Proceedings of the GCC Developers' Summit

June 22nd–24th, 2005
Ottawa, Ontario
Canada

Conference Organizers

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

Review Committee

Eric Christopher, *Red Hat, Inc.*
David Edelsohn, *IBM*
Richard Henderson, *Red Hat, Inc.*
Andrew J. Hutton, *Steamballoon, Inc.*
Janis Johnson, *IBM*
Toshi Morita
Gerald Pfeifer, *Novell*
C. Craig Ross, *Linux Symposium*
Al Stone, *HP*
Zack Weinberg, *Codesourcery*

Proceedings Formatting Team

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

Improving GCC instruction scheduling for IA-64

Andrey Belevantsev
ISP RAS
abel@ispras.ru

Alexander Chernov
ISP RAS
cher@ispras.ru

Maxim Kuvyrkov
ISP RAS
mkuvyrkov@ispras.ru

Vladimir Makarov
Red Hat
vmakarov@redhat.com

Dmitry Melnik
ISP RAS
dm@ispras.ru

Abstract

The instruction scheduler is one of the weakest points of GCC on IA-64 architecture. This paper will describe an ongoing project for improving GCC instruction scheduling for Itanium. We aim at adding support for IA-64 control and data speculation to GCC, doing this similarly to the existing implementation of interblock motions. Our work on this issue forces us to address weakness of the alias analysis used in the scheduler. Absence of the address displacement on IA-64 makes the problem even more relevant. We suggest the following improvements: 1) propagation of points-to information from tree-ssa to RTL for pointers, and 2) tracking pointer arithmetic, which is performed for addressing non-pointer variables, in `alias.c`.

1 Introduction

The Itanium architecture (known as IA-64) was named EPIC by Intel, which stands for Explicitly Parallel Instruction Computing. The program compiled for IA-64 should have instruction level parallelism (ILP) explicitly exposed by the compiler. For this purpose, the architecture provides massive resources such as a huge register file and many execution units. IA-64 also supports features for enhancing ILP, such as data and control speculation, predication, and rotating registers.

An instruction scheduler is a component of the compiler that rearranges instructions to achieve better performance. Aggressive global scheduling is a key to utilize Itanium ILP features [Muthukumar05]. The GCC scheduler doesn't fully obtain this goal. The current approach is not powerful enough to express much ILP, and it doesn't support IA-64 features such as speculation or rotating registers.

This paper presents an ongoing project aimed at improving GCC instruction scheduling for Itanium. The project is sponsored by the Gelato Federation. The primary goal of the project is to add support for control and data speculation to the scheduler. To benefit from this improvement, a problem of weak RTL alias analysis on IA-64 should be also addressed. The problem is important because there is no "base + offset" addressing mode on Itanium. We suggest a two-step solution: propagating points-to information from the Tree SSA to the RTL, and tracking pointer arithmetic within `alias.c`.

The third small improvement, which we suggest, is using standard GCC probability analysis instead of the current historical implementation used by the scheduler.

The rest of the paper is organized as follows. Section 2 contains the description of the current GCC scheduler. Sections 3 and 4 describe in detail our work on supporting speculation and enhancing alias analysis, respectively. Section 5 mentions the changes in probability analysis. Our experimental results are sketched in Section 6. Section 7 outlines the directions for the future work.

2 GCC instruction scheduling

The GCC instruction scheduler is an adopted version of the interblock scheduler from IBM Haifa Labs. Scheduling pass is performed twice, before and after register allocation. The first pass does interblock scheduling, while the second pass schedules extended basic blocks (EBBs) on Itanium and single basic blocks on other platforms.

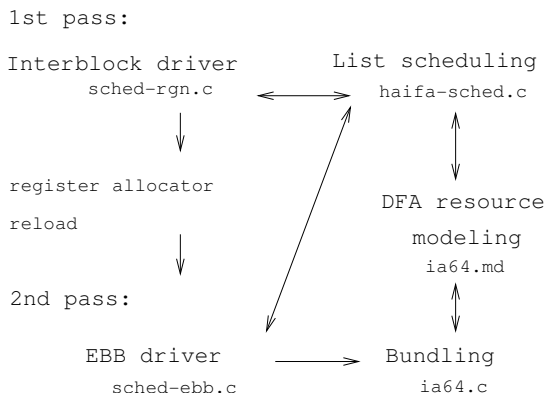

Figure 1: GCC instruction scheduler.

