Reprinted from the

Proceedings of the GCC Developers' Summit

July 18th–20th, 2007
Ottawa, Ontario
Canada

Implementing an instruction scheduler for GCC: progress, caveats, and evaluation

Andrey Belevantsev
ISP RAS
abel@ispras.ru

Maxim Kuvyrkov
ISP RAS
mkuvyrkov@ispras.ru

Dmitry Melnik
ISP RAS
dm@ispras.ru

Alexander Monakov
ISP RAS
amonakov@ispras.ru

Dmitry Zhurikhin
ISP RAS
zhur@ispras.ru

Abstract

A year and a half ago, we started a project on implementing a new, aggressive instruction scheduler for GCC, aiming at VLIW-like architectures. We have followed the selective scheduling approach [Moon97] in essentials. At the moment of writing, we have completed the basic functionality of the scheduler, including support for bookkeeping code creation, instruction unification, register renaming, and forward substitution. The scheduler also performs software pipelining on loop nests and supports control and data speculation on IA-64. The implementation is available on the `sel-sched` branch in the GCC repository.

This paper focuses on the problems we had to solve to implement in GCC this particular scheduling approach, and on the new infrastructure we have created for this purpose: analyzing dependencies on-the-fly; extending APIs of dependency analyzer, cfg engine, and `emit-rtl.c` to initialize data for new insns and basic blocks; and working with insn RHSes as well as with whole instructions. We explain how this mechanism allowed us to add new features such as speculation. We describe the mechanisms used to choose the best instruction on each scheduling iteration: using DFA interfaces, per-region instruction priorities, and cooperating with the backend.

We also describe the effort on improving the IA-64 backend, needed to get performance results from the scheduler, including placement of stop bits, aligning issues, and heuristics implemented inside scheduler hooks. Results of preliminary evaluation of the scheduler on the set of small benchmarks and the SPEC CPU2000 suite are also presented.

1 Introduction

In late 2005, the compiler team of the Institute for System Programming of the Russian Academy of Sciences (ISP RAS) started a project on implementing the aggressive instruction scheduler for GCC. The project is motivated by the importance of instruction scheduling for modern architectures, such as IA-64, PowerPC, and Cell. The existing instruction scheduler, also known as the Haifa scheduler, has a number of limitations that make unconvenient its further development. The most important such limitation is lack of instruction cloning support. Absence of instruction cloning reduces scheduling freedom and does not permit to add more transformations to the scheduling engine, among which one of the most importance is software pipelining. The second limitation is that the scheduling front, i.e. all places in a scheduling region at which we are scheduling at the moment, always contains a single instruction. This implicitly proritizes control flow paths among which the front traverses first over paths that are traversed last.

The project goals were to implement a state-of-the-art approach to the instruction scheduling, focusing on the IA-64 platform as the one that needs good scheduling most, and to create a framework in which it is easy to add more transformations, such as speculation, to increase scheduling power. This framework should consist of the scheduler core, which is capable of scheduling arbitrary DAGs and supports basic instruction transformations, and of the IR manipulation core, which processes new instructions and basic blocks, adding them to the current region and initializing various scheduler data for them. We have chosen selective scheduling [Moon97] as a basis for the scheduler core. It focuses on VLIW architectures and incorporates basic transformations, such as instruction cloning, register renaming, and forward substitution.

The previous paper about this project [Belevantsev06] focused on implementing approach details via GCC infrastructure, and also described some ideas how to improve the basic implementation. This paper introduces the most important parts of the developed framework, namely dependence analysis, manipulation of insns/basic blocks, software pipelining, and target cooperation. We explain how these parts are used to implement basic transformations and speculation support. We also discuss preliminary evaluation results.

The rest of the paper is organized as follows. Section 2 provides an overview of the selective scheduling approach. Section 3 describes parts of the framework listed above. Preliminary results are discussed in Section 4. Section 5 concludes.

2 The selective scheduling approach

In this section, we will sketch the most important ideas of the selective scheduling, namely: 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, and implementing software pipelining on top of the scheduler. A similar section was included in [Belevantsev06], but we reworked it and included in this paper for completeness.

