Reprinted from the

Proceedings of the GCC Developers' Summit

June 2nd–4th, 2004
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
Stephanie Donovan, *Linux Symposium*
C. Craig Ross, *Linux Symposium*

Review Committee

Eric Christopher, *Red Hat, Inc.*
Janis Johnson, *IBM*
Toshi Morita, *Renesas Technologies*
Zack Weinberg, *CodeSourcery*
Al Stone, *Hewlett-Packard*
Richard Henderson, *Red Hat, Inc.*
Andrew Hutton, *Steamballoon, Inc.*
Gerald Pfeifer, *SuSE, GmbH*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*

Swing Modulo Scheduling for GCC

Mostafa Hagog
IBM Research Lab in Haifa
mustafa@il.ibm.com

Ayal Zaks
IBM Research Lab in Haifa
zaks@il.ibm.com

Abstract

Software pipelining is a technique that improves the scheduling of instructions in loops by overlapping instructions from different iterations. Modulo scheduling is an approach for constructing software pipelines that focuses on minimizing the cycle count of the loops and thereby optimize performance. In this paper we describe our implementation of Swing Modulo Scheduling in GCC, which is a Modulo Scheduling technique that also focuses on reducing register pressure. Several key issues are discussed, including the use and adaptation of GCC's machine-model infrastructure for scheduling (DFA) and data-dependence graph construction. We also present directions for future enhancements.

1 Introduction

Software pipelining is an instruction scheduling technique that exploits instruction level parallelism found in loops by overlapping successive iterations of the loop and executing them in parallel. The key idea is to find a pattern of operations (named the kernel code) that when iterated repeatedly, produces the effect that an iteration is initiated before previous ones have completed [3]. Modulo scheduling is a technique for implementing software pipelining. It does so by first estimating a lower bound on the number of cycles it takes to execute the loop. This number is called the *Initiation Interval* — II, and the bound is called a *Minimum II* — MII (see example in Figure 1). Then it tries to place the instructions of the loop in II cycles, while taking into account the machine resource constraints and the instruction dependencies. In case the loop couldn't be scheduled in II cycles it tries with larger II until it succeeds.

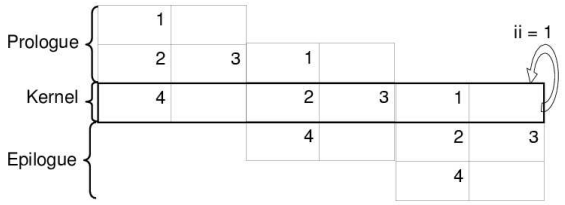

Figure 1: Example software pipelined loop of 4 instructions and the resulting kernel, prologue and epilogue.

Swing Modulo Scheduling (SMS) is a heuristic approach that aims to reduce register pressure [2]. It does so by first ordering the instructions in an alternating up-and-down order according to the data dependencies, hence its name (see section 2.2). Then the scheduling algorithm (section 2.3) traverses the nodes in the given order, trying to schedule dependent instructions as close as possible and thus to

shorten live ranges of registers.

2 Implementation in GCC

Swing Modulo Scheduling (SMS) [2, 3] was implemented as a new pass in GCC that immediately precedes the first scheduling pass. An alternative is to perform SMS after register allocation, but that would require register renaming and spilling in order to remove anti-dependencies and free additional registers for the loop. The new pass traverses the current function and performs SMS on loops. It generates a new schedule for the instructions of the loop according to the SMS algorithm, which is "near optimal" in utilizing hardware resources and register pressure. It also handles long live ranges and generates prologue and epilogue code as we describe in this section.

The loops handled by SMS obey the following constraints: (1) The number of iterations of the loop is known before entering the loop (i.e. is loop-invariant). This is required because when we exit the kernel, the last few iterations are in-flight and need to be completed in the epilogue. Therefore we must exit the kernel a few iterations before the last (or support speculative partial execution of a few iterations past the last). (2) A single basic block loop. For architectures that support predicated instructions, multiple basic block loops could be supported.

For each candidate loop the modulo scheduler builds a data-dependence graph (DDG), whose nodes represent the instructions and edges represent intra- and inter-loop dependences. The modulo scheduler then performs the following steps when handling a loop:

1. Calculate a MII.
2. Determine a node ordering.
3. Schedule the kernel.
4. Perform modulo variable expansion.
5. Generate prolog and epilog code.
6. Generate a loop precondition if required.

