The compiler as a validation and evaluation tool

Gerolf F. Hoflehner

Intel Compiler Lab
Santa Clara, CA
mailto:gerolf.f.hoflehner@intel.com

Daniel M. Lavery

Intel Compiler Lab
Santa Clara, CA
mailto:daniel.m.lavery@intel.com

David C. Sehr

Intel Compiler Lab
Santa Clara, CA
mailto:david.c.sehr@intel.com

Abstract

Like a processor executes flawlessly at different frequencies, a compiler should produce correct results at any optimization level. The Intel® Itanium® processor family with its new features, like the register stack engine and control- and data speculation, provides new and unique challenges for ported software and compiler technology. This paper describes validation and evaluation techniques that can be employed in compilation tools and can help to get a cleaner port of an application, a more robust compilation system and even insights into performance tuning opportunities. Using Itanium as a specific example, the paper explains why the register stack engine (RSE), the large register file, or control- and data speculation can potentially expose bugs in poorly written or compiled software. It then demonstrates validation and evaluation techniques to find or expose these bugs. An evaluation team can employ them to find, eliminate and evaluate software bugs. A compiler team can use them to make the compiler more stable and robust. A performance analysis team can use them to uncover performance opportunities in an application. We demonstrate our validation and evaluation techniques on code examples and provide run-time data to indicate the cost of some of our methods.

1 Introduction

For new implementations of an architecture, applications usually have to be recompiled for peak performance. An Independent Software Vendor (ISV) will more likely recompile an application, if it is cost efficient. It is cost efficient, if the system, the compiler and the source base are reliable and recompilation consistently gives measurable performance gains. This paper describes efficient techniques to evaluate the source code and to validate the compiler during the usual software application test cycle. For the compiler, we additionally propose more aggressive self-validation methods. Our methods can be a step towards reducing the debugging overhead in the software development cycle and thus improve the cost efficiency and practicability of recompilations.

The Itanium® architecture with features like the register stack engine and control- and data speculation, provides new challenges for ported software applications and system tools like the compiler. Any ported legacy software may contain bugs that can be exposed on a new architecture. This paper describes how compiler validation and evaluation techniques can face the challenges that the new architecture features and ported software can provide. We give an overview of potential issues and describe compiler techniques for mastering them. Self-validation techniques are most effective, if they can be employed. Otherwise, the compiler can provide user options and offer its validation and evaluation capabilities for the porting, evaluation, compiler and performance analysis teams. Some of the techniques discussed in this paper can be employed in other source code evaluation tools (e.g. lint on Unix systems) or in binary or assembly rewriting tools also. In this paper, evaluation is the process of testing that the source code is correct. Validation is the process that the compiler (or tool in general) is correct.

1.1 Contributions

The contributions of our paper are:

- we present a dynamic method to detect parameter mismatches in legacy code by utilizing the register stack (instance of a multiple alloc algorithm)
- we describe a compiler algorithm that discovers missing and redundant volatile declarations for variables at compile-time
- we describe a compiler self-validation technique to prevent generated code from NaT (=Not A Thing) consumption faults [10]
- we show how to stress test recovery code
- we evaluate the cost for the multiple alloc and recovery code stress testing techniques

The algorithm for finding missing and redundant volatile declarations is applicable for any architecture. The self-validation scheme can be extrapolated into a new software development strategy. The remaining methods are specific to the Itanium architecture, but can be employed in assembly and binary rewriting tools as well as in compilers.

1.2 Overview

The rest of the paper is structured as follows: Section 2 gives a brief overview of the register stack engine (RSE) and speculation on Itanium. Section 3 describes two common software glitches and how the compiler can detect them. Section 4 demonstrates that the compiler can employ self-checking or instrumentation techniques to catch NaT (=Not A Thing) consumption faults [9] triggered by un-initialized registers or automatic variables. Section 5 presents examples for speculation and recovery code and shows how the compiler can help evaluating correctness of the speculation code and the generated recovery code. Section 6 presents the conclusions.

1.3 Related Work

This paper fits into the software engineering and evaluation field [8] [12]. We feel that our approach is unique in the sense that it integrates self-testing strategies into the compiler, offers a compiler user interface for evaluation and porting teams, and gives specific evaluation and validation algorithms, including a measure for their cost.

2 The Register Stack Engine (RSE) and Speculation

The Itanium architecture has 128 architectural integer registers r0-r127. The upper 96 registers, r32-r127, are stacked. Each procedure can have its own variable size register stack frame of up to 96 registers. The stacked registers within a procedure are referenced as *architectural* registers. The hardware maps them to a micro architecture-dependent number of *physical* registers. For example, the first incoming parameter register in a procedure is referenced as r32. But this could be any physical register from r32 to the number of stacked registers implemented in the microarchitecture. Note that this paper uses the same nomenclature for both the architectural and the physical registers. With the *alloc* instruction [9], the code generator explicitly specifies its register stack frame (Figure 1): the number of incoming parameters (i), the number of outgoing parameters (o), the locally (within the procedure) needed registers (l) and the number of rotating registers (r) used in software-pipelined (swp) loops. The total number of stacked registers per procedure is i+l+o <= 96. The parameter registers overlap for the caller and the callee (Figure 1). The register stack frame is similar to the local memory stack frame, but is managed by the register stack engine (RSE), a processor state machine [9].