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. An 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. A scheduling point is called a *fence*, because when performing software pipelining, code motion through a fence is prohibited. All fences form the scheduling front. 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 each *boundary*[1] of the current group. Then the best operation is chosen and scheduled. Finally, the best operation is moved up to the each group boundary from its original location, possibly creating bookkeeping copies and updating av sets along its moving path (see Figure 1).

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

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

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

    move_op_to_boundary (boundary, best_op);
    schedule_op (best_op, group);

    advance_group_boundary (&boundary);
  }

  return group;
}
```

Figure 1: The `fill_group` routine

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, let us consider the renaming transformation. 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.

First, we determine whether an insn is *separable*, i.e. it can be represented as `lhs = rhs`. The RHS is then added to the av set analogously to other instructions. Later, the choosing code will also find the best target register for the RHS, and the operation will be scheduled as `best_reg = best_rhs`. We will consider

[1]When a group contains conditional jump(s), it can have several boundaries. This is currently not supported.

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:

$$\begin{aligned} avset(n) &= moveup\_set(\bigcup_{x \in Succ(n)} avset(x)) \\ &\quad \bigcup av\_op(n) \end{aligned}$$

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.

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. Usually, it is enough to check for the data dependence between the two. When no dependence exists between them, the operation stays unchanged. With additional transformations enabled, we check whether all dependencies could be eliminated with a certain transformation, e.g. substitution. For example, `x+y` RHS could be moved before the `y=z` copy as `x+z`, i.e. true dependence between these two operations can be eliminated with substitution. In this case the propagation helper will modify the operation before returning (see Figure 2).

Instruction unification happens when av sets of a branch point's successors are joined. In a resulting av set, equal operations are unified into one. When an operation is being scheduled as an RHS, its RHS is compared to the

```
moveup_op(insn, op) {

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

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

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

  /* Try something else...  */
  if (...)

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

Figure 2: The `moveup_op` propagation helper

other operations of the same class. Separable and non-separable operations are considered unequal. The comparison is done via `rtx_equal_p`.

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 it 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 in the parallel group. An important point to note is that register liveness information is needed to be able to choose an instruction. An instruction can be unavailable for scheduling because certain registers are live on the code motion paths of this instruction. Thus, the liveness information should be computed and updated analogously to the av sets data (forming *lv sets*).