Scheduler infrastructure is shown in Figure 1. Generic routines doing list scheduling manipulations for the basic block are located in `haifa-sched.c`. The common part of data dependencies calculation is done in `sched-deps.c`. Routines for interblock and EBB scheduling are in `sched-rgn.c` and `sched-ebb.c`, respectively. The entry point for the scheduling passes is the `schedule_insns` routine. It calculates data dependency info, computes "regions" for interblock scheduling, and schedules every block in the region via `schedule_block`. The scheduler uses `sched-int.h` for communication between the components. Pieces of logic abstracted away from the Haifa part to either interblock or EBB parts are accessed through hooks from the `current_sched_info` structure.

The Haifa scheduler manipulates instructions using list scheduling (see Figure 2). Firstly, the instruction priorities are calculated basing on data dependencies. Scheduling is started with a list of ready and pending instructions. The list is sorted according to the set of heuristics. An instruction is chosen from the ready list and scheduled or queued, if there is not enough resources to issue it. Pending instructions are moved to either the ready list or the queue as their dependencies are satisfied. After scheduling the register live information is updated.

The scheduler communicates with the backend via various hooks. The backend could adjust instruction cost and priority, reorder list of instructions ready for scheduling, etc. The scheduler uses a pipeline hazard recognizer to ask for a possibility of issuing an instruction on the given cycle. The pipeline hazard recognizer is based on a deterministic finite state automaton (DFA). On Itanium, the scheduler uses a nondeterministic version of the automaton for the interblock and the EBB passes. After EBB is scheduled, its instructions are bundled by using deterministic automaton. The bundling algorithm uses dynamic programming for finding the best sequence of the bundles. The algo-

```
schedule_block() {
 /* queue - ready, but could not be
 scheduled on this cycle
 ready - could be scheduled now  */

 foreach (insn in the region)
  if (new_ready (insn))
   move insn to ready;

 while (should schedule)
 {
  move insns from queue to ready;
  while (could issue on this tick)
  {
   insn = choose from ready;
   if (could schedule insn now)
   {
     schedule insn;
     foreach (insn1 with resolved
              dependencies)
      if (new_ready (insn1))
       move insn1 to queue or ready;
   } else
     move insn to queue;
  }
  goto next tick;
 }
}
```

Figure 2: Haifa scheduler pseudocode

rithm performs the forward pass over the EBB to find the sequence. Then the backward pass performs actual inserting templates and NOPs in the sequence.

3 Speculation support

3.1 Overview

Data and control dependencies limit freedom of instruction movement. Using speculation allows the compiler to overcome the dependencies by moving a load through the ambiguous store (doing *data* speculation) or moving a memory load across a branch (doing *control* speculation). Uses of such a load could also be moved. These techniques provide the way of hiding memory latency of moved loads and reduce the execution time.

IA-64 architecture supports both control and data speculations via separation of data loading and possible exception handling. Speculative versions of load instruction are supported. The control speculative (or just *speculative*) load defers possible exceptions. The data speculation (or *advanced*) load saves the address of the load. Later on, *check* instruction detects deferred exception/possible store conflict and either reissues the failed load or branches to the recovery code. The examples of IA-64 control and data speculation are shown on Figure 3 (some instruction completers and nops are omitted for clarity).