After a loop is successfully modulo-sceduled it is marked to prevent subsequent rescheduling by the standard instruction scheduling passes. Only the kernel is marked; the prolog and epilog are subject to subsequent scheduling.

Subsection 3.1 describes the DDG. In the remainder of this section we elaborate each of the above steps.

2.1 Calculating a MII

The minimum initiation interval ("MII") is a lower-bound on the number of cycles required by any feasible schedule of the kernel of a loop. A schedule is feasible if it meets all dependence constraints with their associated latencies, and avoids all potential resource conflicts. Two separate bounds are usually computed—one based on recurrence dependence cycles ("recMII") and the other based on the resources available in the machine and the resources required by each instruction ("resMII") [6]:

$$\text{MII} = \max\{\text{recMII}, \text{resMII}\}.$$

In general, if the computed MII is not an integer, loop unrolling can be applied to possibly improve the scheduling of the loop. The purpose of computing MII is to avoid trying II's that are too small, thereby speeding-up the modulo scheduling process. It is not a correctness issue, and being a lower bound does not affect the resulting schedule.

The "recMII" lower bound is defined as the maximum, taken over all cycles C in the dependence graph, of the sum of latencies along

C divided by the sum of distances along C:

$$\text{recMII} = \max_{C \in DDG} \frac{\sum_{e \in C} \text{latency}(e)}{\sum_{e \in C} \text{distance}(e)}.$$

Computing the maximum above can be done in $\Theta(N^3)$ (worst and best) time, where N is the number of nodes in the dependence graph [6]. We chose to implement a less accurate yet generally more efficient computation of a dependence-recurrence based lower bound, focusing on simple cycles S that contain a single back-arc $b(S)$ (more complicated cycles are ignored, resulting in a possibly smaller lower bound):

$$\text{recMII'} = \max_{S \in DDG} \frac{\sum_{e \in S} \text{latency}(e)}{\text{distance}(b(S))}.$$

(Note that for such simple cycles S, $\text{distance}(e) = 0$ for all edges $e \in S$ except $b(S)$.) This maximum is computed by finding for each back-arc $b(S) = (h, t)$ the longest path (in terms of total latency) from t to h, excluding back-arcs (i.e. in a DAG). This scheme should be more efficient because the number of back-arcs is anticipated to be relatively small, and is expected to suffice because we anticipate most recurrence cycles to be simple.

The "resMII" is currently computed by considering issue constraints only: the total number of instructions is divided by the ISSUE_RATE parameter. This bound should be improved by considering additional resources utilized by the instructions.

In addition to the MII lower-bound, we also compute an upper-bound on the II, called *MaxII*. This upper-bound is used to limit the search for an II to effective values only, and also to reduce compile-time. We set MaxII= $\sum_{e \in DDG} \text{latency}(e)$ (the standard instruction scheduler should achieve such an II), and provide a factor for tuning it (see Section 5).

2.2 Determining a Node Ordering

The goal of the "swinging" order is to schedule an instruction after scheduling its predecessor or successor instructions and as close to them as possible in order to shorten live ranges and thereby reduce register pressure. Alternative ordering heuristics could be supported in the future. (See figure 7 [1] for the swing ordering algorithm).

The node ordering algorithm takes as input a data dependence graph, and produces as output a sorted list of the nodes of the graph, specifying the order in which to list-schedule the instructions. The algorithm works in two steps. First, we construct a partial order of the nodes by partitioning the DDG into subsets $S_1, S2, \ldots$ (each subset will later be ordered internally) as follows:

1. Find the SCC (Strongly Connected Component)/Recurrence of the data-dependence graph having the largest recMII—this is the first set of nodes S_1.
2. Find the SCC with the next largest recMII, put its nodes into the next set S_2.
3. Find all nodes that are on directed paths from any previous set to the next set S_2 and add them to the next set S_2.
4. If there are additional SCCs in the dependence graph goto step 2. If there are no additional SCCs, create a new (last) set of all the remaining nodes.

The second step orders the nodes within each S_i set using a directed-acyclic subgraph of the DDG obtained by disregarding back-arcs of S_i:

1. Calculate several timing bounds and properties for each node in the dependence

graph (earliest/latest times for scheduling according to predecessors/successors—see subsection 4.1 [1]).

