Reprinted from the

Proceedings of the GCC Developers' Summit

June 28th–30th, 2006
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon Incorporated*
C. Craig Ross, *Linux Symposium*

Review Committee

Ben Elliston, *IBM*
Janis Johnson, *IBM*
Mark Mitchell, *CodeSourcery*
Toshi Morita
Diego Novillo, *Red Hat*
Gerald Pfeifer, *Novell*
Ian Lance Taylor, *Google*
C. Craig Ross, *Linux Symposium*
Andrew J. Hutton, *Steamballoon Incorporated*

Proceedings Formatting Team

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

An interblock VLIW-targeted instruction scheduler for GCC

Andrey Belevantsev
ISP RAS
abel@ispras.ru

Maxim Kuvyrkov
ISP RAS
mkuvyrkov@ispras.ru

Vladimir Makarov
Red Hat
vmakarov@redhat.com

Dmitry Melnik
ISP RAS
dm@ispras.ru

Dmitry Zhurikhin
ISP RAS
zhur@ispras.ru

Abstract

Modern VLIW architectures (e.g. IA-64) require instruction level parallelism (ILP) to be explicitly exposed by a compiler. An instruction scheduler is a key compiler component for utilizing ILP. The current GCC scheduler has a number of pitfalls in approaching this goal, including the oldest interblock scheduling algorithm (whose weakness is in prevention of instruction cloning), non-optimal region formation, a traditional two-pass execution scheme (before and after register allocation), and lack of transformations for eliminating false dependencies (e.g. register renaming and forward substitution).

In this paper, we present an approach for implementing new aggressive instruction scheduler for GCC. The scheduling algorithm is inspired with selective scheduling and resource-constrained software pipelining approaches. It is mainly targeted for VLIW-like platforms, but the framework being implemented is general enough and it can be used for other targets in the future. The key features of the approach are as follows: works with DAG regions, supports code motion with adding bookkeeping insns, supports register renaming and forward substitution, and integrates with software pipelining. We discuss the algorithm and its adaptation to GCC, implementation issues, and the current state of the project.

1 Introduction

Current GCC instruction scheduler has a long history. Integration of the Haifa scheduler has started in August 1997, when the first version of `haifa-sched.c` appeared in CVS repository. Since then, the scheduler's source code was significantly cleaned up[1], and new features (such as scheduling of extended basic blocks and DFA scheduling) were added. Nevertheless, the scheduler doesn't work well for the modern architectures, which require exposing instruction-level parallelism during compilation. These are the EPIC architecture [EPIC] and various VLIW embedded systems. The

[1] Still, some bits of the scheduler were integrated in GCC only recently. For example, using GCC edge probabilities instead of scheduler's own evaluation is supported only since February 2006.

major pitfalls of the current scheduler are as follows:

- The oldest interblock scheduling approach. The Haifa scheduler is a variant of dominator-path based scheduling, which appeared first in 1992. It tends to improve performance on the critical path sacrificing other paths in a region. Conceptually better approach should try to improve performance on all paths through the region. The second problem with the approach is that it doesn't support instruction cloning, i.e. code motion is allowed only when creating bookkeeping copies to preserve program correctness is not necessary. Moving up branches is also impossible without instruction cloning.

- Interblock scheduling is placed before register allocator and reload passes. This leads to the following disadvantages: firstly, scheduled insns could be later significantly changed (especially by reload); second, a single instruction before reload could be later splitted into several target insns, so before reload the scheduler works with inaccurate model of pipeline hazards.

- Lack of instruction transformations that eliminate nontrue dependencies. The most important such transformations are partial register renaming and forward substitution. Other examples are predication (which turns control dependencies into data dependencies) and instruction mutation (which turns e.g. shift insn into multiply insn when this allows to execute more insns on the current cycle). It should be noted that register renaming and forward substitution are essential to achieve good results when a scheduler is placed after register allocation.

Since the Haifa scheduler implements code motion without copies, it doesn't support creation of basic blocks or instructions during scheduling. On the contrary, adding support for instruction cloning and other transformations requires the infrastructure for manipulation of instructions and basic blocks. During our project on implementing speculation support in the scheduler we have found that creating such an infrastructure inside the Haifa scheduler is hard and requires a lot of hacking. Besides this, it is desirable to have a framework for evaluating instruction transformations, which should allow adding/removing and enabling/disabling transformations. This is also not easy to achieve with the current implementation.

