

An Empirical Study of Data Speculation Use on the Intel Itanium 2 Processor

Markus Mock

Ricardo Villamarín

José Baiocchi

Department of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260
Email: mock@cs.pitt.edu

Department of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260
Email: rvillsal@cs.pitt.edu

Department of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260
Email: baiocchi@cs.pitt.edu

Abstract—The Intel Itanium architecture uses a dedicated 32-entry hardware table, the *Advanced Load Address Table (ALAT)* to support data speculation via an instruction set interface. This study presents an empirical evaluation of the use of the ALAT and data speculative instructions for several optimizing compilers. We determined what and how often compilers generated the different speculative instructions, and used the Itanium’s hardware performance counters to evaluate their run-time behavior. We also performed a limit study by modifying one compiler to always generate data speculation when possible. We found that this aggressive approach significantly increased the amount of data speculation and often resulted in performance improvements, of as much as 10% in one case. Since it worsened performance only for one application and then only for some inputs, we conclude that more aggressive data speculation heuristics than those employed by current compilers are desirable and may further improve performance gains from data speculation.

I. INTRODUCTION

For the past decade, one of the main sources of increased computer system performance has been found in instruction level parallelism (ILP). However, exploiting additional processor resources faces two particular obstacles: control and data dependences. For example, without any form of speculation, execution cannot pass a conditional branch until the branch outcome has been determined, so that it can be difficult to keep all the functional units of the processor busy. To overcome the limitations on ILP imposed by control dependences, various forms of control speculation in the compiler and processor have been conceived. For example, computer architects have added speculative load instructions to the instruction set architecture (ISA) to enable the compiler to schedule a load before a conditional branch without changing program semantics. Moreover, control speculation in the form of speculative execution (i.e., via speculation hardware, such as branch predictors) is performed by nearly all high performance processors today. To overcome limitations imposed by data dependences, several data speculation approaches have been developed recently [1], [2], [3], [4].

In data speculation, a computation is performed speculatively based on the value of some data item (e.g., a variable), that is not yet available. In particular, a memory load and dependent computations may be executed before a possibly aliased store. Consider the example in Figure 1. The original

```
// other instructions
st8    [r4] = r12
ld8    r6 = [r8];;
add    r5 = r6, r7;;
st8    [r18] = r5
...

ld8.a  r6 = [r8];; // advanced
// other instructions      load
add    r5 = r6, r7;;
// other instructions
st8    [r4] = r12
chk.a  r6, recover
back:
st8    [r18] = r15
...

// somewhere else in program
recover:
ld8    r6 = [r8];;
add    r5 = r6, r7
br     back

(a) before data speculation      (b) after data speculation
```

Fig. 1. Data speculation example. The load into register r6 and the dependent load instruction are speculatively executed before the possibly aliased store. Before the loaded value is used, a special check instruction is used that branches to recovery code to redo the load and add in case the store wrote the same memory location.

code performs a store, then loads a value from memory, uses the value to perform an addition, and finally stores the result back to memory. A data dependence between the first store and the subsequent load must be assumed unless the store and load are known to definitely access different memory locations. Therefore, the load instruction cannot be hoisted above the store to (1) hide the load latency, and (2) make use of possibly available machine resources before the store

Data speculation works by breaking this data dependence and speculatively performing the load and dependent computations. In the example, which shows Intel Itanium assembly code, a load of one operand of an add instruction is hoisted above a possibly aliased store using a special data-speculatively load instruction, called an *advanced load*. After the store, a special check instruction (*chk.a*) is used to determine if misspeculation occurred: if the store modified the same memory location accessed by the load, the check instruction branches to recovery code. The compiler-generated recovery code repeats the load and the add instruction and then branches back to resume normal execution.

To detect misspeculation, the Intel Itanium processor [5] uses a dedicated hardware structure, the *Advanced Load Address Table (ALAT)*, a 32-entry fully-associative table that

Instruction	Operation
ld.a, ldf.a, ldfp.a	GPR and FPR advanced load
ld.c.clr, ld.c.clr.acq, ldf.c.clr, ldfp.c.clr	GPR and FPR check load that clear the ALAT on a hit
ld.c.nc, ldf.c.nc, ldfp.c.nc	GPR and FPR check instructions that reallocate the ALAT entry on miss
ld.sa, ldf.sa, ldfp.sa	GPR and FPR (control-) speculative advanced load
chk.a.clr, chk.a.nc	clearing and nonclearing GPR and FPR advanced load check
invala, invala.e	invalidate all and invalidate individual entry for a GPR or FPR, respectively

TABLE I

DATA SPECULATION INSTRUCTIONS ON THE INTEL ITANIUM PROCESSOR. GPR AND FPR REFER TO GENERAL PURPOSE AND FLOATING POINT REGISTERS, RESPECTIVELY.

holds the addresses of data speculative loads (like the `ld8.a` in the example) and their target registers. Every store instruction clears a matching entry from the ALAT, i.e., store instructions to the address of a previous data-speculative (or *advanced*) load will remove that advanced load’s entry. The `chk.a` detects misspeculation by doing a lookup in the ALAT using the target register of the load. If no matching entry is found, a store must have occurred to the same address as the speculative load and recovery becomes necessary.