Data speculation:

Before	After

`adds r15=r16,r14`
`st8 r14=[r14]`
`nop.i`
`ld8 r18=[r19];;`
`st4 r15=[r33]`
`nop.i`
`ld8 r14=[r18];;`
	`ld8.a r18=[r19];;`
	`adds r15=r16,r14`
	`nop.i`
	`st8 r14=[r14]`
	`ld8.c.clr r18=[r19]`
	`nop.i;;`
	`ld8 r14=[r18]`
	`st4 [r15]=r33`

Control speculation:

Before	After

`mov r1=r42`
`adds r14=1,r8;;`
`cmp4.ltu p6,r14`
`(p6) br.cond bd0`
`ld4 r14=[r33];;`
`add r14=r14,r8`
	`adds r14=1,r8`
	`ld4.s r15=[r33]`
	`mov r1=r42;;`
	`cmp4.ltu p6,r14`
	`(p6) br.cond bf0`
	`chk.s.m r15,b40;;`
	`add r15=r15,r8`

Figure 3: IA-64 control and data speculation

When the advanced load is executed, the load

address is recorded in a hardware table called ALAT (Advanced Load Address Table), which is indexed by the destination register number. Stores writing to the overlapping address and/or advanced loads to the same register remove previously added entry. When a check instruction with the same register number is executed, the ALAT is searched for the entry indexed by this register. If the entry is not found, speculation is considered failed and its results should be recomputed.

When the control speculative load is executed, the Not A Thing (NaT) bit is set on the destination register, if an exception occurs, signaling that the exception is deferred. The NaT bit could be set for each general register. If the NaT bit of a register is set, it is propagated through dependent computations with this register. The control speculative check instruction branches to recovery code, if the NaT bit of the destination register is set.

3.2 General approach

We propose a notion of *speculative blocks* for modeling speculation in the scheduler. Some of the instructions could start such a block (memory loads). Then more instructions could be scheduled speculatively (uses of these loads). Finally, special instructions should end a speculative block (check instructions). Thus, each speculative load and check form a block, which could optionally include uses of the load.

We extend both instruction and dependence data to reflect their speculative properties. The speculative flag of data dependence means "this should be done to overcome me," while speculative status of an instruction means "this is a possible way of scheduling me" (see Table 1 for examples). The status could also mark out the instructions, which are more preferable for speculation, or should not be speculated at all.

The instruction flag is placed in `haifa_insn_data` structure. A speculative status of a dependence could be placed inside `LOG_LINKS`. We have chosen to create a new kind of dependence, `INSN_DEPS`, to be used only by the scheduler and to keep the new flag in it. This solution allows more freedom of experiments with the patch. After the patch is finished, this part can be rewritten to use `LOG_LINKS`.

The life cycle of a speculative instruction is as follows. Firstly, the `INSN_DEPS` dependencies are calculated together with `LOG_LINKS`. An instruction (together with its dependencies) acquires its speculative status in the `new_ready` function. The status affects the rules of choosing an instruction from the ready list for scheduling. Later, when the instruction is selected for speculative scheduling, it can be still last-minute rejected by the backend (e.g., when there are too many subsequent speculations). If the instruction is scheduled, its speculative status is cleared, and other instructions (e.g., its uses) can acquire speculative status, again in `new_ready`.

Speculative instruction is scheduled as follows. If the instruction is supposed to begin a speculative block, it is split on a speculative part and a check part (or simply check). Both the backward and forward dependencies of the original instruction are moved to the check, and a dependence between the speculative and the check part is added. The speculative instruction is scheduled on the current cycle. The check is marked as an instruction for the end of the speculative block, and scheduled later as usual.

3.3 Filling the ready list

An instruction could be placed in the ready list either when scheduling is started, or when other instruction is scheduled and some data dependencies are satisfied. Validity of this action is checked via the `new_ready` function.

Table 1: Speculative status examples

Speculative status is:	Dependence could be broken with:	Instruction could be:
BEGIN_DATA	advanced load	an advanced load
BEGIN_CONTROL	speculative load	a speculative load
BE_IN_DATA	uses of advanced load	a use of advanced load
BE_IN_CONTROL	uses of speculative load	a use of speculative load
FINISH_DATA	—	an advanced check
FINISH_CONTROL	—	a speculative check
HARD_DEP	could not be broken	could not be speculative
WEAK_DEP	preferable for speculation	preferable for speculation

An instruction could be either from currently scheduling basic block or from another one. In the first case, the instruction may have no dependencies, and thus it is ready for scheduling. If the instruction has unsatisfied dependencies, it is a candidate for data speculation. It may either open a new speculative block or go inside the existing one. The latter case happens when the instruction depends only on checks, i.e., it consumes data only from the speculative instructions.