The Itanium® architecture supports control- and data speculation. This enables the compiler to move loads and the instructions that depend on them pasth the barriers imposed by branches and potentially aliasing store. For

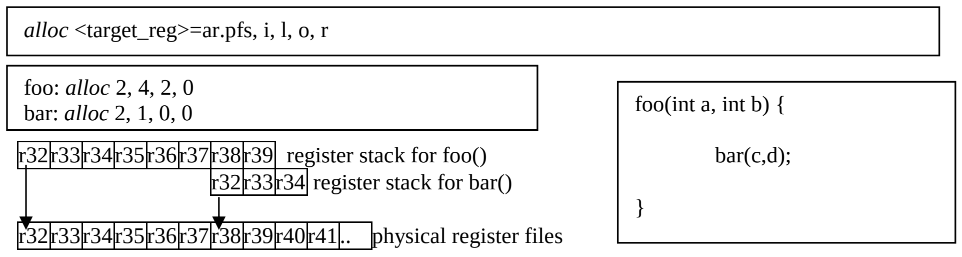

alloc <target_reg>=ar.pfs, i, l, o, r
foo: *alloc* 2, 4, 2, 0
bar: *alloc* 2, 1, 0, 0
r32 r33 r34 r35 r36 r37 r38 r39 register stack for foo()
r32 r33 r34 register stack for bar()
r32 r33 r34 r35 r36 r37 r38 r39 r40 r41 .. physical register files
foo(int a, int b) {
bar(c,d);
}

Fig. 1. alloc instruction and the register stack

control speculation, Itanium provides a speculative load (ld.s) and a validating speculation check (chk.s) instruction (*breaking the branch barrier*). A failed ld.s (e.g. due to a page fault) causes a deferred exception token (a NaT) to be set in the target register of the speculated load. For integer registers, the NaT (= Not a Thing) is encoded in an extra bit for the register. A chk.s instruction gets inserted at the place of the original, non-speculative load instruction. If a NaT bit is set for the operand register of the chk.s, execution branches to recovery code, which re-executes a non-speculative instance of the speculative load and all the dependent instructions, then branches back to the bundle after the chk.s [10] [4]. For data speculation, Itanium provides an advanced load (ld.a) and two advanced load check (chk.a, ld.c) instructions for data speculation. This enables the code generator to schedule a load across a potentially aliasing store (*breaking the store barrier*). At execution, an advanced load records information about its physical target register, memory address and data size in the Advanced Load Address Table (ALAT) [9]. If a subsequent store overlaps, then the hardware invalidates the corresponding ALAT entry to indicate the collision. As with control speculation, recovery code re-executes the non-speculated code and branches to the point after the advanced load check and resumes the regular program execution (Example 1).

```
(1)             ld8.a    V4=[V1]
(2)             add      V5=V4,V6
(3)             …
(4)             st4      [V10]=V11
(5)             chk.a    V4, rec
(6)     cont:   …
(7)
(8)     rec:    ld8      V4=[V1]
(9)             add      V5=V4,V6
(10)            br.cond  cont
```

Example 1: data speculation with recovery code

3 Source code evaluation techniques

We describe two problems in legacy code that could result in a run-time failure in an Itanium port: formal and actual parameter mismatch and missing volatile declarations. The compiler can instrument the generated code in evaluation mode to cause a run-time failure and catch the case of parameter mismatch. In section 3.3 we propose a missing volatile declaration detection algorithm at compile-time.

3.1 Formal and actual parameter mismatch

When the actual parameters in the caller don't match the formal parameters in the callee and there are less actual parameters than formal parameters, the result could be undefined program behavior or a NaT consumption fault. In Example 2 function foo() calls bar(int *a, int b) with only one parameter, bar(a). As bar() has two parameters, it can assume that foo() allocated the parameter registers in its register stack frame.

```
source code:
foo() {
      int *a;
      …
      bar(a);
}

bar (int *a,
int b) {
      *a=b;
}
```

```
pseudo assembly 1.

proc bar:

(1) st[r32]=r33;;
(2) add r33 =
(3) alloc
```

```
pseudo assembly 2.
proc bar:
(1)st [r32]=r33
(2)alloc r34=2,1,…


pseudo assembly 3.

(1)alloc r34=2,1,…
(2)st8 [r32]=r33
```

Example 2: parameter mismatch and 3 assembly code scenarios

Now, if bar accessed the parameter register r33 for b before it allocated the register stack, there are two conceivable run-time scenarios. In pseudo assembly 1 of Example 2, the result would be a register stack error at instruction (2) when bar() tries to write to r33, because the caller foo() did not allocate the parameter register on the register stack. In both, pseudo assembly 2 and pseudo assembly 3 of Example 2, the result could be a NaT consumption fault if the NaT bit for r33 is set. This can happen again because the caller foo() did not allocate and define the corresponding parameter register. The compiler can avoid generating the code in pseudo assembly 1 by blocking instructions that define a parameter register from being scheduled across an alloc instruction. However, there is nothing the compiler can do in the other cases. In case of pseudo assembly 2 or 3, there are three alternatives:

(i) the result will be a NaT consumption fault if the register r33 that bar() receives from the RSE has its NaT bit set

(ii) or there will be a memory corruption error (as an undefined value is stored to memory) that shows up at a different point and time in the program

(iii) or there will be a memory corruption error that never shows up.