2. Calculate the order in which the instructions will be processed by the scheduling algorithm using the above bounds.

2.3 Scheduling the Kernel

The nodes are scheduled for the kernel of the loop according to the precomputed order. Figure 2 shows the pseudo code of the scheduling algorithm, and works as follows. For each node we calculate a scheduling window—a range of cycles in which we can schedule the node according to already scheduled nodes. Previously scheduled predecessors (PSP) increase the lower bound of the scheduling window, while previously scheduled successors (PSS) decrease the upper bound of the scheduling window. The cycles within the scheduling window are not bounded a-priori, and can be positive or negative. The scheduling window itself contains a range of at-most II cycles. After computing the scheduling window, we try to schedule the node at some cycle within the window, while avoiding resource conflicts. If we succeed we mark the node and its (absolute) schedule time. If we could not schedule the given node within the scheduling window we increment II, and start over again. If II reaches an upper bound we quit, and leave the loop without transforming it.

If we succeed in scheduling all nodes in II cycles, the register pressure should be checked to determine if registers will be spilled (due to overly aggressive overlap of instructions), and if so increment II and start over again. This step has not been implemented yet.

During the process of scheduling the kernel we maintain a *partial schedule*, that holds the scheduled instructions in II *rows*, as follows: when scheduling an instruction in cycle T (inside its scheduling window), it is inserted into row $(T \bmod II)$ of the partial schedule. Once all instructions are scheduled successfully, the partial schedule supplies the order of instructions in the kernel.

A modulo scheduler (targeting e.g. a superscalar machine) has to consider the order of instructions within a row, when dealing with the start and end cycles of the scheduling window. When calculating the start cycle for instruction i, one or more predecessor instructions p will have a tight bound $SchedTime_p + Latency_{p,i} - distance_{p,i} \times ii =$ start (see Figure 2). If p was itself scheduled in the start row, i has to appear after i in order to comply with the direction of the dependence. An analogous argument holds for successor instructions that have a tight bound on the end cycle. Notice that there are no restrictions on rows strictly between start and end. In most cases (e.g. targets with hardware interlocks) the scheduler is allowed to relax such tight bounds that involve positive latencies, and the above restriction can be limited to zero latency dependences only.

2.4 Modulo Variable Expansion

After all instructions have been scheduled in the kernel, some values defined in one iteration and used in some future iteration must be stored in order not to be overwritten. This happens when a life range exceeds II cycles—the defining instruction will execute more than once before the using instruction accesses the value. This problem can be solved using modulo variable expansion, which can be implemented by generating register copy instructions as follows (certain platforms provide such support in hardware, using rotating-register capabilities):

1. Calculate the number of copies needed for a given register defined at cycle T_def and

```
ii = MII; bump_ii = true;
ps = create_ps (ii, G, DFA_HISTORY);
while (bump_ii && ii < maxii){
 bump_ii = false;sched_nodes = φ;
           step = 1;
 for (i=0, u=order[i];
      i<|G|; u=order[++i]) do {
  /*Compute sched window for u.*/
  PSP = u_preds ∩ sched_nodes;
  PSS = u_succs ∩ sched_nodes;
  if (PSP ≠ φ ∧ PSS = φ){
   start=max(SchedTime_v + Latency_v,u
             −distance_v,u × ii)∀v ∈ PSP
   end = start + ii;
  }
  else if (PSP = φ ∧ PSS ≠ φ){
   start=min(SchedTime_v − Latency_u,v
             +distance_u,v × ii)∀v ∈ PSS
    end = start - ii; step = -1;
  }
  else if (PSP ≠ φ ∧ PSS ≠ φ){
   estart=max(SchedTime_v + Latency_v,u
             −distance_v,u × ii)∀v ∈ PSP
   lstart=min(SchedTime_v − Latency_u,v
             +distance_u,v × ii)∀v ∈ PSS
   start = max(start, estart);
   end = min(estart+ii, lstart+1);
  }
  else /* PSP = φ ∧ PSS = φ */
   start = ASAP_u; end = start + ii;

  /* Try scheduling u in window.  */
  for (c = start; c != end; c += step)
   if (ps_add_node (ps, u, c)){
    SchedTime_u = c;
    sched_nodes = sched_nodes ∪ {u};
    success = 1;
   }
  if (!success){
   ii++; bump_ii = true;
   reset_partial_schedule (ps, ii);
  }
 }/* Continue with next node.  */
 if (!bump_ii
     &&check_register_pressure(ps)){
  ii++; bump_ii = true;
  reset_partial_schedule (ps, ii);
 }
}/* While bump_ii.  */
where:  ASAP_u is the earliest time u
        could be scheduled in[2]
```

