce a vocabulary for specialized control flow, and *not* to introduce a new scheduling algorithm, there is a lot of overlap between our work and the literature on optimizing compilers.

Specifically, we are motivated by the difficulty in incorporating conditional execution into software pipelined loops. This is a well-researched problem.

The current leader in scheduling algorithms for software pipelined loops is Swing Modulo Scheduling [11]. The most-commonly followed method of incorporating conditional execution into such algorithms is hierarchical reduction [9]. Another alternative is Enhanced Modulo Scheduling [18].

[4] The stationary case does not require any load/store activity in the backing store, but our implementation uses two identical implementations to simplify indexing into the jump table.

Methods which take loops or loop nests containing conditional execution and effectively break them into multiple paths include Multiple-II Modulo Scheduling (MII-MS) [19], All Paths Pipelining (APP) [14], and Split-Path Enhanced Pipeline Scheduling (SP-EPS) [13], both of which allow multiple IIs for a single loop. APP specializes the loop according to branches being taken or not taken, and modulo schedules each version. Additional edges are added to facilitate jumping between different schedules. SP-EPS, on the other hand, takes a control-flow graph for a loop (or loop nest) and expands at branch points by splitting tails. Another approach is the global software pipelining (GURPR-family) [15, 16]. These methods involve significant data dependency graph manipulation, but none of these methods incorporate instruction selection. Incorporating instruction selection capable of finding the code sequences used in our loop overhead examples would be prohibitively expensive.

Enhanced Co-Scheduling [4] extends the state-machine model for pipelined processors, and derives a modulo-scheduling algorithm. Unlike our approach, they start from the machine level and build up to find a useful level of formalization.

Most of these algorithms are designed with Very Long Instruction Word (VLIW) architectures (a form of Multiple Instruction Multiple Data, MIMD) in mind, rather than lightweight, heterogeneous multi-core systems exploiting SIMD parallelism. In a way, heterogeneous chip multiprocessors like the Cell BE have a much simpler model of execution than the VLIW machines and systolic arrays which have been common in specialized research and industrial applications over the last two decades. None of the algorithms cited address the main challenges/opportunities for these newer systems: very heavy branch miss penalties but with software branch prediction, weaker out-of-order execution and multiple dispatch, but rich SIMD parallelism.

Our approach is to define a vocabulary to enable communication between the programmer and the instruction scheduler. Another approach, [1], is to move software pipelining into the domain of the programmer as source-to-source transformations. This approach can be useful, but restricts the discussion to features present in the high-level language, which make it impossible to access novel hardware features.

Other researchers have addressed the efficient use of SIMD parallelism in loops [3], they solved the problem of inter-quadword alignment which is common to all SIMD implementations, especially when parallelizing scalar code. In our approach, code generators would handle these aspects of parallelization. MultiLoop transformations are solving the orthogonal issue of optimizing control flow.

6 Conclusion

Through detailed analysis of four very different examples of numerical computation, we have shown that generators for MultiLoops can eliminate branch miss-prediction and leverage SIMD parallelism to reduce the cost of loop overhead. In addition to enabling significant improvements in code size and/or performance, our descriptions are well-structured, encapsulating complicated SIMD synthesized control flow, and facilitating discussions of high-level transformations with low-level implications.

Our next steps are: a formalization of the specification and transformations, incorporation of symbolic computation to prove required properties of scheduled code, and extension of this framework to inter-SPU communication, which will involve concurrent control-flow arrangements.

Acknowledgements

The authors would like to thank IBM Canada, NSERC, CFI and OIT for research grants which supported this work.

About the Authors

Christopher Anand is an Assistant Professor in the Department of Computing and Software. He is motivated to develop efficient code generation for safety-critical high-performance software systems by his is experience writing Magnetic Resonance Image reconstruction software

at Picker Medical Systems. His Internet address is anandc@mcmaster.ca.

Wolfram Kahl is an Associate Professor in the Department of Computing and Software at McMaster University. He is interested in applications and foundations of, and tool support for formal methods in software development, in particular high-level specification formalisms and programmig languages, graph transformation, and relation-algebraic calculations. His Internet address is kahl@cas.mcmaster.ca.