For this paper, we performed an empirical study to determine how well current state-of-the-art optimizing compilers do in using the data speculation instructions provided by the Itanium architecture. In particular, we measured how often data-speculative instructions are generated, and how they exercise the ALAT table. We wanted to find out how frequent data speculative instructions typically fail during execution and whether capacity misses occur often in practice. Since our study is empirical and based on actual compilers, we also wanted to find out whether current compiler technology is able to make beneficial use of the hardware features provided by Itanium. Specifically, this paper makes the following contributions:

- we empirically evaluate how often state of the art compilers exploit data speculation for SPEC CPU2000 benchmarks by examining their use of data speculation instructions;
- we use hardware performance counters to measure the frequency of misspeculation, and how often misspeculation is due to capacity misses in the ALAT table;
- we performed a limit study for the potential of data speculation by modifying the heuristics used in the Open Research Compiler to always generate data speculation when feasible; the modified aggressive data speculation frequently resulted in improved performance;
- and finally, we demonstrate that generally, current compilers are too conservative in the application of data speculation on the Itanium, which indicates that better speculation heuristics may be possible.

The rest of the paper is organized as follows: Section II provides an overview of the data speculation support provided by the Intel Itanium architecture. Section III describes our experimental setup and the programs we used. Section IV describes the results we obtained; Section V discusses related

work, and finally Section VI concludes.

II. DATA SPECULATION ON THE INTEL ITANIUM PROCESSOR

The Intel Itanium processor provides several instructions to enable data speculation by the compiler. First, both integer and floating point load instructions of different sizes are available in the form of *advanced* loads. In addition to loading the type of value indicated, they also allocate an entry in a special hardware table, the Advanced Load Address Table (ALAT). For that purpose, a register tag based on the physical target register of the load is computed and stored along with the size of the load, and a tag derived from the physical memory address accessed by the load instruction. If there’s already an entry with the same register tag in the ALAT, that entry is removed first.

The intended use of an advanced load instruction is for it to be performed before a possibly aliased (and therefore data-dependent) store instruction, thereby speculatively breaking a potential data dependence. Therefore, a data-speculative sequence of instructions consists of an advanced load, and possibly some instructions that are dependent on the value of that load. In addition, to ensure correctness, the speculative sequence must be concluded by some form of a check instruction that detects misspeculation, and code to recover from misspeculation.

The Itanium processor provides two forms of check instructions: a load check instruction `ld.c` and a more general check and recover instruction `chk.a`. The first form reloads the value if the register tag of the target register is not found in the ALAT, i.e., it performs checking and recovery in one instruction. It is useful if only the load itself was speculative and no instructions dependent on the loaded value were performed. In the latter case, the `chk.a` instruction must be used that branches to recovery code if no entry matching the register tag of the indicated check register is found in the ALAT. The recovery code must reload and recompute any speculatively performed operations and will then typically jump back to resume execution right after the check instruction (as shown in the example in Figure 1).

Table I lists the full set of data speculation instructions provided by the Itanium architecture. `ld.a`, `ldf.a`, and `ldfp.a` are advanced load instructions of integers, floating point values, and pair of floating point values. They load the value and

	Source Lines	Description
equake	1,513	seismic wave propagation simulator
mcf	1,909	combinatorial optimization
bzip2	4,639	compression
gzip	7,757	compression
parser	10,924	word processing
ampp	13,263	molecular dynamics
vpr	16,973	circuit placement and routing
crafty	19,478	chess program
twolf	19,748	placement and global routing
mesa	49,701	graphics
vortex	52,633	object-oriented database
gap	59,482	group theory interpreter

TABLE II
SIZES AND DESCRIPTIONS OF THE PROGRAMS USED IN THE EXPERIMENTS.

allocate an entry in the ALAT with the register tag of the target destination register.¹ The Itanium allows the combination of data and control speculation and provides `ld.sa`, `ldf.sa`, and `ldfp.sa` for that purpose. In addition to allocating an entry in the ALAT, they also generate an exception token [7] that is used to generate an exception once it is determined that the instruction is definitely executed.

The load check instructions are provided in two basic forms: those that clear the ALAT entry after the check (indicated by the `.clr` suffix in the instruction mnemonic). The compiler will use this version if there is no further use of the ALAT entry. Deallocating the ALAT entry decreases the likelihood that a still useful entry may be replaced when the ALAT becomes full. In the `.nc` variant, the ALAT entry is not cleared and can therefore be checked again. This is particularly useful in loops. The compiler may speculatively identify a load instruction as loop invariant and hoist it before the loop. In the loop, a load check instruction can then be used to ensure a correct value in the target register if the load turns out not to be loop invariant. Similar to the load check instructions, the `chk.a` instruction also comes in two flavors, one preserving (`.nc`) and one deleting the ALAT entry (`.clr`). Finally, the instructions `invala` and `invala.e`, where `e` denotes a general purpose or floating point register invalidate all or a specific ALAT entry (the one with the matching register tag). This is useful for explicit management of ALAT contents. For example, the operating system uses `invala` to clear the ALAT table upon context switches.

III. EXPERIMENTAL SETUP AND WORKLOAD

To determine how current optimizing compilers for the Itanium architecture use its data speculation instructions and hardware, we used the SPEC CPU2000 [8] benchmarks, which are commonly used to evaluate CPU and compiler performance. Since only C and C++ benchmarks (the majority of the suite) can be compiled by all the compilers we used, we performed our experiments only for them. `gcc` and `art`

¹For the `ldfp.a` instruction, which loads a pair of floating point values, only one of the registers is used to compute the register tag.

were excluded because for them, some compiler configurations (e.g., when using interprocedural optimization) produced incorrect code.