Based on the above considerations, the project on implementing new aggressive interblock instruction scheduler for GCC was started. The ultimate goal of the project is to create an infrastructure capable of scheduling arbitrary DAGs and supporting various instruction transformations, among which are instruction cloning, partial register renaming, forward substitution, code motion of branches, and unification. The scheduler infrastructure should support fine-grained control over transformations to allow e.g. enabling speculation support on ia64 and disabling instruction cloning on embedded targets. The other goal is to create an infrastructure for manipulating instructions and control flow graph during scheduling. Some of required functionality could be borrowed from the previous project on speculation support.

We have chosen selective scheduling [Moon97] as a basis, also using thoughts from resource-constrained software pipelining (RCSP) [Aiken95] and percolation scheduling [Nicolau85]. Selective scheduling provides a thorough summary of scheduling ideas also mentioned in RCSP and percolation scheduling approaches, so it serves as a great starting point. Its basic algorithms are significantly

reworked by us for the GCC implementation.

We are trying to follow evolutional approach in the implementation process. Basic pieces of the infrastructure that provide the simplest code motion features are implemented first, and then more instruction transformations are added. Our goal is to get the working scheduler as early as possible, and then keep it working after addition of every single transformation.

The project is an ongoing work started in September 2005. At the moment of this writing we are in the middle of the way. This paper describes the current status of the project focusing on the improvements we made to the basic algorithms. The rest of the paper is organized as follows. Section 2 provides an overview of the basic scheduling algorithms of the selective scheduling approach focusing on the key ideas we use for the GCC scheduler. Section 3 covers the most important implementation challenges and solutions that we designed for them. Our current progress is sketched in Section 4, while Section 5 concludes.

2 The basic algorithm

Selective scheduling approach contains the following key ideas of the interblock scheduler design: separation of the available insn computation and the actual code motion stages, formulation of the computation stage through simple propagation routines, incremental recomputation of the available insns, *partial* register renaming through scheduling *right-hand sides* (RHSes) instead of whole insns, representation of a VLIW insn as a tree for moving up branches. We use all these concepts in our implementation and explain them in the below subsections.

2.1 The main scheduling routines

The scheduler takes as an input an arbitrary part of the control flow graph that forms a DAG. The driver routine breaks the CFG onto several DAGs and schedules each of them separately via `schedule_region` routine, which contains the main scheduling loop. Each iteration of this loop tries to gather a *parallel group* of instructions at the currently scheduling point, and then to advance the point. A parallel group corresponds to a single VLIW instruction when targeting to a VLIW architecture, or to a regular instruction in other cases. The loop terminates when no more instructions are left for scheduling.

Filling a parallel group is handled by the `fill_group` routine. Its driver loop adds new instructions to the current parallel group until target resources and data dependencies permit. First, the set of available operations, or *av set*, is computed for the current scheduling point. Then the best operation is chosen and scheduled. Finally, the best operation is moved up to the scheduling point from its original location, possibly creating bookkeeping copies and updating av sets along its moving path (see Fig. 1).

The basic routines do not change when more transformations are added to the scheduler. Instead, the new functionality is incorporated into the computation, choosing, and code motion stages. For example, partial register renaming is done through scheduling RHSes instead of whole instructions. An instruction is eligible for register renaming when it is a store to a register, i.e. of the form `(set (reg) (rhs))`. For such an instruction, only its RHS participates in the scheduling process. The RHS is added to the av set, the choosing routine besides the best operation also finds a target register for the RHS, and the operation is scheduled

```
fill_group(insn) {
  group = create_empty_group ();

  while (1)
  {
    av_set = compute_av_set (insn);

    best_op = choose_best_op (av_set);
    if (best_op == NULL)
      break;

    schedule_op (best_op);
    move_op (insn, best_op);
    advance_scheduling_point (&insn);
  }

  return group;
}
```

Figure 1: The `fill_group` routine

as `best_reg = best_rhs`. Corresponding bookkeeping copies for this instruction are created during code motion in `move_op`. We will consider that some insns are scheduled as RHSes in the below subsections and will use a term *operation* to denote either an insn or an RHS.