Data speculation would be valid iff the speculative motion preserves the correctness of register operations. Consider the following code snippet:

```
<current scheduling point>
 add  r3  = r3, r4
 st  [r6] = r3
 ld   r4  = [r5]
```

A store to `r4` may not be moved speculatively to the current scheduling point, because this move would violate the anti dependence between `ld` and `add` instructions. However, only true dependencies could be overcome with speculation. To reflect this issue, the speculative status is assigned to the true dependencies only. Besides this, if the dependence is *weak* (i.e., unlikely) or *hard* (very probable), its speculative status (and thus its motion) is encouraged or disallowed accordingly.

The second case is checking the candidate instruction from other basic block for the validity of interblock motion. When the candidate has no dependencies, it is placed in the ready list for regular scheduling, if it is exception-free instruction. This is not the case with memory loads, which are exception-risky. Such instruction will be considered for control speculation, if its execution probability is high. In this case the instruction will start the control speculative block.

3.4 Sorting the ready list

After the ready list is filled with the appropriate instructions, it is sorted according to the set of heuristics of the Haifa scheduler and the backend. Then the `max_issue` function is used to select from the list the first instruction, which if scheduled, will allow scheduling the maximal number of other instructions. This instruction is determined with limited depth backtracking using DFA interface.

Speculative instructions affect both the sorting rules and the `max_issue` logic. The follow-

ing heuristics are used for prioritizing speculative instructions in these functions:

- prefer a non-speculative instruction to speculative one;
- prefer a data speculative instruction to control speculative one;
- if both instructions are data speculative, prefer one with a smaller number of dependencies; break ties with preferring one with a greater number of weak dependencies (Section 4.4 explains when the dependencies are considered weak);
- if both instructions are potential control speculative ones, prefer the one with greater execution probability.

3.5 Backend support

Changes in target `.md` file are required to support generation of speculative instructions. In `ia64.md`, the speculative load is represented with `[parallel (set reg mem) (unspec lda/ldc)]` pattern. A check is represented with `[parallel (set reg mem) (unspec reg)]` pattern. The `unspec` part is used as a marker for a speculative instruction. It also prevents optimizers from breaking the `parallel` block. Using `set` allows backend to treat the instruction as an ordinary load, for example when bundling. `(unspec reg)` creates the dependence between the load and the check instructions. A latency of the check instructions is set to zero in DFA description. This allows other instructions to be placed in the same bundle as checks.

Besides, a number of hooks is added for proper cooperation between the scheduler and the backend. The hooks are as follows: `can_be_speculative_p(insn,status)` checks for possibility of an instruction to be speculative with the given status; `generate_recovery_code(insn,status)` splits an instruction on the load and the check part with the given status.

3.6 Current implementation status

We have implemented the general infrastructure for the speculation support. As of now, speculation of the integer and the float loads, and speculation of the predicated instructions (`COND_EXEC`) are supported. The current implementation does not support generation of a full recovery code. This disallows for speculating uses of control/data speculative loads. The control speculation is also not fully correct because of `chk.s` instruction, which requires the address of the recovery code to be specified. The current implementation specifies the address with zero offset (i.e. jump on self).

We have improved region formation in the interblock scheduler. The existing algorithm will only form a non-trivial acyclic region for a function that has no loops. If the function has a loop, then non-trivial regions can be formed from the loop body, and all other basic blocks will form single block regions. This significantly reduces the possibilities for interblock motions.

We have fixed this approach as follows. When the loops are selected, we perform additional traverse of the flow graph in topological order and search for the basic blocks that are not yet assigned to any region. We mark these blocks with their region numbers as follows. If all predecessors of block `X` are assigned to the same region `R`, then `X` is also assigned to `R`. If `X` belongs to `R`, and its successor `Y` doesn't belong to `R`, then all successors of `X` should not belong to `R`. The process of assigning regions to blocks is repeated until no changes are observed.

4 Memory disambiguation support

4.1 Overview

To benefit from data speculation support, good memory disambiguation is required. This means an aggressive alias analysis is to be performed. Furthermore, speculative scheduling makes demands for *hints* from aliasing machinery. The hints may be given for such pairs of memory references, which cannot be proven independent, but likely are (see Section 4.4). This technique is extremely useful when interprocedural analysis may not be performed [Muthukumar05].