A. Experimental Procedure

We produced the program binaries for the different configurations and computed static counts of the data speculative instructions by counting how many advanced load, load check, and check instructions were generated in each case (cf. Table III). To characterize the dynamic data speculation behavior and use of the ALAT, we used the `perfmon` tool [9], version 2.0 for Linux kernel version 2.4. `Perfmon` uses the Itanium’s hardware performance monitoring counters to obtain information about dynamic program performance. Specifically, we used it to count the following events:

- the total number of `ld.c` or `chk.a` check instructions that were performed (i.e., the `INST_CHKA_LDC_ALAT_ALL` counter);
- the number of those check instructions that failed (i.e., the `INST_FAILED_CHKA_LDC_ALAT_ALL` counter); and finally,
- the number of ALAT capacity miss events that occurred, counted as the number of times a new ALAT entry replaced a still active one (i.e., the `ALAT_CAPACITY_MISS_ALL` counter).²

We compiled and ran the program on an otherwise unloaded Hewlett Packard ZX6000 workstation, with two 1.3 GHz Itanium 2 processors, 1 GB of RAM, running Redhat Linux 7.2 (Advanced Workstation Edition). The program were run 20 times and we computed 95% confidence intervals shown as error bars in the graphs in Section IV. For the execution time results in Figure 2, we ran the benchmarks only ten times to save time (but still computed 95% confidence intervals).

B. Programs

Table II lists the programs that we used to conduct our experiments. The programs range from about 1,000 lines of

²Note that this is not the same as the number of advanced load checks that missed in the ALAT due to capacity; that number may be potentially higher since the replacement of one entry may result in several failing check instructions.

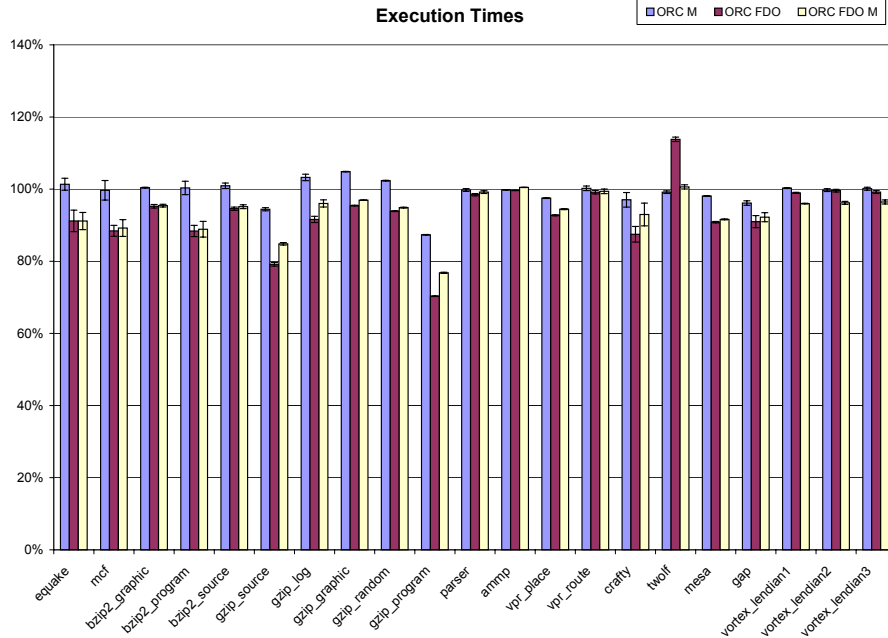


Fig. 2. Relative execution times of the different versions of ORC. The execution time of the standard ORC version has been normalized to one and is not shown in the graph. The three bars are for ORC with our modifications (ORC M), the original ORC with FDO, and ORC with our modification and FDO (ORC FDO M).

code to approximately 60,000 lines of code and cover a wide range of tasks. For our experiments, we ran the benchmarks on their SPEC-supplied reference inputs. Some benchmarks process multiple input files as part of their reference runs. Since the different inputs sometimes revealed different (runtime) data speculation characteristics, the figures in Section IV indicate on which inputs the respective benchmarks were run. For example, `bzip2_program` indicates that the application `bzip2` was run on input `program`.

C. Compilers

We used three compilers to conduct our experiments, GNU gcc version 3.4.2, version 2.1 of the Open Research Compiler (ORC) [6], and Intel’s commercial ecc compiler, version 7.1. We compiled the applications in two configurations. The first used the highest optimization levels (`-O3`) but no feedback-directed optimization. The second configuration produced an executable based on profile information by first compiling the programs, then running them on the SPEC-provided training (`train`) inputs, and then compiling them using the profile information obtained during this profile run. In our graphs in Section IV, `ECC` refers to the configuration without feedback information and `ECC FDO` to the configuration with feedback-based optimization (and analogously for ORC). In all cases, we performed interprocedural optimization using the `-ipa` switch for the ORC compiler, and the `-ipo` switch for the ecc compiler.

D. Limit Study

To determine whether more aggressive generation of data-speculative instructions would exercise the ALAT structure more aggressively, and to determine whether there is room for better data speculation heuristics, we also modified ORC in several places to force it to generate data-speculative instructions whenever possible without compromising program correctness.