2.2 Computation stage

The task of the computation stage is to gather all instructions available for scheduling along all execution paths. The simple way to do this is to traverse the DAG starting from current scheduling point in reverse topological order. When visiting an instruction (a graph node n), first a set of the insns available immediately after n is computed as a union of n's successors' av sets. Then this set is propagated through n by filtering out its elements that could not be moved up past n. Finally, n's operation is collected and added to the set:

$$avset(n) = moveup\_set(\bigcup_{x \in Succ(n)} avset(x)) \bigcup av\_op(n)$$

As the code motion stage invalidates the av set found, it should be recomputed after scheduling each single insn. The recomputation should be done incrementally to avoid high overhead. It can be noticed that after code motion av sets become invalid only along the moving path and could be restored using the valid sets from other basic blocks. Hence the key idea of the computation stage is to save the intermediate av sets at the beginning of each basic block to avoid recomputating the sets from scratch.

```
moveup_op(insn, op) {

  /* Ok to move if no dependence.  */
  if (!data_dep_between (insn, op))
    return op;

  /* Try substitution.  */
  if (true_dep (insn, op) && rhs_p (op)
      && copy_insn_p (insn))
    {
      dst = SET_DEST (insn);
      src = SET_SRC (insn);

      if (dst_is_in (op, dst))
        return substitute (op,
                           dst, src);
    }

  /* Can't do anything.  */
  return NULL;
}
```

Figure 2: The `moveup_op` propagation helper

The `moveup_set` routine filters its input set with the `moveup_op` helper, which determines whether the given operation could be propagated through the current insn. The `moveup_op` logic depends on the type of the code motion we want to support. When scheduling whole insns, it is enough to check for the data dependence between the two insns. However, when forward substitution is supported through RHS scheduling, we can do better. For example, `x+y` RHS could be moved before the `y=z` copy as `x+z`, i.e. true dependence

```
//current scheduling point
//best_op: z = b + c

if (...) {
  a = b;
} else {
  a = c;



}

z = b + c;
```

(a) Before the traversal

```
//current scheduling point
z = b + c;

if (...) {
  a = b;
} else {
  a = c;
  //bookkeeping copy
  z = b + c;
}
//found and deleted:
//z = b + c;
```

(b) After traversing 'then' path

```
//current scheduling point
z = b + c;

if (...) {
  a = b;
} else {
  a = c;
  //found and deleted:
  //z = b + c;
}
```

(c) After traversing 'then' and 'else' paths

Figure 3: Creating bookkeeping code

between these two operations can be eliminated with substitution. In this case the propagation helper will return the modified operation (see Fig. 2).

2.3 Code motion stage

When the av set is calculated, the scheduler chooses the best element of the set (either an instruction or an RHS) for moving into the current group. The task of choosing the best operation from the av set is orthogonal to the rest of the scheduler and usually is driven by implementation-dependent heuristics, so it is covered in the next section. Here we assume that the best operation `best_op` is chosen and now the task of the code motion stage is to actually move it up to the current scheduling point.

The code motion process is driven by the `move_op` routine. It traverses the DAG starting at the current scheduling point in search of the original operations (from which `best_op` could be derived). Let's assume first that we're scheduling only instructions, then it's enough to search just for `best_op`. When the operation is found, it is deleted from its original place and moved to the parallel group. Then the routine backtracks and continues the traversal. When backtracking along the already traversed code motion path, bookkeeping copies of `best_op` are inserted on edges that join the current moving path from outside. When the traversal explores other code motion path and sees already created bookkeeping copy, it is recognized as original operation and deleted in the same way (see Fig. 3). This allows to create only necessary bookkeeping code.