The alias analysis on IA-64 is weakened by the absence of displacement. For such architectures, GCC uses pointer arithmetic to compute load and store pointers for e.g. array references. Then GCC fails to disambiguate these references because of different base registers. Consider the following example (inspired by [Gupta03]):

```
void foo (int *a, int c) {
    a[1] *= c;
    a[2] += c;
}
```

The addresses of array elements are computed as shown below. This prevents GCC from disambiguating memory stores to `r341` and `r346`, because the GCC aliasing doesn't track pointer arithmetic.

```
r342 [a] = r328 [sfp]
r341 = r342 + 4
<...>
r346 [a] = r328 [sfp] + 8
```

We implement a two-step solution for the problem. The alias analysis used in the scheduler is improved by propagating points-to information from the Tree SSA to the RTL level. Additionally, tracking of alias arithmetic is performed on the RTL level. Aliasing machinery also gives hints to the scheduler using a notion of *weak* dependency.

4.2 Propagating points-to data

Propagation of alias information needs support both on the Tree SSA and RTL levels. Points-to information gathered on tree-ssa for a pointer is merged from all SSA versions of the pointer. If there is a version of the pointer that hasn't points-to data computed, or the `pt_anything` flag is set, then the flag is also set on merged points-to set. Resulted set is saved in a hash table for the later use on the RTL level. This work is performed when quitting from the SSA form, so additional may alias pass is inserted before `pass_del_ssa`.

On the RTL level, tree expressions are associated with registers addressing MEMs. The links to the original tree expressions are used for disambiguation of MEMs in the `*_dependence` functions. Two cases are supported: both MEMs have the `REG_EXPR` links, or one MEM has the `REG_EXPR` link, and other one has the `MEM_EXPR` links. This allows to disambiguate the following cases:

- `ptr->field` and `*p`, if points-to sets for `ptr` and `p` doesn't overlap;
- `arr[index].field` and `*p`, if points-to set for `p` doesn't contain `arr`;
- `var.field` and `*p`, if points-to set for `p` doesn't contain `var`;
- `arr[index]` and `*p`, if points-to set for `p` doesn't contain `arr`.

More chances for disambiguation could be brought up with using structure aliasing and memory classes. The structure alias analysis generates *structure field tags* (SFTs) for all inner subobjects of a structure. The SFTs are kept as "subvariables" of a structure variable and are used to specify aliases with structure fields in the points-to sets. This information is saved in the hash table analogously to regular points-to data. The `get_subvars_for_var` routine is patched to look up subvariables not only in the `var_ann` field, but also in the hash table. Then it is used on the RTL level for another chance of disambiguation.

Using memory classes for disambiguation is partly supported by GCC with disambiguating global and stack variables. The `pt_malloc` attribute may be also used for disambiguation of stack and heap variables. Unfortunately, this attribute is not properly propagated in the Tree SSA alias analysis (as of March 2005).

4.3 Tracking pointer arithmetic

The patch targeted for tracking "base + offset" arithmetic was suggested by Sanjiv Gupta in 2003 both on the GCC Summit [Gupta03] and in the gcc-patches mailing list [GuptaGCC]. We use the infrastructure of the patch in our experiments. The patch handles offsets within lower *m* bits of abstract address value, where *m* equals to the bit size of the `HOST_WIDE_INT` type. That is, on IA-64 the patch is able to distinguish up to 64 distinct displacements. Each pseudoregister is mapped to an *address descriptor* at each program point. This structure contains a defining instruction for the pseudo and the set of possible displacements (called mod-*k* residues set, $k = 2^m$) represented as a bit set in a variable of the `HOST_WIDE_INT` type.

During the analysis stage, address descriptors are propagated through the `SET` and `PLUS` instructions, and merged in corresponding control flow points. Data flow information for the given program point is represented as a list of address descriptors and saved at the end of each basic block. The lists are then used when answering aliasing queries in the `true_dependence` function. Disambiguation routine finds defining instructions for the MEMs, computes valid data flow information for the instructions, and checks if corresponding address descriptors have the same base pseudo, and their residues sets do not overlap.