Obviously, the ultimate remedy for the situation in Example 2 would be to declare prototypes and provide a warning free port. An alternative is for the compiler to instrument the application so the software bug gets detected at evaluation time. This gives the programmer the opportunity to add the missing parameter in the callee or clean up the dead parameter in the caller. In evaluation mode, the compiler inserts an extra alloc instruction that allocates the exact amount of outgoing registers (actual parameters) before the call. At the callee side it saves the last incoming (formal) parameter and restores it immediately before allocatin bar()'s register stack frame. If there is a parameter mismatch between the caller and the callee, this would result in a register stack error at the restore (instruction 3: in the assembly for bar in Example 3). The extra alloc on the callee side is necessary, as in the usual single alloc compilation model the alloc instruction in the function entry would allocate the maximum number of outgoing registers needed in any call. This is shown in the assembly for foo in Example 3: because foobar() has two parameters, foo would allocate two outgoing parameter registers. Before the call to bar(), foo shrinks the register stack to match the actual number of parameters (1 in the example). After the call to bar, foo restores the original register stack (Example 3).

```
foo() {
int *a;
int c;

      bar(a);

      foobar(a,c);
}
bar (int *a, int b)
{
      *a=b;

}
```

```
assembly for foo:

1:proc foo;
2:alloc r32=0,1,2,0
3:
4:alloc r10=0,1,1,0
5:mov r33=
6:br.call bar
7:alloc r10=0,1,2,0
8:
9:mov r33 =
10:mov r34 = …
11:br.call foobar
```

```
assembly for bar:

1:proc bar;
2:mov r20=r33
3:mov r33=r20
4:alloc r34=2,1,0,0
```

Example 3: using multiple allocs

This evaluation techniques is not only an option when the source code is available. It is also possible to apply the same technique in an assembly re-writing tool or in a binary rewriting tool. However, in these cases our evaluation strategy would require more effort. For example, it would be necessary to employ data-flow analysis for finding the exact number of parameters at each function call.

3.2 Cost of detecting parameter mismatches at run-time

To indicate the cost of this parameter mismatch evaluation method, we ran the SPEC CINT2000 benchmark suite on an Itanium® 2 system. The 12 C/C++ benchmarks are compiled with the Intel® Itanium Compiler [4] [11] on Microsoft® Windows Server 2003 with SPEC base options for optimal performance. A compiler switch controls the generation of the evaluation code. Figure 2 shows the percentage increase in execution time per benchmark for the binary with evaluation code compared to the binary without the extra code. All timings are for the complete reference input set. There are only four benchmarks, 197.parser, 252.eon, 254.gap and 256.bzip2, which run more than 1% slower with the evaluation code. None of the benchmarks slows down by more than 2%. There are three reasons for the slow down:

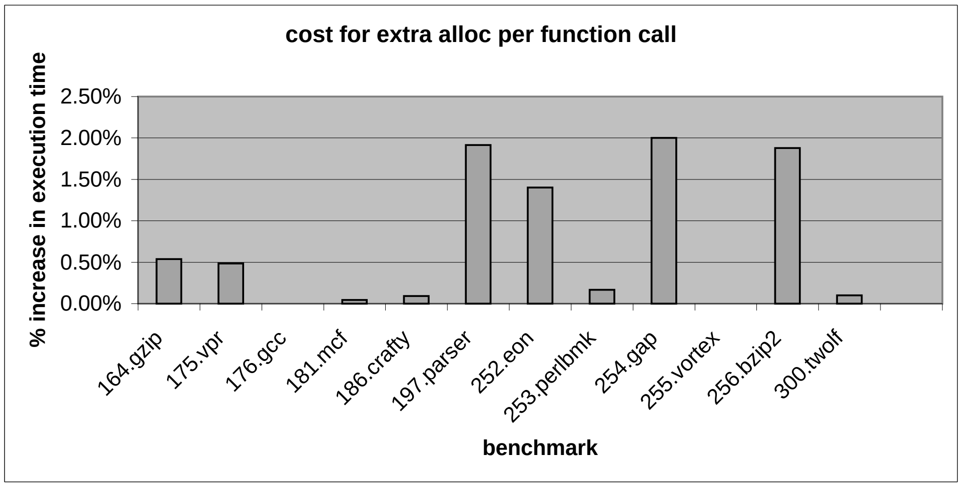


Fig. 2. run-time cost for extra alloc and moves for evaluation method

(i) there are extra alloc and move instructions executed

(ii) there are extra dependencies between the alloc and move instructions and the alloc instruction and instructions that access the outgoing registers