The code motion process is driven by the `move_op` routine. It traverses the DAG starting at the group boundaries in search of the original operations (from which `best_op` could be derived). Let's assume first that we have not performed any transformations, then it's

```
//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

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. When register renaming is used, the operation is replaved with a copy `old_dest=new_dest`. 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. All predecessors of a join point should be in the same region as the join point itself, so that bookkeeping copies will be definitely scheduled later. 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 Figure 3). This allows creating only necessary bookkeeping code.

The process is more complicated when additional transformations are performed. It is not enough to search for the `best_op` because it could be changed by a transformation (e.g. substitution or speculation). Hence, we should "untransform" `best_op` when traversing 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. Also, during backtracking the current form of the `best_op` at the node being traversed should always be retained to allow correct creation of bookkeeping code.

2.4 Software pipelining

Selective scheduling supports software pipelining of innermost loops via manipulating fences. When a loop region is being scheduled, current set of fences represent "dynamic" backedges in the sense that code motion through those fences is not allowed. Therefore, at any time a scheduling region is actually acyclic. Moreover, each time a fence is advanced, a new edge in the region becomes the backedge, so that code motion through the edge that was previously a backedge is now allowed (see Figure 4, where current backedges are shown in red).

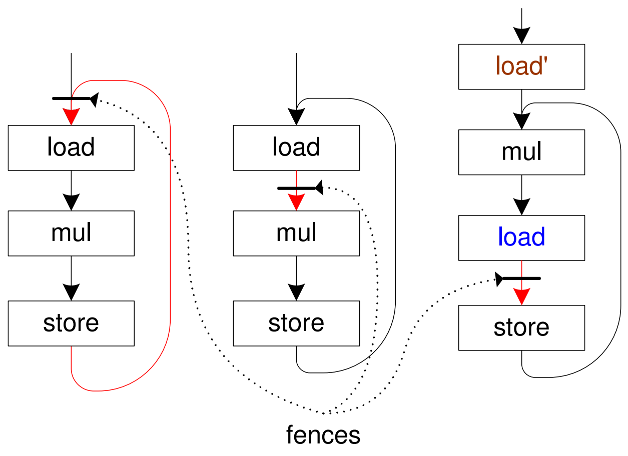

Figure 4: Software pipelining in the scheduler

The actual pipelining happens when an instruction is being moved up along the real backedge of the loop. The bookkeeping copy of the moved instruction is created before the loop header, forming the prologue of the pipelined loop (on the figure, the pipelined load is shown in blue, and its bookkeeping copy in the prologue is shown in brown). Analogously, the epilogue is created when a conditional jump to the beginning of the loop is being moved up, and bookkeeping copies are created on both targets of this jump. However, moving jumps with creation of bookkeeping copies is currently not supported.

3 Implementation highlights

In this section we discuss the most interesting parts of the scheduler implementation. For each part, we explain the way it was implemented, why we chose this way, and which scheduler features need this part to work. We start with describing basic data structures of the scheduler. Then we follow with initialization mechanism of this data for the new instructions, basic blocks, and regions coming to the scheduler. We describe handling dependencies via the dependence analyzer, supporting multiple fences, implementing register renaming, instruction transformations, and software pipelining. We finish with discussing support of target-dependent part of the scheduler, i.e. using the DFA interface and scheduler hooks to choose the best instruction and to ensure the correctness of the resulting schedule.

3.1 The basic data structures

There are four types of data in the scheduler: per-region, per-basic block, per-instruction, and per-operation data. The per-region data is being worked with analogously to the Haifa scheduler, as we reuse the same code for region recognition except when pipelining outer loops. The per-basic block and per-instruction data in GCC is stored in vectors indexed by corresponding uids.

An instruction together with its data is represented to the scheduler as a *virtual* instruction, or *vinsn*. A vinsn holds all the information that can be derived from the insn pattern only, i.e. it is not flow-sensitive. This includes:

- Whether an instruction can be cloned or not (if not, we cannot create bookkeeping copies for such an instruction. Examples of such instructions are calls and assembly statements.),
- Whether an instruction can be separated on a left-hand side and a right-hand side expressions, and those expressions themselves, if it can,
- Sets of registers used, set, or clobbered by this instruction. The sets are needed for checking liveness restrictions on a destination register when an operation is being selected, and updating liveness information after code motion.

A vinsn type also serves as a smart container for the flow-insensitive information. This information changes only when an operation is transformed while being propagated through the flowgraph, so it is desirable to allow sharing of vinsns to reduce memory footprint of the scheduler. The container supports this via lazy copying of vinsn data on request, when the operation is being changed, and attaches the same vinsn to another operation otherwise.