Figure 2: Algorithm for Scheduling the Kernel

used at cycle T_use, according to the following equation:

$$\left\lfloor \frac{T_{\text{use}} - T_{\text{def}}}{\text{II}} \right\rfloor + \text{adjustment} \quad (1)$$

where "adjustment" = -1 if the use appears before the def on the same row in the partial schedule, and zero otherwise. The total number of copies needed for a given register def is given by the last use.

2. Generate the register copy instructions needed, in reverse order preceeding the def:

$$r_n \leftarrow r_{n-1}; r_{n-1} \leftarrow r_{n-2}; \ldots r_1 \leftarrow r_{def}$$

and attach each use to the appropriate r_m copy.

2.5 Generating Prolog and Epilog

The kernel of a modulo-scheduled loop contains instances of instructions from different iterations. Thus a prolog and an epilog (unless all moves are speculative) are needed to keep the code correct.

When generating the prolog and epilog, special care should be taken if the loop bound is not known. One possibility is to add an exit branch out of each iteration of the prolog, targeting a different epilog. This is complicated and increases the code size (see [1]. Another approach is to keep an original copy of the loop to be executed if the loop-count is too small to reach the kernel, and otherwise execute a branch-less prolog followed by the kernel and a single epilog. We implemented the latter because it is simpler and has smaller impact on code size.

3 Infrastructure Requirements for Implementing SMS in GCC

The modulo scheduler, being a scheduling optimization, needs to work with a low level representation close to the final machine code. In GCC that is RTL. The SMS algorithm requires several building blocks from the RTL representation:

1. Identifying and representing RTL level loops—we use the CFG representation.
2. Building data dependence graph (for loops) with loop carried dependencies—we implemented a Data Dependence Graph (DDG).
3. An ordered linked list of instructions (exists in the RTL). Union, intersection, and subtraction operations on sets of instructions—we use the `sbitmap` representation.
4. Machine resource model support, mainly for checking if a given instruction will cause resource conflicts if scheduled at a given cycle/slot of a partial schedule.
5. Instruction latency model—we use the `insn_cost` function.

We now describe the DDG and Machine model support.

3.1 Data Dependence Graph (DDG) Generation

The current representation of data dependencies in GCC does not meet the requirements for implementing modulo scheduling; it lacks inter-loop dependencies and it is not easy to use. We decided to implement a DDG, which provides additional capabilities (i.e. loop carried dependencies) and modulo-scheduling oriented API.

The data dependence graph is built in several steps. First, we construct the intra-loop dependencies using the standard LOG_LINKS/INSN_DEPEND structures by calling the sched_analyze function of haifa-sched.c module; a dependence arc with distance zero is then added to the DDG for each INSN_DEPEND link. We then calculate inter-loop register dependencies of distance 1 using the df.c module as follows:

1. The latency between two nodes is calculated using the `insn_cost` function of the scheduler.
2. For each downwards reaching definition, if there is an upwards reaching use of the same register (this information is supplied by the df analysis) a TRUE dependence arc is added between the def and the use.
3. For each downwards reaching definition find its first definition and connect them by an OUTPUT dependence, if they are distinct. Avoid creating self OUTPUT dependence arcs.
4. For each downwards reaching use find its first def, if this is not the def feeding it (intra-loop) add an ANTI inter-loop dependence. Avoid creating inter-loop ANTI register dependences—modulo variable expansion will handle such cases (see 2.4). FLOW dependence exists in the opposite direction;

Finally, we calculate the inter-loop memory dependencies. Currently, we are over conservative due to limitation of alias analysis. This issue is expected to be addressed in the future. The current implementation adds the following dependence arcs, all with distance 1 (unless the nodes are already connected with a dependence arc of distance 0):

1. For every two memory writes add an inter-loop OUTPUT dependence.

2. For every memory write followed by a memory read (across the back-arc) add a TRUE memory dependence.

3. For every memory read followed by a memory write across the back-arc add an ANTI memory dependence.

The following general functionality is provided by the DDG to support the node-ordering algorithm of SMS:

- Identify cycles (strongly connected components) in the data dependence graph, and sort them according to their recMII.