Specifically, we modified ORC’s heuristic in the following ways:

- 1) ORC uses a control variable `cutting_set_size` to decide whether to generate data speculation in some cases. This variable is equal to one more than the number of compensation (basic) blocks that are required to compensate for the effects of mispeculation. We commented out the comparison with this variable in the scheduler (which handles the conversion of normal to data-speculative loads) to force the generation of data speculation regardless of the number of compensation blocks required.
- 2) Dependences between stores and loads that are not definite, i.e., that are not guaranteed to access the same memory are normally treated as “relaxed” dependences, which makes them eligible for data speculation. For instance, a store through a pointer followed by a load through another pointer will typically create an indefinite dependence (unless the static alias analysis is able to rule

out any possibility of accessing the same memory). Even though they are indefinite, some edges are normally treated as unrelaxed edges by ORC, for instance, when the scheduling heuristic indicates that the earliest time the dependent load can be scheduled is later than the latest time it can be scheduled without making the schedule longer. We modified the code to always treat indefinite dependences as relaxed dependences to force data speculation for them.

- 3) Finally, ORC uses a profitability heuristic to assess whether data speculation might be profitable. This heuristic, for instance, limits the number of data dependences that a speculative load is allowed to violate, and speculation is not performed beyond that maximum. We changed the profitability heuristic to always return `true`, i.e., assume data speculation is always profitable when the profitability heuristic is consulted.

IV. RESULTS

In our experiments we found that `gcc` is currently unable to exploit the data speculation features of the Itanium processor and never generates any data-speculative features. For none of the benchmarks, `gcc` generated any data-speculative instructions. We checked the GNU website (<http://www.gnu.org/>), which revealed that the previously developed support for data speculation had been found to be too immature to be integrated into the `gcc` release. The discussion in the subsequent sections therefore discusses only the results for the other compilers.

A. Static Data Speculation Results

Table III shows the static number of advanced load, load check and and general check (`chk.a`) instructions that were generated for the compilers used in our study in their different configurations. Based on the static count of data-speculative instructions generated, it appears that ORC is usually more aggressive when it comes to data speculation. Based on the number of advanced load instructions, in 8 out of the 12 applications ORC generates a higher number of speculative instructions than `ecc`; in some cases, a considerably higher number. For instance, for `vpr` 160 compared to 9, and for `twolf` 765 instead of 53. When FDO is used, `ecc` is even more careful and now for all applications ORC generates more advanced loads than `ecc`. `ecc` seems to be particularly careful to avoid failed data speculation and the associated performance penalty.

Turning on feedback-directed optimization in ORC, decreases speculation in about half of the cases and increases it for the other half. For instance, for `ammp`, instead of 138 only 102 advanced loads are generated but for `gzip` 27 instead of one are generated. For ORC, we also looked at the overall performance impact resulting from using FDO. Figure 2 shows the relative execution time of the different ORC configurations that we used, with the unmodified ORC execution time (without FDO) normalized to one. The second bar for each application shows the relative execution time of ORC with FDO (without our modifications). As can be seen in

the graph, except for `twolf` the programs perform better with ORC when FDO is used; `twolf` slows down by somewhat under 20%. Whether the slowdown is due to the less aggressive data speculation (594 versus 765 advanced load) instructions, is not clear, although Figure 3 seems to suggest that this is not the case as the fraction of failed checks at run time is roughly the same. Furthermore, when data speculation became more aggressive, it seems that it usually improved performance, or (since other factors may be contributing) at least did not have a significant negative impact, since the percentage of failed speculation did not change between ORC and ORC with FDO (cf. Figure 3).

Table III also shows the effect of the modifications to ORC that we made. For all applications, the generation of data speculation whenever possible, increases the number of data-speculative instructions. Figure 4 shows the effect of our changes graphically. With the sole exception of `equake`, data speculation increased significantly, for `gzip` by more than five times. As we'll see below, this improved performance for some inputs (over 10% when compressing a program), but slowed down performance for others (e.g., a slowdown of about 5% for compressing a log file).

B. Dynamic Data Speculation Results

1) *Results for the ecc Compiler:* The dynamic performance counter data confirms that the `ecc` compiler appears to be generally quite careful when generating data speculation, in particular, when no profile information is used. Figure 5 shows that the misspeculation rate for `ecc` without FDO is generally below 0.5%, and only for `vortex` somewhat over 1%. The misspeculations go up when FDO is used, i.e., `ecc` is then more aggressive. Generally, this did not have a negative impact on overall performance.

Figure 6 shows how often entries in the ALAT were replaced by new entries because it was full as a fraction of the total number of data speculation checks performed. Only for the floating point applications `equake` and `mesa` we did see a significant number of them; for the former only when FDO was used.

2) *Results for the ORC Compiler:* Compared to the `ecc` compiler, ORC in general is more aggressive in the use of data speculation. `crafty`, stands out with over 16% of misspeculation as shown in Figure 3. `mcf`, `bzip2`, `parser`, and `ammp` are other applications that show at least about 1% misspeculation, whereas the other applications show almost no misspeculation. Together with `vpr` (for the `place` input), `crafty` shows ALAT capacity misses. However, compared to the total number of ALAT checks, those still account for only 0.1% (691,000 capacity misses versus 542,350,000 executed check instructions). They are further reduced when FDO is used, which indicates that ORC is able to take advantage of profile information to apply speculation more judiciously.