The process is more complicated when RHSes are also scheduled. It is not enough to search for the `best_op` because it could change through substitution. Hence when traversing through a copy instruction, we should "unsubstitute" `best_op` to reproduce its original form and add the resulting operation to the set of operations we're searching for. Back to our previous example, when `x+z` is the best operation and we're traversing through `y=z`, we don't know whether this operation was moved up earlier as `x+y` (through substitution) or as `x+z` (unchanged), thus we should search for both forms below the copy. To reduce the number of operations we should search for, the set of these operations is intersected with the av sets saved in basic blocks. This is possible because the available operations which can be found below a DDG node should be in its av set.

When the original operation is found, it is either removed or changed to a copy `old_dest=new_dest`, when register renaming is used. Then during backtracking the current form of the `best_op` at the node being tra-

versed should always be retained to allow correct creation of bookkeeping code.

2.4 Tree instruction

The convenient form for representing a parallel group (i.e. instructions that could be emitted at the same cycle) is a *tree* instruction (introduced in [Ebcioğlu88]). Each node of the tree instruction corresponds to a branch test, and leaf nodes correspond to instruction labels where their parent can branch to. Each edge of the tree instruction corresponds to sequential operations. Tree correctness and its execution semantics is determined by the target. For example, when the target is not capable of multiway branching, the tree would not contain any branch test nodes.

The concept of tree instruction is necessary for doing code motion of branches. In this case the computation stage takes branches into account allowing them to be propagated up to the next branch. The code motion stage analogously searches also for branches and creates bookkeeping copies of them. The `fill_group` routine is changed to emit the insns into the tree instruction. When a branch is scheduled and there's still place in the current parallel group, an extra scheduling point (or *boundary*) is created. The `compute_av` and `move_op` routines are called for each group boundary, thus allowing to schedule instructions at both targets of the branch.

3 The GCC implementation

The basic scheduling algorithms sketched in the previous section are reworked and somewhat improved for the GCC implementation. The key improvement that we make in our implementation is the use of data dependencies to avoid unnecessary computations during the computation stage. The idea here is that we don't want to collect operations that would anyway be filtered out during later traversal. Removing such operations as early as possible reduces the time needed for each update of the av sets, which happens on each scheduling iteration.

The second major implementation feature is handling a number of instruction transformations through annotating dependencies with an extra data. The data is used to decide whether the scheduler is able to break the particular dependence using the given transformation. For example, data speculation is handled by annotating dependencies with a *spec* flag (what kind of speculation should be used) and a *weakness* (how "probable" is this dependence); forward substitution is handled by placing substitutable and non-substitutable dependencies to different lists. The advantage of this approach is that when adding new transformation, it allows natural extension of the existing framework. Namely, the basic infrastructure remains untouched, and the only changes go to the dependence data structures and the propagation routines of computation and code motion stages.

Other implementation changes include computation of register liveness sets needed for register renaming and CFG/insn handling infrastructure for code motion. These changes are necessary to implement the concepts of the previous chapter in any real compiler. Besides those, we discuss implementation and target specific portions of the scheduler that are not covered in the basic approach, namely choosing the best instruction for scheduling and the best register for renaming, and region formation.

3.1 Input regions

The scheduling approach we use permits any kind of DAG regions as an input. The needed

functionality is to find whether a basic block or an edge belongs to the current region and the possibility to extend a region when new basic blocks are created. The former is used during region traversal, whereas the latter should be done during the code motion stage. At the moment of writing we use the region infrastructure from the Haifa scheduler extended with the patches for the ia64 speculation support [IA64Speculation]. In the future we consider to use the region finder implemented for GCC by Kenneth Zadeck [ZadeckRegions].

3.2 Data structures and their computation

There is a number of data structures needed by the scheduler that can be represented as linked lists: a path in a DAG, parallel group boundaries, fences[2], and av sets. These structures are backed up by a single linked list interface. Custom list types are implemented through this interface, and their data is accessed through a union contained in a list node analogously to `struct rtx_def`. This provides a uniform interface for accessing, manipulating, and iterating over the list-like data structures. All per-instruction data is stored in the `s_i_d` array indexed by `INSN_UID` analogously to the Haifa scheduler.