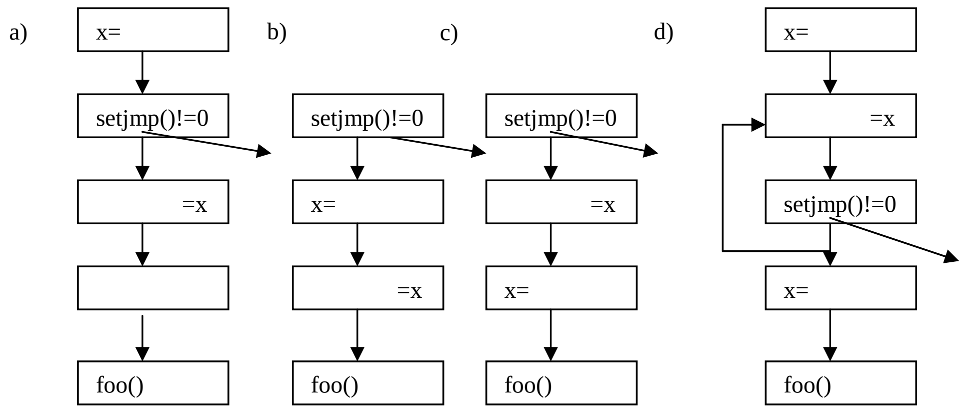
(iii) there is potentially an increase in code size because of the extra alloc and extra move instructions and because of the extra dependencies

There are no parameter mismatches for the SPEC CINT2000 benchmark suite, at least not on the paths executed by the ref input set.

3.3 Detecting missing volatile declarations in functions that call setjmp

On the Itanium architecture the compiler can allocate more data objects in registers. Thus the probability of exposing missing volatile declarations in the source code is higher than on other architectures. Missing volatile declarations are usually costly in terms of debugging time, and any method to detect them early should be helpful. According to the ANSI C standard a program has to declare an automatic variable as volatile if there is a use after a longjmp() of the variable that is defined on a path from a setjmp() call to any other call that could result in a longjmp() call [2].

In principle, the compiler can detect missing volatile declarations by employing variations of the classical reaching definition and liveness data-flow analysis algorithms [1]. Example 4 demonstrates some of the cases a volatile detection algorithm has to cope with. It shows snapshots a) – d) of control-flow graphs with all the definitions and uses of variable x in the graph. In a,b), the program does not need a volatile declaration for variable x: in case a), the definition is before the setjmp() call, in case b), there is no use on the path from the setjmp() call to the definition. In cases c) and d), however, variable x must be declared as volatile: in both cases the program intends to use the "last" value of x on the path after a return from a longjmp() call. The example assumes that foo() could trigger a call to the longjmp() function.



Example 4: defs and uses in the presence of setjmp()

We propose Algorithm 1 for detecting missing volatile declarations. It employs a variant of the reaching definition analysis and a classical liveness analysis on the cyclic control-flow graph. The first step in Algorithm 1 is a reaching definition analysis with three extensions:

(i) each call kills any variable definition reaching the call

(ii) a setjmp call is a definition

(iii) definitions of output parameters are ignored

The liveness analysis in the second step gives a list of automatic variables that are live at a setjmp() call. The third step associates a list of reachable call statements to each setjmp() call. The fourth step is a linear pass over each setjmp() call: it checks for each variable in the live list of the setjmp() call if there is a matching definition. In that case the variable may need to be declared as volatile and is collected in a "may volatile" list. The final step of the algorithm does a symbol table lookup for the members of the must volatile list and issues a report message when the volatile declaration may be missing in the source code.

Algorithm 1 is a context-insensitive algorithm that finds variables that may have to be declared as volatile in the source code. We note without proof that this is the best one can hope for: in general detecting variables that are to be declared as volatile is undecidable. The rationale for this is that a program could halt on a path from a setjmp() call to (one of) its corresponding longjmp() calls.