An operation is an element of an availability set, so it describes a single scheduling entity as viewed by the algorithm. An operation is represented as a vinsn together with the flow-sensitive information, i.e. the data that describes this vinsn during the propagation process. It should be noted that the initial per-insn data is actually the operation initialized for the original position of the insn in the scheduling region. This data is mainly used to prioritize operations to be able to select the best one for scheduling.

3.2 Initializing new insns and basic blocks

When scheduling, new instructions can be created explicitly as bookkeeping copies. These are immediately initialized. However, new instructions and basic blocks could also be created implicitly as a result of modifying control flow.[2] For basic blocks, we provide specific implementation of the `create_basic_block` hook, which records created blocks in a local vector. Similarly, we add an RTL hook that is called whenever an insn is being emitted. The hook records new instructions in another local vector.

Finally, we provide a number of wrappers for the control flow API functions, e.g. for `redirect_edge`, `split_block`, and some others. A typical wrapper

[2]This is not a problem for the cfglayout mode. We don't use it to be able to schedule jumps.

calls the original function, then it adds the recorded blocks to the current region. In the very end, the new instructions are initialized. This is because instruction initialization procedures require the consistent control flow to work.

3.3 Handling data dependencies

Handling dependencies in the selective scheduler is more complicated than in the Haifa scheduler. This is due to several reasons. First, it is important to know which part of an insn has caused the dependence; we can overcome some of dependencies with renaming and substitution. Second, it is hard to deduce correct dependencies of a bookkeeping copy from the original instruction. For example, the source insn is dependent on certain conditional jumps in the source code, but the copy will depend only on some of them, and those also vary between different fences. Third, insn dependencies may change due to transformations, such as substitution or renaming. We have decided to support on-the-fly mechanism of calculating dependencies, in other words, make them "dynamic" instead of "static." The mechanism consists of two parts: managing dependence contexts and adding hooks to the dependence analyzer.

A *dependence context* describes the state of dependencies during analysis (declared as `struct deps`). When processing an insn, the analyzer will find dependencies between this insn and the insns recorded in the current dependence context. For the Haifa scheduler, a dependence context always contains all insns starting from a region head up to the current insn. In the selective scheduler, we use dependence contexts for various purposes.

Before scheduling a region, we run initial dependence analysis pass that has the same dependence context as the Haifa scheduler. This is used for initializing various flags such as `SCHED_GROUP_P` or `CANT_MOVE`. During the computation stage, a dependence context has only one instruction through which we propagate an operation in the `moveup_op` routine. When the best insn is being chosen for scheduling, a dependence context is used to check whether all input data for an operation is ready, i.e. all producers are done. For that purpose, we maintain a separate dependence context for each fence, which includes all instructions that were scheduled on this fence.

During the above tasks, we require the analyzer to perform different actions.[3] For this purpose, we have added a number of callbacks to the dependence analyzer, namely `{start, finish}_{insn, lhs, rhs}`, `note_reg_{set, use, clobber}`, and `note_dep, note_mem_dep`. These hooks are called by the analyzer when corresponding events are happened and perform some user-defined actions. For example, in `moveup_op` we record the insn part upon which the operation depends. When checking all the producers of an operation, we calculate the earliest cycle on which the operation could be scheduled.

The price of the flexibility of this on-the-fly analysis is that the user code should implement caching of dependence queries by itself. Without this, compile time spent in the dependence analysis is unacceptable. We are working on the caching mechanism, but do not yet have any numbers.

3.4 Supporting multiple fences

For the selective scheduler, supporting multiple fences means supporting multiple scheduling states at the same time. This task mainly boils down to localizing in the fence structure some previously global variables and flags. This includes the DFA state, `state_t`, and the dependence context; current cycle, number of insns scheduled on this cycle, last scheduled insn, did we have a stall, etc.

However, this is not enough from the target point of view. Previously, a target could assume that the scheduler has a unique state which could be saved for the sake of scheduler hooks implementation. E.g., the IA-64 backend saves last scheduled insn and the DFA state to be able to place stop bits. With the new scheduler, this tecnique does not work. To solve this problem, we have introduced a concept of a *target context*.