The patch has two major pitfalls. First of all, the algorithm suggested is very expensive. Consider the following loop:

```
for (i) {
    x = a[i − 1];
    <...>
    a[i + 1] = y;
}
```

The algorithm will iterate over this loop exactly *k* times (64 times on IA-64), because on each iteration mod-*k* residues set for `i` will change. The patch tries to handle such situations with looking at the `PLUS` instructions where destination and first source registers are the same. This doesn't work as expected because different registers are commonly used for holding value of `i` on subsequent iterations. Second pitfall is that the patch doesn't handle auto increment expressions, failing to adjust register value during data flow analysis.

The scheduler works only with acyclic regions of flow graph. Thus, data dependencies between loop iterations could be abandoned when scheduling. In the example above, the scheduler doesn't need to know that `a[i-1]` and `a[i+1]` may access the same memory locations between iterations. It is sufficient to know that these expressions are independent within a single iteration. Using this idea, we have modified the algorithm of data flow analysis used by

the patch. To minimize the number of required iterations, we traverse the flow graph in topological order and propagate the data flow from visited to unvisited nodes. In other words, we don't allow propagation of the data flow along backedges. This allows to complete the data flow analysis during one iteration, and computed information is exactly what is needed by the scheduler.

The patch was also fixed to handle auto increment expressions. The algorithm used the `note_stores` function to propagate the data flow through an instruction. This function doesn't notice auto increments. We use `for_each_rtx` to find auto increments in RTX and update data flow information. This change allows to use the patch in all `{true,output,anti}_dependence` functions.

4.4 Weak dependencies

As noted above, alias analysis could help greatly to speculative scheduling with giving hints for useful data speculation. The hints are organized in the form of weak dependencies. Weak dependency serves as an attribute of the true dependency. The following hints are used for deciding on weakness of a dependency:

- Two pointers have disjoint points-to sets, but one of the sets has the `pt_anything` flag set (see Section 4.2);
- Two pointers are different function parameters;
- Two pointers have different bases, that are different function parameters.

In contrast, a dependency should not be broken with data speculation, when two pointers are reported to have intersecting points-to sets. This designates high probability of a speculation failure. Other heuristics are proposed in [Muthukumar05], but they are covered either with structure aliasing or with the `-fstrict-aliasing` flag.

5 Probability analysis support

The interblock scheduler computes execution probability of each basic block relative to each other block when initializing data structures of the region. Execution probability of basic block X relative to block Y is used as a cutoff when instructions from X are considered for interblock motion into Y. Historically, evaluation of the probabilities in the scheduler is very inaccurate. Given basic block X, each outgoing edge from X is considered to have equal probability, if this edge doesn't leave the region. If some edges from X leave the region, then aggregate probability of taking these edges is assumed 10%, and remaining 90% is again distributed evenly among the rest of outgoing edges.

We fix this situation by using GCC framework for estimating execution frequency of a basic block. This is possible either with profile information (`-fbranch-probabilities`) or with prediction (`-fguess-branch-prob`, turned on at `-O2` and higher). An old scheme is used in all other cases.

Two hard-coded scheduler parameters influence the number of instructions considered for interblock motions. These are maximal instruction latency (which is 3) and minimal probability cutoff for basic blocks (which is 40%). A bigger first parameter and a smaller second parameter make the interblock motion more aggressive. We have introduced new scheduler parameters for using instead of old hard-coded values and adjusted the values according to the

results of the experiments. Increased minimal conflict delay (up to 5–6) shows more interblock motions than default value of three. Lowering the probability cutoff to 25–30% helps only when profile information is available.

6 Experimental results

Our work is based on HEAD 20050407 snapshot. This snapshot doesn't allow us to test all four proposed patches together. We need to schedule additional `pass_may_alias` just before quitting the SSA form to perform propagation of alias information. Unfortunately, it is not possible to run alias analysis after ivopts, because it is not always successful in correct update of aliasing information. This should be corrected soon by Diego Novillo. While the bug is still in the mainline, we are testing only speculation, pointer arithmetic, and probability patch together.