C. Results for the Limit Study

As discussed in Section IV-A, the modified version of ORC that generates data speculation as much as possible,

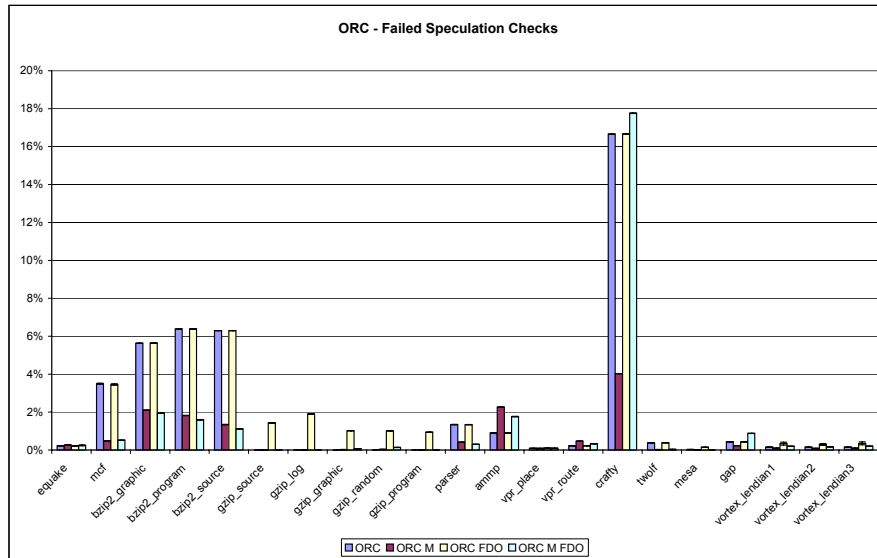


Fig. 3. The graph shows the number of failed check instructions (either `chk.a` or `ld.c`) as a fraction of the total number of checks executed for the ORC compiler, both with and without feedback-directed optimization.

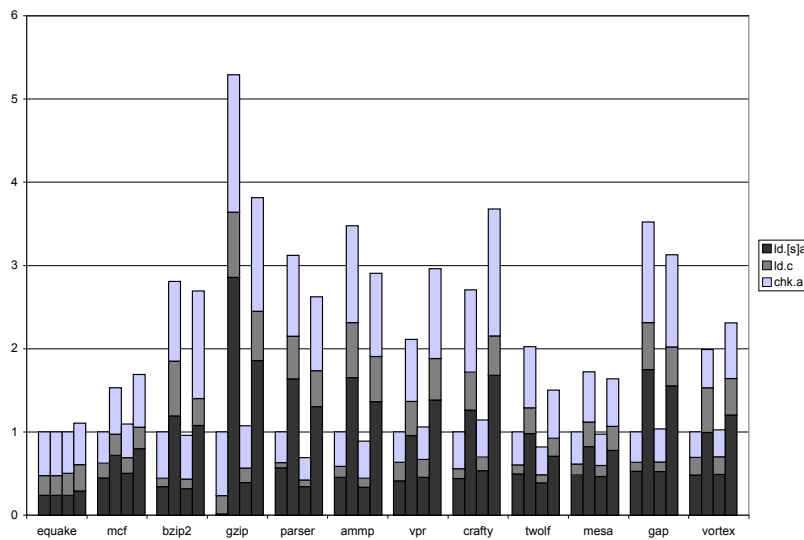


Fig. 4. This graph shows the relative number of (static) data-speculative instructions generated for the different ORC compiler configurations. For each application, the configurations are (from left to right): ORC unmodified, ORC unmodified with FDO, ORC with our modifications, and finally, ORC with our modifications and FDO. For example, for `gzip` the modified version of ORC generated more than five times as many data-speculative instructions than the unmodified version when no FDO was used, and almost four times as many when FDO was used.

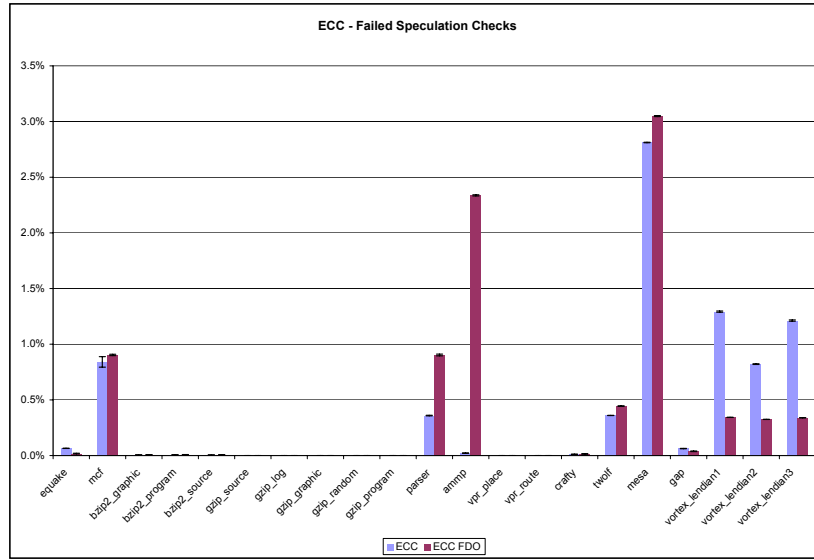


Fig. 5. The graph shows the number of failed check instructions (either `chk.a` or `ld.c`) as a fraction of the total number of checks executed for the ecc compiler, both with and without feedback-directed optimization.