The most important role is played by the availability and liveness sets (av and lv sets). This is because the contents of av and lv sets should be valid on each scheduling iteration, and their initial computation and update is not trivial. The basic approach for computing av sets suggests to traverse a DAG collecting all operations on the way and filtering out those that can't be propagated through currently visiting node. This solution is not optimal: an operation collected from the bottom of the DAG can be moved up somewhere to the middle before it would be filtered out, wasting time for unnecessary propagation between the two nodes. An ideal solution here would be to traverse a dependence graph[3] instead of a DAG. However, this is not possible due to: a) control dependencies should be represented explicitly, which leads to the quadratic complexity of the algorithm, and b) a copy of av set should be left at the beginning of each basic block to allow quicker code motion. We are implementing the combined approach, i.e. the algorithm still traverses a DAG, but uses dependencies to decrease the number of computations needed when visiting a node.

3.2.1 Instruction dependencies and av sets

Instruction dependencies are split into two types—*hard* and *weak* ones—and placed to different lists accordingly.[4] Weak dependencies are those that can be broken by the scheduler, while hard ones are those that can't. Only true dependencies can be weak, while hard ones are {anti,output}-dependencies and all others (for example, created from `ASM`s or `SCHED_GROUP_P` insns). Weak dependencies that require different scheduling transformations to be broken (for example, either with substitution, if the insn is scheduled as a RHS, or with data speculation) can further be splitted onto several lists. The Haifa scheduler will see all dependencies as hard ones.

Maintaining several dependence lists helps to distinguish weak dependencies (which are of the most interest to the scheduler) among others. We can easily check whether an instruction has hard dependencies (not traversing the

[2] A *fence* is a point at which the next parallel group will be created and filled.

[3] Given that it contains both data and control dependencies.

[4] On the current GCC mainline this separation is implicit: the whole dependence list is maintained sorted, and weak dependencies are located after hard ones.

whole dependence list), or traverse and sort only weak ones. However, this comes at the expense of writing additional code to handle all dependencies at once.

Creation of dependencies and their separation by types happens in `sched-deps.c`. During the computation stage the node visiting algorithm looks as follows:

- When entering a node, first a union of node's successors' av sets is computed the same way as in the original approach, i.e. the visiting routine is called recursively.
- Let's denote node's operation as *op*. If *op* has any hard dependencies, it is not added to the av set. If *op* has only substitutable weak dependencies, then it's added to the set, and all *op*'s producers are notified that when they are visited, *op* should be substituted. This is achieved by maintaining additional list of operations that need substitution for each copy instruction. If *op* has no dependencies (or has weak speculative dependencies), it's also added to the list.
- When a copy node is being visited, we should check whether any pending substitutions should be performed on operations from the av set. As we maintain a separate list of such operations, we don't need to traverse the whole set searching for them. Instead, a substitution is performed for each *pend_op* from node's pending list, and all weak dependencies of node's *op* are added to *pend_op*'s dependencies. These dependencies also need to be checked the same way as in the previous step: if *op* has any hard dependencies, then all *pend_ops* should be removed from the av set.
- When the node visited is a basic block head, a copy of the intermediate av set is left at the node analogously to the original approach.

Consider Fig. 4 as an example. Node 4 doesn't have any hard dependencies, but has one weak substitutable dependence, so its operation can be scheduled as an RHS: `y+1` is added to the av set and to the pending ops of Node 2. Node 2 has no dependencies so it's safe to add it and all its pending ops to the av set. Before adding pending operations they should be substituted, so `y+1` becomes `z+1`.

```
(1) if cc0          av(1) = {z, z+1}
     /   \
      (2) y = z     pend_ops(2) = {y+1}
           \
      (3) if cc1    av(3) = {y+1}
           /
    (4) u = y+1     weak_dep(4) = {2},
                    is substitutable
```

Figure 4: Using dependencies during `compute_av`

3.2.2 Liveness sets

Liveness information is stored as `regsets` and computed using the scheme similar to the computation of av sets. The `compute_lv` driver also traverses a DAG and updates the intermediate lv sets. First, the intermediate lv set is calculated as a union of node's successors' lv sets. Then all registers that are set or clobbered by the visiting insn are subtracted from the lv set, and all registers that are used by the insn are added to the set. A copy of the intermediate lv set is saved at the beginning of each basic block to allow faster updating.