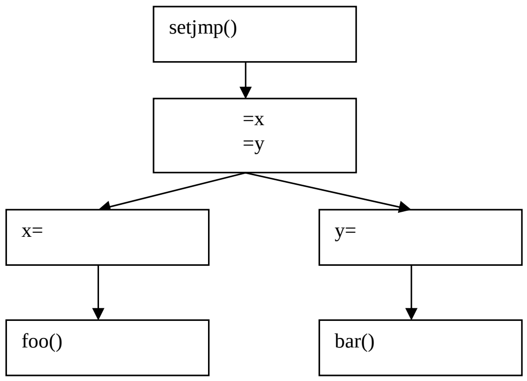
Algorithm 1 can also report variables that are declared as volatile, but actually don't have to be. In this case, the algorithm would report that a volatile declaration actually is redundant. Using this diagnosis will benefit program performance if variables that have been declared as volatile unnecessarily are used on the hot path of the function.

```
Step 1:  Input:   cyclic control-flow graph (cfg) of basic blocks
         Method: reaching definition analysis for all definitions.
                   -   setjmp() is a definition also
                   -   ignore output parameters
                   -   each call kills any  variable definition reaching the call (setjmp() defs
                       are not killed!)
         Output:  each call stmt has a set of definitions $DEF_{call}$ reaching the call
Step 2:  Input:   cfg as in step 1
         Method: perform liveness analysis for all automatic variables
         Output:  for each setjmp() call, a list of $LIVE_{setjmp}$ variables at the call
Step 3:  Input:   $DEF_{call}$ lists from Step 1
         Method: foreach call do
                   foreach d in $DEF_{call}$ do
                     if (d is a setjmp() call) then
                       LIST_ADD($CALL_{setjmp}$, call);
                     fi
                   od
                 od
         Output:  for each setjmp() a list of call stmt $CALL_{setjmp}$ reached by the setjmp() call
Step 4:  Input:   the $DEF_{call}$ , $LIVE_{setjmp}$  and $CALL_{setjmp}$ lists from steps 1, 2 and 3
         Method: List MAY_VOLATILE=[];
                 foreach setjmp() call do
                   foreach variable V in $LIVE_{setjmp}$ do
                     if(V is in $DEF_{call}$ for any call in $CALL_{setjmp}$)then
                       LIST_ADD(MAY_VOLATILE, V);
                     fi
                   od
                 od
         Output: list MAY_VOLATILE of automatic variables that must be declared as volatile
Step 5:  Input    MAY_VOLATILE list from step 4
         Method: foreach V in MAY_VOLATILE do
                   if (V is not declared as volatile) then
                     Report('V may need volatile modifier');
                   fi
                 od
         Output:  a list of  variables that may need to be declared as volatile for each function in the
                  compilation unit
```

Algorithm 1: Detecting missing volatile declarations

We note that Algorithm 1 can be extended in many ways to give more accurate results. For example, it can be made context sensitive (see Example 5 below). When the compiler can view the whole application (*whole-program analysis*), the intra-procedural algorithm can be extended to an inter-procedural algorithm: using a call graph analysis [14] it could determine all functions whose transitive closure in the call graph contains a call of the longjmp() function (*trigger calls*). Then the algorithm could consider only definitions that are on a path from a setjmp() call to a trigger call. Let's assume that in Example 5 foo() is a trigger call, but bar() is not. Using this information, Algorithm 1 applied to Example 5 would report that variable x has to be declared as volatile, while a volatile declaration for variable y is not necessary.

This inter-procedural algorithm could be used for detecting *dead* setjmp() calls also: when there is no path from the setjmp() to a trigger call, then the setjmp() is dead and can be removed. However, in the absence of whole-program analysis, the algorithm conservatively must consider any function call as a trigger call, unless the user provides some clues to the compiler about functions that are not trigger calls in the application.



Example 5: using trigger call information from the call graph analysis

We remark that an algorithm similar to Algorithm 1 can be used to eliminate redundant catch regions in C++ programs as well.

3.4 Section Summary

With its larger register file and its register stack engine the Itanium® architecture is more likely to expose bugs in poorly implemented software applications. Two examples for this are parameter mismatches and missing volatile declarations. In both cases, an Itanium compiler can help detecting these issues either at compile-time or at run-time. While the parameter mismatch issue in section 3.1 could be easily avoided by providing prototypes, many applications have a long source code history and the run-time evaluation method in section 3.1 could be a cheap alternative to static analysis methods. Section 3.3 presented a general, intra-procedural algorithm for detecting missing volatile declarations at compile-time. This algorithm can be extended to an inter-procedural algorithm and may be used to find redundant volatile declarations and even dead setjmp() calls also. Both methods can be made available to porting and evaluation teams by compiler user options, e.g. ecc –eval param_match, find_volatile_variables.

4 Preventing NaT consumption faults

In this section we describe validation algorithms that a compiler can employ to prevent NaT consumption faults in generated code. It is especially useful in compiler that aggressively try to minimize the usage of NaT bits (NaT consumption). The first section gives post-pass algorithms to detect possible NaT consumption faults, which could potentially be introduced by optimizing compilers. The second section gives a generic stress-testing technique to force NaT consumption faults at run-time during the software test cycle. We note that the RSE does not clear the NaT bits for the registers on the register stack frame. When the generated code stores a stacked register before it gets defined, a NaT consumption fault could occur at run-time [9].