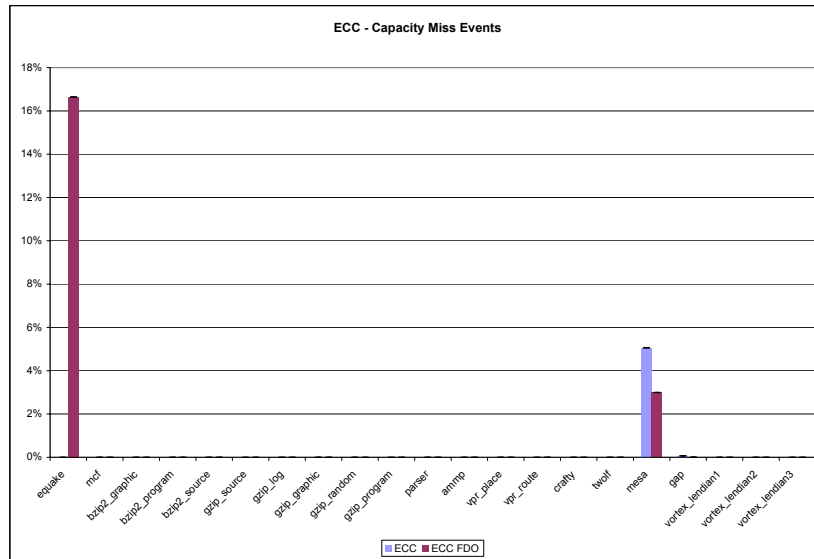


Fig. 6. The graph shows the number of ALAT capacity miss events as a fraction of the total number of data speculative checks performed for the ecc compiler, with and without feedback-directed optimization.

App	ecc			orc			orcM		
	ld.[s]a	ld.c	chk.a	ld.[s]a	ld.c	chk.a	ld.[s]a	ld.c	chk.a
quake	12	0	14	9	9	20	9	9	20
mcf	37	6	36	33	13	28	53	19	41
bzip2	30	0	30	41	12	67	143	79	115
gzip	0	0	0	1	15	53	197	54	114
parser	233	100	120	414	46	271	1196	374	712
ampp	39	8	38	138	40	127	503	202	355
vpr	9	3	6	160	86	142	370	159	290
crafty	38	26	12	201	54	204	578	210	454
twolf	53	24	31	765	166	611	1511	475	1133
mesa	1767	577	1324	1975	545	1582	3369	1212	2471
gap	3334	1153	2096	1323	270	922	4398	1413	3043
vortex	1279	380	983	911	404	581	1884	1015	874

App	eccfdo			orcfdo			orcfdoM		
	ld.[s]a	ld.c	chk.a	ld.[s]a	ld.c	chk.a	ld.[s]a	ld.c	chk.a
quake	0	0	0	9	10	19	11	12	19
mcf	22	5	17	37	14	30	59	19	47
bzip2	4	0	4	38	14	63	129	39	155
gzip	0	0	0	27	12	35	128	41	94
parser	188	122	66	250	58	197	951	316	649
ampp	23	5	22	102	33	136	415	166	305
vpr	8	2	6	175	85	151	536	194	418
crafty	8	1	5	244	75	205	770	217	701
twolf	36	6	30	594	152	516	1088	335	893
mesa	2035	834	1263	1897	540	1555	3180	1191	2339
gap	774	145	609	1318	282	998	3910	1172	2785
vortex	1220	386	1003	924	403	613	2279	828	1273

TABLE III

THIS TABLE SHOWS THE NUMBER OF DATA-SPECULATIVE INSTRUCTIONS GENERATED FOR THE DIFFERENT COMPILERS AND CONFIGURATIONS. `ecc` AND `orc` REFER TO THE INTEL AND OPEN RESEARCH COMPILER, RESPECTIVELY; `eccfdo` AND `orcfdo` WHEN THESE COMPILERS WERE USED WITH FEEDBACK-DIRECTED OPTIMIZATION; `orcM` AND `orcfdoM` IS ORC MODIFIED TO GENERATE DATA SPECULATION WHENEVER POSSIBLE, I.E., WITH THE MODIFICATIONS DESCRIBED IN SECTION III-D.

produces code with a significantly higher (static) number of data-speculative instructions (up to five times as many as shown in Figure 4. This is also reflected in the number of advanced load checks that are executed at run time, shown in Figure 7. We see a drastic increase in the total number of load check instructions that are executed compared to the original version of ORC, regardless whether FDO is used or not. Interestingly, however, the number of misspeculations does not go up proportionally. In fact, for almost all applications the misspeculation ratio goes down considerably (cf. Figure 3).

Often, this also results in reduced execution time, as shown in Figure 2. Application `gzip` ran about 10% faster (on one input), `vpr` and `crafty` showed some small improvements, as did `gap` and `mesa`. For the others, the additional speculation did not result in any noticeable performance changes. Only `gzip` showed some slowdowns, but not on all inputs: compressing a program and source code ran faster, whereas compressing the other inputs ran slower but only by a few percent.

This indicates that the currently used heuristic to generate data speculation can be improved and even though it is already more aggressive than the one used by the `ecc` compiler, being more aggressive still can further improve execution times. The number of capacity miss events seems to support this: while the fraction of capacity miss events goes up significantly, in general it seems to be not stressing the ALAT excessively

(cf. Figure 8). However, it would be interesting to study the effects on overall performance with a larger ALAT table so that misspeculation due to capacity problems would be eliminated.