- Find the set of all predecessor/successor nodes for a given set of nodes in the data dependence graph.

- Find all nodes that lie on some directed path between two strongly connected subgraphs.

3.2 Machine Resource Model Support

During the process of modulo scheduling, we need to check if a given instruction will cause resource conflicts if scheduled at a given cycle/slot of a partial schedule. The DFA-based resource model in GCC [4] works by checking a sequence of instructions, in order. This approach is suitable for cycle scheduling algorithms, in which instructions are always appended at end of the current schedule. In order for SMS to use this linear approach, we generate a trace of instructions cycle by cycle, centered at the candidate instruction, and feeding it to the DFA [5]. Figure 3 describes the algorithm that checks if there are conflicts in a given partial schedule around a given cycle. Several functions are made available to manipulate the partial schedule, the most important one is *ps_add_node_check_conflicts* described in Figure 4; it updates the partial schedule (tentatively) with a new instruction at a given cycle, and feeds the new partial schedule to the DFA. If it succeeds it updates the partial schedule and returns success, if not it resets the partial schedule and returns failure. The major drawback of the above mechanism is the increase in compile time; there are plans to address this concern in the future.

```
/* Checks if PS has resource
   conflicts according to DFA,
   from FROM cycle to TO cycle.  */
ps_has_conflicts (ps, from, to){
  state_reset (state);
  for (c = from; c <= to; c++) {
    /* Holds the remaining issue
       slots in the current row.  */
    issue_more = issue_rate;
    /* Walk DFA through CYCLE C.  */
    for (I = ps->rows[c % ps->ii)];
         I; I = I->next) {
      /* Check if there is room for the
         current insn I.*/
      if (! issue_more
          || state_dead_lock_p (state))
        return true;
      /* Check conflicts in DFA.*/
      if (state_transition (state, I))
        return true;
      if (DFA.variable_issue)
        issue_more=DFAissue(state, I);
      else issue_more--;
    }
    advance_one_cycle ();
  }
  return false;
}
```

Figure 3: Feeding a partial schedule to DFA.

4 Current status and future enhancements

An example of a loop and its generated code, when compiled with gcc and SMS enabled (`-fmodulo-sched`) is shown in Figure 5. The kernel combines the fmadds of the current iteration with the two lfsx's of the next iteration. As a result, the two lfsx's appear in

```
/* Checks if a given node causes
   resource conflicts when added to
   PS at cycle C.  If not add it.  */
ps_add_node_check_conflicts (ps, n, c)
{
  ps_n = add_node_to_ps (ps, n, c);
  from = c - ps->history;
  to = c + ps->history
  has_conflicts
    = ps_has_conflicts(ps, from, to);

  /* Try different slots in row.  */
  while (has_conflicts)
    if (!ps_insn_advance_column(ps,
                                ps_n))
      break;
    else has_conflicts
      = ps_has_conflicts(ps,
                         from, to);

  if (! has_conflicts)
    return ps_n;
  remove_node_from_ps(ps, ps_n);
  return NULL;
}
```

Figure 4: Add new node to partial schedule

the prolog and the fmadds appears in the epilog. This could help hide the latencies of the loads. The count of the loop is decreased to 99, and no register-copies are needed because every life range is smaller than II,

Following are milestones for implementing SMS in GCC.

First stage (Approved for mainline)

1. Implement the required infrastructure: DDG (section 3.1), special interface with DFA (section 3.2).
2. Implement the SMS scheduling algorithm as described in [3, 2].
3. Support only distance 1 and register carried dependences (including accumulation).

```
float dot_product (float *a,
                   float *b){
  int i; float c=0;
  for (i=0; i < 100; i++)
     c += a[i]*b[i];
  return c;
}
            (a)
L5:
        slwi r0,r2,2
        addi r2,r2,1
        lfsx f13,r4,r0
        lfsx f0,r3,r0
        fmadds f1,f0,f13,f1
        bdnz L5
        blr
            (b)

Prolog: addi r2,r2,1
        lfsx f0,r3,r0
        lfsx f13,r4,r0
        li r0,99
        mtctr r0
L5:
        slwi r0,r2,2
        addi r2,r2,1
        fmadds f1,f0,f13,f1
        lfsx f13,r4,r0
        lfsx f0,r3,r0
        bdnz L5
Epilog: fmadds f1,f0,f13,f1
        blr
           (c)
```

Figure 5: (a) An example C loop, (b) Assembly code without SMS, (c) Assembly code with SMS (`-fmodulo-sched`), on a PowerPC G5.