4.1 A simple post-pass algorithm for preventing NaT consumption faults

For control speculated data and their uses, the NaT bit can be set. If the register allocator spills these data, it has to use the st8.spill/ld8.fill instructions, which consume one NaT bit in the ar.unat application register. For non-speculated data the register allocator can use the regular st8/ld8 instructions, which do not consume a NaT bit, for spilling and filling. But if non-speculated data are allocated into a register that is used before defined, a NaT consumption fault may still occur. The reason is that a general register has a NaT bit associated with it. If the register is not defined, then the NaT bit is not cleared. Specifically, the RSE does not guarantee that the NaT bits are clear for the registers on the register stack frame. Example 6 shows how a NaT consumption fault can be introduced in the compiler. We assume that the register allocator in an Itanium compiler implements a graph-based coloring scheme as described in ([6]). In Example 6 variable x is initialized in the first iteration at the bottom of the loop. When it gets spilled outside the top inner loop, then the program would execute the spill code before it executes the initialization of variable x at the bottom. When x does not get speculated, the register allocator may use the regular st8/ld8 instructions (instruction 3 and 8 in Example 6) for spilling. But then, if the variable x is assigned to a stacked register that has its NaT bit set, a NaT consumption fault results at program execution time.

```
while (outerloop) do

  while (innerloop) do
  …
  od
  …
  if (first_iteration) then
    x = …;
  else
    b = foo(x);
  fi
od
```

```
pseudo-assembly:
1:outerloop:
2:     add r10=96,sp
3:     st8[r10]=r60
4:     innerloop:
5:           …
6:           br innerloop
7:     add r10=96,sp
8:     ld8 r60=[r10]
9:     …
10:    first_iteration:
11:          mov r60=…
12:          …
13:    else:
14:          mov r80=r60
15:          br.call foo
```

Example 6: outer and inner loop with high register pressure

Obviously, the register allocator can conservatively always use the st8.spill/ld8.fill instructions and thus consume an extra NaT bit. Or it could use the NaT consumption prevention algorithm that we propose in Algorithm 2. If it detects a register that could trigger a NaT consumption fault, Algorithm 2 inserts initialization code into the function entry blocks. This clears the NaT bit and prevents a potential NaT consumption fault at run-time. In any case, the NaT consumption fault prevention algorithm can be used in the compiler as a safety net. It can warn the compiler developer about that issue during compiler testing and it can insert the initialization code to guarantee program correctness, should it detect a potential NaT consumption fault.

Algorithm 2 runs after register allocation and employs classical availability and liveness analysis for all registers. It uses this information to find registers that are used before they are defined in step 3 (with the notable exception of incoming parameter registers). A register is used before it is defined, if there exists any basic block in the control-flow graph at which the register is live_at_entry, but not available_at_entry. The algorithm adds these registers in a list (NAT_CAND in Algorithm 2) in one pass over the control-flow graph (step 3). Finally it inserts initialization code for the register in that list at each function entry and reports the issue in a compiler report.

```
Step 1: Input:   cyclic control-flow graph (cfg) of basic blocks
        Method:  perform availability analysis for all stacked registers
        Output:  list of register AVAIL_at_entry at each basic block
Step 2: Input:   cyclic control-flow graph (cfg) of basic blocks as in step 1
        Method:  perform liveness analysis for all stacked registers
        Output:  list of stacked register LIVE_at_entry for each basic block
Step 3: Input:   cfg with AVAIL_at_entry and LIVE_at_entry lists from step 1 and step 2
                 List NAT_CAND = [];
                 foreach basic block bb do
                       foreach register R in bb.LIVE_at_entry do
                             if (!register R in bb.AVAIL_at_entry) then
                                   LIST_ADD(NAT_CAND, V);
                             fi
                       od
                 od
        Output: list NAT_CAND of stacked registers with a potentially set NaT bit
Step 4: Input    NAT_CAND list from step 3
        Method:  foreach function entry block eb do
                       foreach R in NAT_CAND do
                             initialize R in eb; //insert mov R=0
                             REPORT('generated uninitalized register');
                       od
                 od
        Output:  validated generated code and a compile time report
```

Algorithm 2: NaT consumption fault prevention

4.2 Run-time method to test for un-initialized registers

The previous section discussed one specific instance when the compiler could introduce an un-initialized register that could cause a NaT consumption fault at run-time, and demonstrated how the compiler can safely catch and repair this situation. However, un-initialized registers could come also from un-initialized variables or bugs in aggressive compiler optimizations, like partial-redundancy elimination [13] [7] [14] and global code scheduling [5] [3] [4]. In any case, the un-initialized registers can be detected and reported by Algorithm 2. But the result of the algorithm may be conservative in the sense that it reports un-initialized registers although they are not. One example for this are rotating registers in a software-pipelined loop [15] [4] that are defined in a previous rotation. In this case, Algorithm 2 may report such a register as un-initialized although it is not. To find the actual un-initialized registers, Algorithm 2 can be complimented or substituted by Algorithm 3, which instruments compiled code in function entries so that every register used in the function has its NaT bit initially set. For NT applications and Linux kernels, the instrumentation code is a speculated load from address zero. For Linux applications, a speculated load from the kernel address space guarantees that the NaT bit for the target register is set.