V. RELATED WORK

Many researchers have studied data speculation and how it can be used to improve ILP and performance in general [1], [2], [3], [4], [10], [11], [12], [13], [14]. Zhai et al. [10], [11] have looked at data speculation to enable aggressive thread-level speculation. They schedule past data dependences, which is similar to the use of advanced load and check instruction on the Itanium processor. However, their recovery mechanism is based on rollback (rewind) of execution to the point before the data dependence violation occurred.

August et al. [14] evaluate a hardware structure, the memory conflict buffer, which could be seen as a precursor of the ALAT. Like the ALAT, it is used to support data-speculative instructions. It is evaluated on a synthetic architecture based on the HP PA-7100 processor and shown to achieve modest performance gains. The paper does not give details on how many data speculation instructions were generated and how much the memory conflict buffer was exercised by them.

González and González [13], [3] were one of the first to study the possible ILP gains resulting from data speculation. In [13] they specifically look at predicting a load address

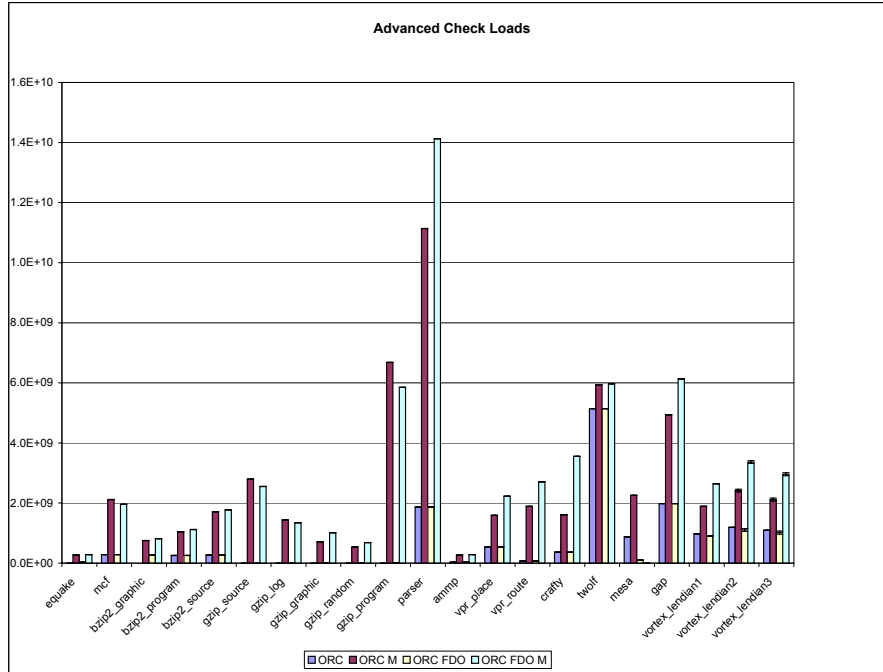


Fig. 7. This graph shows the total number of dynamically executed advanced load check instructions (ld.c or chk.a) for the different configuration of the ORC compiler. ORC M refers to the version of ORC we modified as described in Section III-D, and FDO to the configuration where feedback-directed optimization was used.

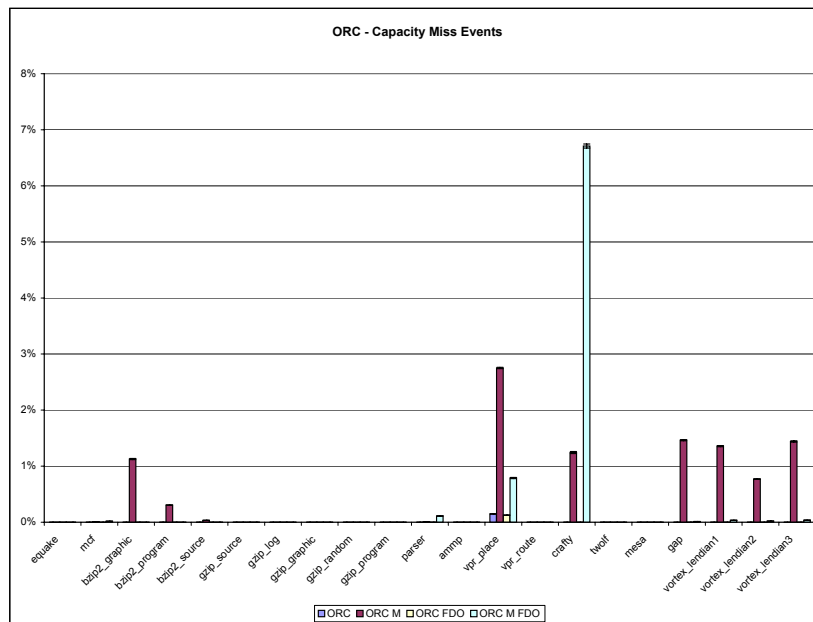


Fig. 8. The graph shows the number of ALAT capacity miss events as a fraction of the total number of data speculative checks performed for the ORC compiler, with and without feedback-directed optimization.

and speculatively performing the load and any data dependent instructions. In their experiments, they found performance gains of up to 35% resulting from this approach. In their follow-up work [3], they evaluate specific value predictors for their performance potential in supporting this style of program optimization. This work was later extended to multithreaded architectures [12].