A target context is used to localize all the variables that are needed internally by the target. From the scheduler point of view, a target context is an opaque type, represented as a `void*` pointer. The scheduler implements a number of API functions to manipulate target contexts. These functions are mainly wrappers of corresponding `targetm.sched` hooks, which should be implemented by the target to handle needed variables. No hooks are needed if the target doesn't want to save any bits of the scheduler state.

[3]The default action of the analyzer is creating dependence lists.

3.5 Implementing register renaming

Register renaming applies only to separable operations. The difference in scheduling between separable and non-separable operations begins in the propagation stage. When an operation is being scheduled as an RHS, we discard all non-RHS dependencies, i.e. anti and output dependencies. (We may try some transformations to eliminate the remaining RHS dependencies, e.g. via substitution.)

On the choosing stage, we calculate the set of available target registers for each operation from the resulting av set. This requires to mark as used all registers that are live on all code motion paths to originators of an operation. To determine these paths, we need to perform the recursive search similar to that of the code motion routine. During this search, we also check for registers that are live on exits from the current region and on targets of conditional branches that do not lie on code motion paths. All these registers cannot be used as a target register for a given operation because of overlapping live ranges.

Additionally, we need to take into account all registers that are unavailable due to target-dependent considerations. A list of these conditions is taken from `regrename.c` and is roughly as follows:

- A register's class and mode should be compatible with all uses of original register;
- A register is neither a leaf register in a leaf function, nor a call saved register, nor a frame pointer;
- A register is set in the `regs_ever_live` array. We plan to support using call-saved registers in the future, so this restriction could be lifted.

Among available registers, we prefer one of the original registers, or the oldest one that was previously used for renaming. The last restriction to be checked is that the resulting instruction should be recognizable.

When scheduling before reload, all liveness restrictions are checked in the same way. Some of the unavailable register checks are performed too, for the sake of hard registers that are already introduced. For pseudo registers, we can always generate the new target register for an operation. Currently we do not control register pressure resulting from this, which is to be done in the future. As a result, we do not run selective scheduling before reload by default.

3.6 Implementing insn transformations

As noted above, additional instruction transformations could be implemented to eliminate more dependencies when scheduling. Currently, we have implemented forward substitution and speculation support. Both transformations follow the same scheme.

When propagating a separable operation through an insn, we consider RHS dependencies of the operation. If those are eligible for the transformation, we perform it and update the operation. Currently, the default dependence hooks for the selective scheduler will record the single dependence with the merged status of all dependencies found.[4] This behavior can be changed to record all the data if needed.

When searching for original operations during code motion, we have to perform the reverse transformation to be able to find original operations. As we don't memorize the exact pattern of the original transformation, "untransforming" an operation could yield multiple results. For example, unsubstituting `y=x+x` through `z=x` results in four possible combinations. Similarly, when a control speculative load is being propagated back through a jump, unspeculation results in both regular load and control speculative load. The resulting set of untransformed operation is also filtered through `recog` to leave only valid insns. When the original operation will be found, the correct form of the resulting set will be used during backtracking to create required bookkeeping copies.

3.7 Implementing software pipelining

As noted in Section 2.4, support for software pipelining of innermost loops is provided via manipulating fences. From the implementation point of view, this is taken care of by an iterator used for traversing a scheduling region. The iterator is called `FOR_EACH_SUCC`, and its task is to enumerate all successors of an insn. The successors are considered valid depending on the flags

[4] I.e., when several dependencies are found between the operation and the insn, the resulting status will reflect the "hardest" dependence.

passed to the iterator. For example, SUCCS_NORMAL will return all successors belonging to the current region, SUCCS_ALL will not filter any successors, etc. When pipelining, successors reached through a back edge are considered valid; otherwise, they are filtered out and will not be returned for a normal traversal.