This instrumentation method helps finding un-initialized registers by triggering a NaT consumption fault at run-time. This still may require some debugging of the root cause, but it is cheaper and well worth the effort to find bugs early in the software development cycle [12].

```
Step 1: Input:   cyclic control-flow graph (cfg) of basic blocks
        Method: - linear scan to find all registers used in the function.
                - delete parameter registers and registers defined in an entry block from the list
        Output:  list of register USED in the cfg

Step 2: Input    USED list from step 1
        Method: foreach function entry block eb do
                    foreach R in USED do
                        set NaT bit for R in eb;
                        ■  for Windows/ Unix kernels:
                           ld8.s R=[r0]
                        ■  eg. for Unix applications:
                           mov   rx=-1;;
                           ld8.s R=[rx] for Unix apps

                    od
                od
        Output:  instrumented code to catch un-initalized registers at run-time evaluation
```

Algorithm 3: NaT bit instrumentation of used registers in the entry blocks

4.3 Section Summary and Outlook

We presented validation algorithms that detect at compile- or run-time potentially un-initialized registers that could cause a NaT consumption fault at run-time. Since the results of our compile-time algorithm can be conservative, it can be complimented or substituted by an instrumentation method that forces a set NaT bit for each register used in a function. Both methods can be employed in assembly or binary-rewriting tools as well. All methods in this section can be extended to work for floating-point registers also.

5 The stress-testing of recovery code

In this section we quickly review control- and data speculation and give an example for recovery code. We discuss the types of errors the compiler could introduce in section 5.1 and propose testing and validation strategies in section 5.2. Section 5.3 presents run-time data for the validation costs.

5.1 Control- and data speculation

The Itanium® processor family provides a speculative load (ld.s) and a validating speculation check (chk.s) instruction for control speculation. For data speculation, the Itanium architecture provides an advanced load (ld.a) and two advanced load check (chk.a, ld.c) instructions for data speculation. This enables the code generator to schedule a load across a potentially aliasing store. In both cases, when the speculation was not successful, the check instruction jumps to the recovery code, which re-executes the non-speculated load and its dependent instructions [9] [16]. Example 7 shows recovery code for both, control- and data speculation. From the validation point of view, two problems can arise: a) the compiler generates wrong recovery code or b) the compiler does not generate the (load) check instruction at all. Both missteps can result in hard to debug non-deterministic program behavior.

```
 (a) original code:     (b) with control- and data speculation
                            and recovery code

      ld8   V2=[V1]                   ld8    V2=[V1]
                                      ld8.sa V7=[V6]    // control+data
      st8   [V3]=V4                   st8    [V3]=V4
                                      ld8.s  V8=[V7]    // control
      add   V5=V4,V3                  add    V5=V4,V3
                                      add    V9=V9,V8
      br.cond cont                    br.cond cont
      ld8   V7=[V6]                   chk.sa V7,rec1
      ld8   V8=[V7]             ret:  chk.s  V8,rec2
      add   V9=V9,V8            cont: ..
   cont:                        rec1: ld8    V7=[V6]
                                      ld8.s  V8=[V7]
                                      add    V9=V9,V8
                                      br.cond ret
                                rec2: ld8    V8=[V7]
                                      add    V9=V9,V8
                                      br.cond cont
```

Example 7: control and data speculation with recovery code

5.2 Testing and validation strategies for control- and data speculation

The key to stress testing recovery code is to force the branch to recovery code to be always taken and the load check to be always executed. This *force branch method* makes use of the fact that the advanced check for register r0 always misses in the Advanced Load Address Table (ALAT) [9]. Thus the advanced check for r0 always branches to its recovery code. For control speculation, the compiler (or a rewriting tool) can convert the control check into an advanced control check instruction. Then the chk.a can be converted to check r0, which forces the branch to the recovery code to be taken. The only caveat is that a chk.a can only be bundled in an M-syllable [14]. Thus, to convert a speculation check in an I-syllable, the compiler has to split the bundle with the chk.s in a new bundle and re-bundle the code locally (Figure 3). A load check instruction can be converted to a regular load instruction.

However, there could also be the case that the compiler erroneously did not generate the check instructions at all. In this case, the compiler could instrument the code to force a NaT consumption fault at run-time. One method to do this is to convert advanced loads to speculative loads from a *canary* address. A canary address guarantees that the speculative load will

```
force branch  method               canary address method
                                   ra: register which contains the canary address
conversions:                       converstions:
chk.s → chk.a r0                   ldx.a  rx=.. → ldx.sa rx=[ra]
chk.a → chk.a r0                   ldx.s  rx=.. → ldx.s   rx=[ra]
ldx.c → ldx

issue:                             issue:
chk.s in I-Unit                    does not work for post-increments
➔ re-bundle in M-Unit,             ➔ split    ldx.a rx=[ry],off into:
or split bundle and move                      ldx.a rx=[ra]
chk.a r0 into M-syllable                      add  ry=off, ry
```

Fig. 3. stress – testing methods for recovery code

miss and the NaT bit for the target register of the speculative load will be set. On Windows or for Unix kernels, address zero can serve as the canary address. For Unix applications, a kernel address would do the same job. In this case, the compiler could reserve a special address register and load the canary address into it at function entry. Then every speculative load can be instrumented so it loads from the canary address and will fail. One caveat of this method is, however, that it does not work for speculative post-increment loads. In this case, the compiler can split the speculative post-increment load into a regular load from a canary address and an address increment for the regular address.