Lin et al. [15], [16], [17], use the Itanium's data speculations support to perform more general speculative optimizations. In particular, they extend Chow et al.'s extension of static single assignment form (called HSSA [18]) to incorporate speculative alias information obtained by alias profiling. This allows their compiler (an extension of ORC) to speculatively perform partial redundancy elimination, possibly eliminating redundancies that are only redundant if certain memory operations are not aliased. Through a creative use of advanced load, load check and check instructions, their approach ensures the correctness of speculatively optimized code even when the alias relationships during actual program execution differ from the profiled alias relationships. As a result, their compiler tends to make more aggressive use of the data speculation instructions than the compilers examined in this study.

VI. CONCLUSIONS AND FUTURE WORK

In our evaluation of data speculation use by current state of the art compilers for the Intel Itanium architecture, we found that generally, compilers appear to be quite conservative and almost never oversubscribe the ALAT by performing too much speculation. Our modifications to ORC's data-speculation heuristic that generates data speculation whenever possible, resulted in much higher data speculation and frequently improved performance (of up to 10%) and negatively impacted performance only for one application (for some inputs). In the future, we would like to find a practical heuristic that approaches the performance of the "always speculate" modification but avoids over-speculation as much as possible.

REFERENCES

- [1] Y. Sazeides, S. Vassiliadis, and J. E. Smith, "The performance potential of data dependence speculation & collapsing," in *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*. IEEE Computer Society, 1996, pp. 238–247.
- [2] A. Moshovos, S. E. Breach, T. N. Vijaykumar, and G. S. Sohi, "Dynamic speculation and synchronization of data dependences," in *Proceedings of the 24th annual international symposium on Computer architecture*. ACM Press, 1997, pp. 181–193.
- [3] J. González and A. González, "The potential of data value speculation to boost ilp," in *Proceedings of the 12th international conference on Supercomputing*. ACM Press, 1998, pp. 21–28.
- [4] L. Hammond, M. Willey, and K. Olukotun, "Data speculation support for a chip multiprocessor," in *Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*. ACM Press, 1998, pp. 58–69.
- [5] J. H. Crawford, "Guest Editor's introduction: Introducing the Itanium processors," *IEEE Micro*, vol. 20, no. 5, pp. 9–11, Sept./Oct. 2000. [Online]. Available: <http://dlib.computer.org/mi/books/mi2000/pdf/m5009.pdf>; <http://www.computer.org/micro/mi2000/m5009abs.htm>
- [6] "Open research compiler, version 2.1," 2003, <http://ipf-orc.sourceforge.net/>.

- [7] R. Zahir, J. Ross, D. Morris, and D. Hess, "Os and compiler considerations in the design of the ia-64 architecture," in *Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*. ACM Press, 2000, pp. 212–221.
- [8] "SPEC CPU2000," 2000, <http://www.specbench.org/>.
- [9] "Perfmon," 2003, <http://www.hpl.hp.com/research/linux/perfmon/>.
- [10] A. Zhai, C. B. Colohan, J. G. Steffan, and T. C. Mowry, "Compiler optimization of scalar value communication between speculative threads," in *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*. ACM Press, 2002, pp. 171–183.
- [11] —, "Compiler optimization of memory-resident value communication between speculative threads," in *Proceedings of the international symposium on Code generation and optimization*. IEEE Computer Society, 2004, p. 39.
- [12] P. Marcuello, J. Tubella, and A. González, "Value prediction for speculative multithreaded architectures," in *Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture*. IEEE Computer Society, 1999, pp. 230–236.
- [13] J. González and A. González, "Speculative execution via address prediction and data prefetching," in *Proceedings of the 11th international conference on Supercomputing*. ACM Press, 1997, pp. 196–203.
- [14] D. I. August, D. A. Connors, S. A. Mahlke, J. W. Sias, K. M. Crozier, B.-C. Cheng, P. R. Eaton, Q. B. Olaniran, and W. mei W. Hwu, "Integrated predicated and speculative execution in the impact epic architecture," in *Proceedings of the 25th annual international symposium on Computer architecture*. IEEE Computer Society, 1998, pp. 227–237.
- [15] J. Lin, T. Chen, W.-C. Hsu, P.-C. Yew, R. D.-C. Ju, T.-F. Ngai, and S. Chan, "A compiler framework for speculative optimizations," *ACM Trans. Archit. Code Optim.*, vol. 1, no. 3, pp. 247–271, 2004.
- [16] J. Lin, T. Chen, W.-C. Hsu, and P.-C. Yew, "Speculative register promotion using advanced load address table (alat)," in *Proceedings of the international symposium on Code generation and optimization*. IEEE Computer Society, 2003, pp. 125–134.
- [17] J. Lin, T. Chen, W.-C. Hsu, P.-C. Yew, R. D.-C. Ju, T.-F. Ngai, and S. Chan, "A compiler framework for speculative analysis and optimizations," in *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*. ACM Press, 2003, pp. 289–299.
- [18] F. C. Chow, S. Chan, S.-M. Liu, R. Lo, and M. . Streich, "Effective representation of aliases and indirect memory operations in ssa form," in *Proceedings of the 6th International Conference on Compiler Construction*. Springer-Verlag, 1996, pp. 253–267.