A prologue is formed by bookkeeping copies created when an insn is being propagated through a loop header. As noted earlier, bookkeeping insns should be scheduled and thus should be placed in the same region. For this purpose, a *preheader* block is created as a placeholder for these insns. This ensures that new insns will end up in the current region. However, we will not schedule them during the first pass over the region, because there is no way to reach these instructions. Moreover, when multiple fences exist during pipelining, we may end up with unscheduled bookkeeping code in the middle of the loop. To solve this problem, we run the second pass that schedules all previously unvisited basic blocks of the region.

With pipelining, an instruction could be scheduled more than once. To ensure that pipelining process will terminate, we prefer instructions that were scheduled less times when choosing. However, as we also check that all input data for an instruction should be ready, this is not enough. For example, a producer could be pipelined on each scheduling iteration, forcing the consumer to be unavailable for scheduling. This problem does not exist in the original approach, as it assumes that all instructions have the latency of one. In our case, we just limit the number of times an instruction could be pipelined.

For pipelining of outer loops, two things are changed in the basic approach. First, scheduling regions are formed from loop nests produced by the loop optimizer. The regions are created starting from the innermost loop. A loop region consists of the loop preheader created by the loop optimizer and the loop body excluding all basic blocks belonging to its inner loops. After scheduling the loop, its preheader is moved to the region corresponding to its outer loops, so that the prologue code will be scheduled together with the outer loop code. When loop latches are shared, i.e. the latch of the outer loop also belongs to the inner loop, it is also moved to the outer loop region.

When pipelining an outer loop region, the inner loop body (which is already scheduled) is treated like a barrier, i.e. no code motion is possible through it.[5] The FOR_EACH_SUCC iterator is changed to let the scheduler to reach the blocks of the outer loop that are below the inner loop. For this purpose, when a fence is about to move to the inner loop, which is normally not allowed as this would be another region, we detect this and return inner loop exits as actual successors for this fence.

To be able to use the loop optimizer API, we should work harder to preserve consistent loop structures in the cfgrtl mode. As we don't change much of the control flow, but merely query loop properties, this requires relatively small adjustments to the loop optimizer (for example, ensuring that a loop preheader edge is a fall-through edge) and some support inside the control flow wrappers implemented in the scheduler, namely updating a latch and a header.

3.8 Target-specific details

The target-dependent part of the scheduler is the choosing stage, when we select the best instruction for scheduling and the best target register for this instruction (when register renaming is active). The latter problem is covered in Section 3.5. As for the choosing mechanisms, we reuse three existing features of GCC: instruction priorities, DFA lookahead scheduling, and scheduler hooks.

Instruction priorities are calculated on the initialization phase of the scheduler, when the first pass of the dependence analysis is performed, using the code from the Haifa scheduler. When scheduling, instructions from an availability set are sorted using the code similar to the rank_for_schedule function. We consider a number of sorting criteria besides priotities, namely the *spec* attribute,[6] the number of times an instruction was scheduled, and SCHED_GROUP_P bits.

After sorting, a ready list is formed from the available instructions, and the list is passed to the DFA lookahead engine (i.e., max_issue function) and then to the scheduler hooks from the targetm.sched structure. The lookahead engine tries to choose such an instruction that: a) issuing it will then allow issuing as

[5]The original approach suggests treating an inner loop body as a super instruction, but we think it isn't worth the trouble. Nevertheless, this could be implemented in the future.

[6]The spec attribute equals to the number of conditional jumps, which are traversed by an operation, such as the operation is unavailable on some of the targets of this jump.

many instructions as possible, and b) issuing it will allow issuing the instruction with the highest priority on the current cycle. It does so by checking resource restrictions and pipeline hazards using the finite state automaton built from the functional units description of the target [Makarov03].

The scheduler hooks are used by the target to change the sorting order and the lookahead engine behavior. Sometimes, as on IA-64, calling hooks are required to get the correct schedule. The hooks should be compatible with both schedulers, so we call them at exactly the same moments during scheduling as the Haifa scheduler does. However, as we construct the ready list from the final av set, which actually contains operations instead of insns, the target is unable to change any of the insn parameters by looking at the ready list (for example, the PowerPC backend tries to modify instruction priorities). We plan to support this in the future.