Tables 2–4 show the results of SPECINT runs with and without our patches. Base tuning is `-O2`, and peak tuning is `-O3` respectively. We have also tried the patches with tweaked inlining parameters. We approximately double the parameters, because aggressive inlining increases possibilities for speculation. In the tables we use "spec" to denote the speculation patch, "pa" to denote the pointer arithmetic patch, "prob" to denote the probability patch, and "inline" to denote the inlining patch. The reference compiler results are denoted with "ref."

The tested patches do not contain improved region heuristic and do not use weak dependencies as hints for speculation. The patches do not improve SPEC results in average when used as are. Tweaking inlining parameters allow patches to show 0.8% speedup in average when comparing to the tweaked compiler, and 1.97% speedup when comparing to the reference compiler. When looking at `-O3` results, it could be noted that our patches smooth a negative effect of aggressive inlining on certain benchmarks. This is because automatic inlining is turned on at `-O3`.

Benchmark	Old	New	Diff, %
164.gzip	622	619	–0.48
175.vpr	812	811	–0.12
176.gcc	915	912	–0.33
181.mcf	690	670	–2.90
186.crafty	862	858	–0.46
197.parser	717	719	+0.28
252.eon	646	649	+0.46
253.perlbmk	822	814	–0.97
254.gap	542	574	+5.90
255.vortex	883	890	+0.79
256.bzip2	671	669	–0.30
300.twolf	971	967	–0.41
SPECint_base2000	752	753	+0.13
164.gzip	656	655	–0.15
175.vpr	829	824	–0.60
176.gcc	913	910	–0.33
181.mcf	693	685	–1.15
186.crafty	854	854	~0.00
197.parser	772	774	+0.26
252.eon	655	654	–0.15
253.perlbmk	828	800	–3.38
254.gap	541	574	+6.10
255.vortex	893	901	+0.90
256.bzip2	674	674	~0.00
300.twolf	970	964	–0.62
SPECint2000	763	763	~0.00

Table 2: ref vs. spec+pa+prob, base=`-O2`, peak=`-O3`

Table 5 contains newer results achieved for SPECFP at `-O2` on HEAD 20050503 snapshot. In this table the probability patch is included in the speculation patch, and the alias patch contains both the pointer arithmetic patch and the

Benchmark	Old	New	Diff, %
164.gzip	620	620	~0.00
175.vpr	808	810	+0.25
176.gcc	913	914	+0.11
181.mcf	686	684	–0.29
186.crafty	863	859	–0.46
197.parser	713	720	+0.98
252.eon	716	724	+1.12
253.perlbmk	819	816	–0.37
254.gap	539	574	+6.49
255.vortex	878	891	+1.48
256.bzip2	668	670	+0.30
300.twolf	970	968	–0.21
SPECint_base2000	756	762	+0.79
164.gzip	655	656	+0.15
175.vpr	841	845	+0.48
176.gcc	899	900	+0.11
181.mcf	693	694	+0.14
186.crafty	863	857	–0.70
197.parser	772	778	+0.78
252.eon	750	755	+0.67
253.perlbmk	824	822	–0.24
254.gap	539	577	+7.05
255.vortex	889	907	+2.02
256.bzip2	680	681	+0.15
300.twolf	975	971	–0.41
SPECint2000	772	778	+0.78

Table 3: inline vs. inline+spec+pa+prob, base=`-O2`, peak=`-O3`

Benchmark	Old	New	Diff, %
164.gzip	622	620	–0.32
175.vpr	812	810	–0.25
176.gcc	915	914	–0.11
181.mcf	690	684	–0.87
186.crafty	862	859	–0.35
197.parser	717	720	+0.42
252.eon	646	724	+12.07
253.perlbmk	822	816	–0.73
254.gap	542	574	+5.90
255.vortex	883	891	+0.91
256.bzip2	671	670	–0.15
300.twolf	971	968	–0.31
SPECint_base2000	752	762	+1.33
164.gzip	656	656	~0.00
175.vpr	829	845	+1.93
176.gcc	913	900	–1.42
181.mcf	693	694	+0.14
186.crafty	854	857	+0.35
197.parser	772	778	+0.78
252.eon	655	755	+15.27
253.perlbmk	828	822	–0.72
254.gap	541	577	+6.65
255.vortex	893	907	+1.57
256.bzip2	674	681	+1.04
300.twolf	970	971	+0.10
SPECint2000	763	778	+1.97