The sets of registers that are used, set, or clobbered by an instruction are calculated as a side effect of dependence analysis in `sched_deps.c`. The sets are the part of per-instruction dependence data structure `deps_`

`insn_data` and are stored in `d_i_d` array indexed by `INSN_LUID`. Initial lv sets are not computed from scratch but rather taken from `global_live_at_start` sets.

3.3 Choosing the best instruction

When the av set is computed, it's time to choose the best available operation from the set. This task is usually based completely on heuristics. The basic set of heuristics to use is taken from the Haifa scheduler, i.e. critical path, register pressure, control flow probability, etc. Data speculation adds to those the weakness degree of speculative dependence. The point that should be made here is that the role and weight of each heuristic is to be determined during evaluation and tuning of our implementation, which are yet in the future. As of now, we'd like to make several points regarding the difficulties in choosing the best operation that come from forward substitution and register renaming transformations.

When scheduling an RHS, there's a number of problems to be solved. First, a target register for storing the result of RHS should be chosen. This register should not be live along the moving path of the RHS from its original location(s) to the current scheduling point. This condition is not easy to check because during the choosing stage we don't know this moving path. The path is implicitly constructed during the code motion stage for the chosen operation, but before this we need to know this path for *all* available RHSes. Current implementation of this step is slow and actually performs the traversal similar to that of `move_op` for all members of the av set. The better way would be to calculate the sets of available registers for each RHS during the computation stage.

If original register of an RHS is available, then it's always the best choice. If not, then the best register is chosen in the way analogous to that of `regrename.c`. When no registers are available, the RHS cannot be scheduled. In the future, the logic for spilling a register in favor of using it for renaming should be implemented, namely using function saved registers for renaming.

The second problem is using the DFA pipeline descriptions for modeling the processor state. When scheduling an insn, DFA is used analogously to the Haifa scheduler, i.e. to check whether target resources permit scheduling of the insn on the current cycle. The first cycle multipass scheduling [Makarov03] will also be adapted from the Haifa scheduler to work directly on av sets. When scheduling an RHS, a valid insn should be formed first and then passed to the DFA interface. The possible optimization would be to extend the DFA interface to be able to handle RHSes at least for some cases.

The last problem is scheduling of multilatency operations. Fortunately, it could be solved with adding an extra dependence attribute called *tick*. A tick holds the simulated cycle number on which producer's result will be ready[5]. An instruction cannot be considered for scheduling until the current simulated cycle reaches a maximum from ticks of instruction's dependencies.

3.4 Code motion

The basic thing that is needed for the implementation of the code motion with support for instruction cloning is an interface for manipulating instructions and basic blocks. Two typical problems to be solved are creating bookkeeping insns and creating basic blocks for bookkeeping code. The code which solves those tasks and extends global data structures

[5] In the Haifa scheduler this information is represented as a per-instruction attribute called `INSN_TICK`.

of GCC is already in place. The extra code to be written is the proper extension of the data structures specific to the scheduler. This task for creating basic blocks for recovery code was solved in the ia64 speculation patch and would be taken from there.

The other point to make here is that one of the expensive parts of `move_op` is substitution/unsubstitution that should be done on the operations we are searching for. Fortunately, it is possible to make use of dependencies analogously to the computation stage. Substitution should be tried only for those copy insns that are found in the weak dependencies of the searched operations. It is also possible to save the original form of the operation during the computation stage and use it without performing actual substitution.

3.5 Target-specific details

A target affects the scheduling process through a number of hooks. Analogously to the Haifa scheduler the scheduler hooks can be used to override the choosing decisions and instruction costs. The new hooks we add are the hook to determine dependence type (for example, forbid creating of weak dependencies), the hook to disable scheduling an insn as an RHS, the hook to affect register choosing decisions, and the hooks for controlling speculation support similar to those of the ia64 speculation patch. The features of `cc0` targets (for example, scheduling the `cc0` user right after the setter) would be handled via dependence attributes. Target-specific reorganization passes such as bundling on ia64 would remain untouched. For this purpose, we will preserve the feature of marking the insns that start a new cycle with TImode. In the future, it is possible to consider integrating the bundling with the scheduler, but this doesn't seem to worth the trouble.