The last thing to note is that the target is able to make use of the support for multiple scheduling points by implementing its own target context type, as described in Section 3.4. Any global variables that keep bits of the scheduler state should be localized in this context.

4 Current results

At the moment of this writing, we have implemented all the basic functionality of the selective scheduler. The basic framework supports register renaming, forward substitution, control and data speculation, pipelining of innermost loops, and pipelining of loop nests. The scheduler bootstraps on ia64 and powerpc (we only check `c`, `c++`, and `fortran`). The implementation is available on the sel-sched branch in the GCC SVN repository since January 2007. By default, the selective scheduler works as the second scheduling pass.

All results posted below are for HP rx6000 Itanium 2 servers. We didn't do any performance tuning on other platforms. Alexander Kirnasov from Samsung tested the new scheduler on a private 4.2-based branch on the Cell architecture and got up to 6-7% speedups on tests with a complex control flow [Kirnasov07]. He has utilized bigger scheduling regions (via `extend_rgns`) and bigger scheduling windows (up to 256 insns) than default.

4.1 Initial performance tuning

At the end of 2006 we have started to work on performance tuning of the scheduler. This work is done on the Itanium architecture. As a result, we have made a number of tweaks both to the IA-64 backend and to the scheduler itself. The backend fixes include alignment changes, better placement of stop bits, and memory dependence tweaks.

The default function alignment for IA-64 is now set to 64 bytes, and the default loop alignment is 32 bytes. This is consistent with the values used by the Intel compiler. The function alignment equals to the icache line size, and the loop alignment ensures that the processor frontend can deliver two bundles per cycle to the backend. Additional care should be taken of the loop branch label, which holds the alignment, to stay with the loop. When an extra block is created for the prologue code, this property doesn't hold anymore, and this is one of the reasons for us to force a loop preheader edge to be fallthrough.

It is desirable to place a stop bit after every cycle to avoid excessive stalls. This is because when any of the instruction's inputs are not ready, the whole instruction group will stall until the latest data will be available. This early placement is especially dangerous for an FP code with lots of long latencies, and this is what the Intel compiler does. We have implemented this feature via fixing the `dfa_new_cycle` hook and the final placement of stop bits.

With memory dependence tweaks we've tried to reflect the Itanium hardware better. For floating-point loads/stores, we consider all may-aliased memory dependencies to have a zero cost, and we increase the cost of a must-aliased dependency. We are also trying to limit the number of memory operations in an instruction group.

As for the scheduler improvements, these include numerous bugfixes and tweaks to the choosing heuristics. The big change is rescheduling the code of pipelined loops with disabled pipelining. This is motivated by a situation when pipelined instructions, which are moved from the beginning to the end of the loop, leave "holes" in the schedule. These holes can be potentially filled with the other instructions. However, the pipelined part of the loop will not change during this process.

Benchmark	`-O2`, pipelining, no spec		`-O2`, pipelining, control spec	
	Speedup, %	Code size, %	Speedup, %	Code size, %
Whetstone	–4.11	1.67	–3.47	1.67
Tfftdp	1.89	6.22	2.00	7.33
Sim	1.73	4.57	–0.03	3.20
Shuffle	1.00	1.20	0.70	1.51
Linpackc	5.00	1.31	1.52	4.28
Heapsort – High MIPS	0.00	1.17	1.43	1.96
Low MIPS	0.69		–2.07	
Dhrystone 1.1	–0.28	2.40	–1.84	2.38
Dhrystone 2.1a	–0.09	1.76	–2.27	1.31
c4 : Fhourstones 1.0	–0.07	1.77	1.02	–0.85
Black Jack	0.70	0.66	0.70	0.57
Nsieve – High MIPS	16.46	1.01	17.08	0.33
Low MIPS	4.49		4.25	
FFT size=1000000	2.84	4.99	segfault	–2.71
FLOPS – MFLOPS(1)	9.69	2.74	12.36	–7.73
MFLOPS (2)	10.92		10.91	
MFLOPS (3)	13.37		13.35	
MFLOPS (4)	21.02		21.01	
Hanoi: 29 disks	1.71	0.00	3.43	–0.60
Fibonacci	–6.77	0.00	–6.94	0.00
30 Queens	22.58	0.72	–101.78	1.53
MatMult – normal	–2.90	7.51	11.59	20.42
temporary variable in loop	–2.94		11.76	
unrolled inner loop. factor of 4	3.85		19.23	
pointers used to access matrices	5.41		10.81	
transposed b matrix	–1.39		1.39	
interchanged inner loops	0.00		1.61	
blocking. factor of 4	–12.18		–12.90	
4x4 subarray (from T. Maeno)	3.28		–3.28	
4x4 subarray (from D. Warner)	14.29		3.57	
Robert's algorithm	1.37		1.39	
Scimark	1.01	3.24	3.06	3.96
FFT	4.89		6.16	
SOR	–0.07		3.74	
Monte Carlo	1.26		1.28	
Sparse Matmult	1.67		3.60	
LU	0.08		0.34	
Blowfish – set_key	–0.01	3.83	0.75	3.52
raw ecb	0.02		0.74	
cbc	0.44		1.20	

Table 1: Results on aburto benchmarks, compared to the Haifa scheduler. Speedups and code size changes are shown. The scheduling window is set to 64 instructions.