4. Support for live ranges that exceed II cycles by register copies.
5. Support unknown loop bound using loop preconditioning.
6. Prolog and epilog code generation as described in Section 2.5.
7. Preliminary register pressure measurements—gathering statistics.

Second stage

1. Support dependences with distances greater than 1.
2. Improve the interface to DFA to decrease compile time.
3. Support for live ranges that exceed II cycles by unroll & rename.
4. Improve register pressure heuristics/measurements.
5. Improve computation of resMII and possibly recMII lower bounds.
6. Unroll the loop if tight MII bound is a fraction.

Future enhancements [tentative list]

1. Consider changes to DFA to make SMS less time consuming when checking resource conflicts.
2. Consider spilling during SMS if register pressure rises too much.
3. Support speculative moves.
4. Support predicated instructions and if-conversion.
5. Support for live ranges that exceed II cycles by rotating registers (for appropriate architectures).

5 Compilation Flags for Tuning

We added the following four options for tuning SMS:

sms-max-ii-factor. This parameter is used to tune the SMS_MAX_II threshold, which affects the upper bound for II (maxII). The default value for this parameter is 100. Decreasing this value will allow modulo scheduling to transform only the loops where a relatively small II can be achieved.

sms-dfa-history. The number of cycles considered when checking conflicts using the DFA interface. The default value is 0, which means that only potential conflicts between instructions scheduled in the same cycle are considered. Increasing this value may result in higher II (possibly less loops will be modulo scheduled), longer compile-time, but potentially less hazards.

sms-loop-average-count-threshold. A threshold on the average loop count considered by the modulo scheduler; defaults to 0. Increasing this value will result in applying modulo scheduling to additional loops, that iterate on average fewer times.

max-sms-loop-number. Maximum number of loops to perform modulo scheduling, mainly for debugging (search for first faulty loop). The default is -1 which means to consider all relevant loops.

6 Conclusions

In this paper we described our implementation of Swing Modulo Scheduling in GCC. An example of its effects is given in Section 4. The major challanges involved using the DFA-based machine model of GCC, and building a data-dependence graph for loops including inter-loop dependences. The current straightforward usage of the machine model is time-consuming and should be improved, which involves changes to the machine model. The inter-loop dependencies of the DDG should be built more accurately, in-order to allow more aggressive movements by the modulo scheduler. The DDG is general and can be used by other optimizations as well. Additional opportunities for improving and tuning the modulo scheduler exist, including register pressure and loop unrolling considerations.

7 Acknowledgments

One of the key issues in implementing SMS in GCC is the ability to use DFA resource modeling. We would like to thank Vladimir Makarov for helping us understand the DFA interface, the fruitful discussions that led to extending the DFA interface, and his assistance in discussing and reviewing the implementation of SMS in GCC.

References

[1] Josep Llosa Stefan M. Freudenberger. Reduced code size modulo scheduling in the absence of hardware support. In *Proceedings of the 35th Annual International Symposium on Microarchitecture (MICRO-35)*, November 2002.

[2] J. Llosa, A. Gonzalez, E. Ayguade, and M. Valero. Swing modulo scheduling: A lifetime sensitive approach. In *Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques (PACT '96)*, pages 80–87, Boston, Massachusetts, USA, October 1996.

[3] J. Llosa, A. Gonzalez, E. Ayguade, M. Valero, and J. Eckhardt. Lifetime-sensitive modulo scheduling in a production environment. *IEEE Trans. on Comps.*, 50:3, March 2001.

[4] Vladimir N. Makarov. The finite state automaton based pipeline hazard recognizer and instruction scheduler in gcc. In *Proceedings of the GCC Developers Summit*, May 2003.

[5] Vladimir N. Makarov. Personal communication. 2004.

[6] B.R. Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proceedings of the 27th Annual International Symposium on Microarchitecture (MICRO-27)*, November 1994.