Table 4: ref vs.inline+spec+pa+prob, base=`-O2`, peak=`-O3`

propagation patch. "Data" and "Control" denote the speculation patch with only data or control speculation enabled, respectively. "All" denotes the merged patch. The best achieved results for SPECFP, which are at `-O3` with only the speculation patch enabled, and denoted as "Best," are provided for clarity.

7 Conclusions

In this paper we have presented the results of the project of improving GCC instruction scheduling for IA-64 platform. We have chosen to implement the framework for speculation support as the main goal of the project. Other infrastructure changes are needed to make this framework profitable. The changes should improve alias analysis on the RTL level. We proposed to propagate points-to information from the Tree SSA to RTL level, and to track pointer arithmetic within `alias.c`. The latter change should compensate for the absence of displacement on IA-64. During our work on the project we have implemented some other scheduler improvements.

Benchmark	Data	Control	Spec	Alias	All	Best (Spec at `-O3`)
168.wupwise	+0.71	+1.43	+1.43	+1.66	+0.95	–0.47
171.swim	–0.30	–0.30	+0.30	–0.44	–0.15	–0.15
172.mgrid	0.00	0.00	+0.30	+4.79	+5.09	–4.84
173.applu	+0.24	0.00	+1.18	–0.71	–0.24	+0.95
177.mesa	–1.09	0.00	+2.89	+0.14	+0.82	+1.73
178.galgel	+2.51	–5.75	+2.51	–5.92	–3.41	+8.76
179.art	+1.05	+0.06	–0.17	–0.06	+0.58	–0.23
183.equake	–0.90	–0.23	–1.13	–0.23	–0.45	–2.24
187.facerec	+0.19	0.00	–1.12	–3.16	–2.99	+2.87
188.ammp	+18.84	+0.15	+18.84	–1.52	+16.87	+20.68
189.lucas	+0.12	–0.12	–0.36	–0.12	0.00	0.00
191.fma3d	+0.73	–0.36	+0.36	+0.73	+2.93	–0.72
200.sixtrack	+2.43	0.00	+2.08	–1.04	+1.04	+3.16
301.apsi	+1.12	–0.67	+0.45	–4.45	–3.13	+5.57
SPECfp2000	+1.71	–0.57	+1.89	–0.76	+1.14	+2.46

Table 5: SPECFP results at `-O2`, % to the reference GCC

The current experimental results (yet with unfinished patch) show that we have taken the right direction when trying to get the performance improvements for IA-64 with fixing instruction scheduling. We hope that our next long term project in this direction will be implementing a new interblock scheduler for GCC. We think that the new scheduler should follow one of DAG approaches with support of instruction cloning. The parts of the framework we created in the current project will be reused in the new scheduler.

8 Acknowledgments

We'd like to thank Vladimir Makarov for guiding us in the wonderful world of GCC hacking. This project would not exist without his consulting. We would like to thank Diego Novillo and James Wilson for answering our questions and giving helpful comments. And we thank the Gelato Federation and HP for giving us the possibility to work on improving GCC and for their attention to the GCC project.

References

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

[Gupta03] Sanjiv K. Gupta, Naveen Sharma. Alias Analysis for Intermediate Code. Proceedings of GCC Summit, June 2003.

[GuptaGCC] `http://gcc.gnu.org/ml/gcc-patches/2003-06/msg01764.html`

[Intel] Intel Itanium Architecture Software Developer's Manual. Volume 1: Application Architecture. Volume 3: Instruction Set. `http://www.intel.com/design/itanium/manuals/iiasdmanual.htm`

[Muthukumar05] Kalyan Muthukumar. Compiling for Intel Itanium processor.

Presented on Gelato GCC Workshop, January 2005.

[Novillo05] Diego Novillo. Private communication. 2005.

[Wilson05] James E. Wilson. Private communication. 2005.