4 Current progress

We planned the implementation process in a number of stages. As noted above, we've tried to follow the evolutional approach and keep the working scheduler after each stage. The stages are as follows:

- Implement the basic infrastructure, i.e. the driver, computation and code motion routines. Do not allow any interblock code motions or any transformations.
- Add interblock motions without copies and adopt the DFA interface to the new infrastructure. The resulting scheduler should be similar to the Haifa's.
- Allow instruction cloning during scheduling. This step requires support for creating bookkeeping code and liveness checking analogous to the Haifa one.
- Allow forward substitution and register renaming. After completion of this step and some cleanup the code can be placed in the FSF branch.
- Support creation of hard and weak dependencies in `sched-deps.c`. Use created dependencies in computation and code motion stages.
- Implement a tree instruction support, which would allow implementing code motion of branches.

The current status of the project is exactly in the middle of this way. At the moment of writing (April 2006) support for instruction cloning is implemented (i.e., checking of liveness information and bookkeeping code creation), and all the pieces are being tested together. The scheduler works on x86 in place of the old sched2 pass. However, benchmarks were not yet run on any platform.

5 Conclusions

In this paper, we present an effort of implementing new aggressive interblock instruction scheduler for GCC. The project goal is to design and implement a scheduling framework that can be easily extended to support a number of instruction transformations, such as register renaming, forward substitution, instruction mutation. Implementing those transformations allows to place the scheduler after register allocation passes. Following newer approach than the Haifa scheduler would improve scheduling for modern architectures such as EPIC.

Implementing a new scheduler for GCC has to be a long project. Taking ideas of selective scheduling and RCSP as a start, we have added better use of data dependencies, additional transformations such as speculation, and more complicated register renaming in our implementation. Now we are in the middle of the way, and there's still a lot of things to be done. The major features that are yet to be implemented are tree instruction support, code motion of branches, better use of dependencies, and using function saved registers for renaming. The future work would be to implement software pipelining on top of the scheduler, which is quite natural with our approach.

6 Acknowledgments

We'd like to thank Vladimir Makarov from Red Hat for extensive consulting on this project. The insights to instruction scheduling and GCC world provided by Vlad are invaluable. We thank Diego Novillo, Daniel Berlin, and James Wilson for giving helpful comments to this and other GCC work that we're doing. And we'd like to thank HP Company for sponsoring this project and the Gelato Federation for their attention to Itanium and GCC.

References

[Aiken95] Alexander Aiken, Alexandru Nicolau, and Steven Novack. Resource-Contrained Software Pipelining. IEEE Transactions on Parallel and Distributed Systems, 6(12), pp. 1248–1270, December 1995.

[GCCInternals] `http://gcc.gnu.org/onlinedocs/gccint`

[Ebcioğlu88] Kemal Ebcioğlu. Some design ideas for a VLIW architecture for sequential natured software. In *Parallel Processing (Proceedings of IFIP WG 10.3 Working Conference on Parallel Processing)*, North Holland, Amsterdam, 3–21, 1988.

[EPIC] Michael S. Schlansker, B. Ramakrishna Rau. EPIC: An Architecture for Instruction-Level Parallel Processors. HP Laboratories Palo Alto Technical Report HPL-1999-111, February 2000. `http://www.hpl.hp.com/techreports/1999/HPL-1999-111.pdf`

[IA64Speculation] `http://gcc.gnu.org/ml/gcc-patches/2005-12/msg01924.html`

[Makarov03] Vladimir Makarov. The finite state automaton based pipeline hazard recognizer and instruction scheduler in GCC. In *Proceedings of GCC Developers' Summit*, Ottawa, Canada, June 2003.

[Moon97] Soo-Mook Moon and Kemal Ebcioğlu. Parallelizing Nonnumerical Code with Selective Scheduling and Software Pipelining. ACM TOPLAS, Vol 19, No. 6, pages 853–898, November 1997.

[Nicolau85] Alexandre Nicolau. Percolation Scheduling: a Parallel Compilation Technique. Technical Report. UMI Order

Number: TR85-678., Cornell University, 1985.

[ZadeckRegions] `http://gcc.gnu.org/ml/gcc-patches/2005-09/msg01888.html`