4.2 Testing on small programs

We have started the tuning process using the well-known benchmark package compiled by Alfred Aburto [AburtoBenchmarks]. Current results can be found in Table 1. The biggest speedups on these tests can be attributed to the pipelining of innermost loops, and sometimes, as in `nsieve`, of outermost loops. Even when no pipelining happens in the outer loop, the prologue of the inner loop is scheduled together with the rest of the outer loop code, which allows to shave off some cycles.

When control speculation is enabled, we can pipeline loads, which are exception-risky instructions. This increases speedups for `matmult`, `scimark`, and `flops`, while overspeculating for `linpack` and `queens`. We are working on further performance analysis of the control speculation support.

It should be noted that on small testcases we've seen a lot of local fluctuations in performance. These are partly fixed by the backend changes described above, especially alignment adjustments and memory dependence tweaks. The other problem is cache conflicts, which are not modeled by the scheduler. To fix this, we can enable prefetching loop arrays by default on Itanium, starting at `-O2`.

4.3 Testing on SPEC CPU2000

We have been looking on the SPEC CPU2000 performance since February 2007. Current results can be found in Table 2. As with the small benchmarks, most speedups are because of the combination of pipelining, renaming, and control speculation. The backend tweaks are helpful mainly to the floating-point code with its long latencies, while the SPEC INT part does not change significantly.

The `perl` slowdown is due to the linker relaxation done when optimizing references to global variables. We are placing stop bits between loads of addresses of global variables, which get removed by the linker, but stop bits are left intact. The proper solution to this problem will be to remove useless stop bits during the relaxation [Wilson07].

5 Conclusions

In this paper, we presented a project on implementing an 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 improves scheduling for modern architectures such as IA-64 and Cell.

The basic parts of the scheduler infrastructure are now implemented and can be found at the sel-sched branch in the GCC repository. The branch bootstraps with `c`, `c++`, and `fortran` enabled on ia64 and powerpc. We are currently working on performance tuning for Itanium. Our first results show that the pipelining, speculation, and renaming are beneficial for the floating-point code.

We aim to prepare the scheduler for inclusion into GCC 4.4 near the end of 2007. For this purpose, we will continue to work on performance tuning and compile time improvements. Other change that will be needed is using the dataflow engine for updating register liveness information.

6 Acknowledgments

We'd like to thank Vladimir Makarov from Red Hat for extensive consulting of this project. The insights to instruction scheduling and GCC world provided by Vlad are invaluable. We thank Daniel Berlin, Zdeněk Dvořák, Jan Hubička, Diego Novillo, Sebastian Pop, 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

[AburtoBenchmarks] Alfred Aburto's system benchmarks. Could be found at `ftp://gd.tuwien.ac.at/perf/benchmark/aburto`

[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.

[Belevantsev06] Andrey Belevantsev, Maxim Kuvyrkov, Vladimir Makarov, Dmitry Melnik, and Dmitry Zhurikhin. An interblock

Benchmark	`-O2`, pipelining, no spec		`-O2`, pipelining, control spec	
	Speedup, %	Code size, %	Speedup, %	Code size, %
164.gzip	2.14	0.61	4.00	0.89
175.vpr	–2.22	2.32	–2.38	2.85
176.gcc	X	1.56	X	1.62
181.mcf	0.00	1.25	1.68	0.03
186.crafty	–1.27	2.45	–1.28	2.52
197.parser	–1.19	1.45	–1.04	1.76
252.eon	1.87	2.01	3.20	2.09
253.perlbmk	–7.46	2.16	–7.53	2.29
254.gap	0.71	3.71	0.18	3.86
255.vortex	0.73	1.76	–2.96	1.86
256.bzip2	4.88	1.86	11.93	1.94
300.twolf	0.90	2.07	0.10	2.90
SPECint2000	–0.13	1.93	0.43	2.05
168.wupwise	1.40	2.03	4.64	2.27
171.swim	2.38	4.35	5.08	8.70
172.mgrid	5.59	6.28	11.60	16.91
173.applu	–0.23	3.79	1.61	16.48
177.mesa	–1.86	4.60	–0.14	4.77
178.galgel	–0.29	3.24	1.55	5.48
179.art	–3.13	1.32	–2.68	3.78
183.equake	0.00	1.97	–0.45	3.87
187.facerec	0.76	4.56	1.54	5.31
188.ammp	2.99	2.76	5.29	3.75
189.lucas	1.17	1.64	1.66	1.87
191.fma3d	0.00	3.56	–1.83	5.33
200.sixtrack	3.93	2.70	6.21	3.90
301.apsi	1.97	4.52	6.73	6.59
SPECfp2000	1.03	3.38	2.85	6.36

Table 2: Results on SPEC CPU 2000 benchmarks, compared to the Haifa scheduler. The scheduling window is set to 64 instructions. Pipelining of outermost loops is disabled.

VLIW-targeted instruction scheduler for GCC. In *Proceedings of GCC Developers' Summit 2006*, Ottawa, Canada, June 2006.

[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`

[Kirnasov07] Alexander Kirnasov. Private communication, 2007. `kirnasov@mail.ru`

[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.

[Wilson07] Jim Wilson. Re: Questions regarding address relaxation on IA-64. Sent to the binutils mailing list in March 2007 and archived at `http://sourceware.org/ml/binutils/2007-03/msg00319.html`

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