References

[1] Yosi Ben-Asher and Danny Meisler. Towards a source level compiler: Source level modulo scheduling. In *2006 Intl. Conf. on Parallel Processing Workshops (ICPPW'06)*, pages 298–308. IEEE Computer Society, 2006.

[2] International Business Machines Corporation, Sony Computer Entertainment Incorporated, and Toshiba Corporation. *Cell Broadband Engine Programming Handbook*. IBM Systems and Technology Group, Hopewell Junction, NY, 1.0 edition.

[3] Alexandre E. Eichenberger, Kathryn O'Brien, Kevin O'Brien, Peng Wu, et al. Optimizing compiler for the cell processor. In *PACT '05: Proc. 14th Intl. Conf. Parallel Architectures and Compilation Techniques*, pages 161–172. IEEE Computer Society, 2005.

[4] R. Govindarajan, N. S. S. Narasimha Rao, E. R. Altman, and Guang R. Gao. Enhanced co-scheduling: A software pipelining method using modulo-scheduled pipeline theory. *Int. J. Parallel Program.*, 28(1):1–46, 2000.

[5] IBM. *PowerPC Microprocessor Family: Vector/SIMD Multimedia Extension Technology Programming Environments Manual*. IBM Systems and Technology Group, Hopewell Junction, NY, 2005.

[6] IBM. *Synergistic Processor Unit Instruction Set Architecture*. IBM Systems and Technology Group, Hopewell Junction, NY, October 2006.

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

[8] J. A. Kahle, N. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the cell multiprocessor. *IBM Journal of Research and Development*, 49(4/5):589–604, July/September 2005.

[9] Monica S. Lam. Software pipelining: an effective scheduling technique for vliw machines. *SIGPLAN Not.*, 39(4):244–256, 2004.

[10] Tanya M. Lattner. An implementation of swing modulo scheduling with extensions for superblocks. Master's thesis, University of Illinois at Urbana-Champaign, 2005.

[11] Josep Llosa. Swing modulo scheduling: A lifetime-sensitive approach. In *PACT '96: Proc. 1996 Conf. Parallel Architectures and Compilation Techniques*, page 80, Washington, DC, USA, 1996. IEEE Computer Society.

[12] R.M. Ramanathan. *Extending the World's Most Popular Processor Architecture, New innovations that improve the performance and energy efficiency of Intel architecture.* Intel Corporation, 2006.

[13] SangMin Shim and Soo-Mook Moon. Split-path enhanced pipeline scheduling. *IEEE Trans. Parallel Distrib. Syst.*, 14(5):447–462, 2003.

[14] Mark G. Stoodley and Corinna G. Lee. Software pipelining loops with conditional branches. In *MICRO 29: Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, pages 262–273, Washington, DC, USA, 1996. IEEE Computer Society.

[15] Bogong Su, Shiyuan Ding, Jian Wang, and Jinshi Xia. Gurpr—a method for global software pipelining. In *MICRO 20: Proc. 20th Annual Workshop on Microprogramming*, pages 88–96. ACM Press, 1987.

[16] Bogong Su and Jian Wang. Gurpr*: a new global software pipelining algorithm. In *MICRO 24: Proc. 24th Annual Intl Symp. on Microarchitecture*, pages 212–216. ACM Press, 1991.

[17] Wolfgang Thaller. Explicitly staged software pipelining. Master's thesis, McMaster University, Department of Computing and Software, 2006. http://sqrl.mcmaster.ca/~anand/papers/ThallerMScExSSP.pdf.

[18] Nancy J. Warter, Grant E. Haab, Krishna Subramanian, and John W. Bockhaus. Enhanced modulo scheduling for loops with conditional branches. In *MICRO 25: Proceedings of the 25th annual international symposium on Microarchitecture*, pages 170–179, Los

Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[19] Nancy J. Warter-Perez and Noubar Partamian. Modulo scheduling with multiple initiation intervals. In *MICRO 28: Proceedings of the 28th annual international symposium on Microarchitecture*, pages 111–119, Los Alamitos, CA, USA, 1995. IEEE Computer Society Press.