The canary address method catches both, invalid recovery code and missing check instructions, for control- and data speculation. However, especially for Unix applications, it requires more implementation efforts than the force branch method.

5.3 Cost of validating testing control- and data speculation

To measure the run-time cost of the canary address method, we compiled the SPEC CINT2000 benchmark suite on an Itanium® 2 system running the Microsoft® Windows Server 2003. We used the base options for optimal performance and the reference input set. The compiler speculates more aggressively when feedback profiling information is available. Figure 4 shows the slowdown in execution time per benchmark for the binary that forces speculation to fail compared to the regular binary. A slowdown factor of 1 means that the executable runs 2 times faster in normal execution mode than in stress-testing mode. Thus a slowdown factor of 2 means that the executable runs 3 times faster in normal execution mode than in stress-testing mode. As our data show, for most benchmarks the execution time for a binary that stress-tests recovery code increases between 50% and 100%. The notable exceptions are 176.gcc and 300.twolf. Both run about 3 times slower when each speculation is forced to fail.

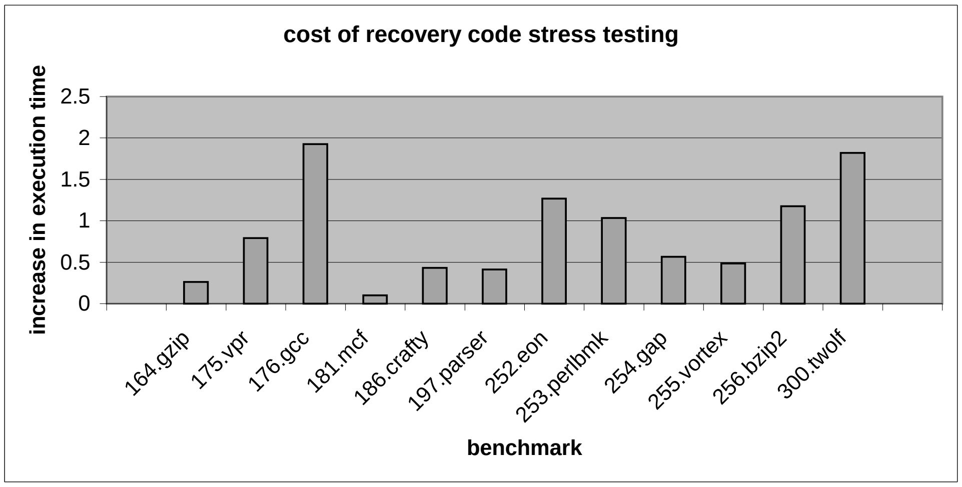


Fig. 4. increase in execution time when stress-testing recovery code

6 Conclusions

We designed and implemented various compiler validation and evaluation methods. Some of the methods are generally applicable, but some are specific to the Itanium architecture. In section 3 we discussed source code evaluation techniques to find parameter mismatches at run-time and missing volatile declarations at compile-time. In section 4 we demonstrated compiler self-validation techniques, which safeguard generated code from NaT consumption faults. In section 5 we presented the forced branch and canary address method for stress-testing recovery code.

The techniques presented in this paper can be made available as user options in the compiler.

The time overhead for the user driven evaluation methods is usually one run of the test system. The run-time cost for the parameter mismatch evaluation algorithm in section 3.1 is practically negligible. Detecting missing volatile declarations can be done at compile-time when compiling an applicaton. This requires no additional test cycle.

In a production compiler the compile-time spent in self-validation methods must be low. This can be achieved by using well-known and well-tested algorithms and by keeping the validation space small. This addresses also the question about "how to validate the validation algorithm?"

Stress-testing recovery code on average roughly doubles the total execution time for the 12 C/C++ benchmarks in the SPEC CINT2000 benchmark suite.

In future work, we will productize the prototype implementations described in this paper and collect more data on their usability and success rate. We also feel that self-validation can be built into the usual design/implementation/test cycle and can help improve software robustness and scalability. This can be done by a dual programming approach where validation is designed and implemented in the software development cycle. We argue that this would make software systems more robust, because algorithms invariants would be checked dynamically and consistently. It needs more data and experience to judge the practicability of this idea.

Also, the evaluation and validation algorithms enable the derivation of evaluation and validation metrics. For example, it would be interesting to gain insight into test coverage, like what percentage of the recovery code in an application actually was covered by the stress-testing method.

Finally, we are looking into optimizing validation and evaluation algorithms. For example, we plan to employ load safety techniques [3] to reduce the overhead caused by speculated code.

7 Acknowledgements

Thanks to the anonymous software vendors for using our compiler and making us aware of the methods discussed in the paper.

References

[1] A. Aho, J. Ullman,“Priciples of Compiler Design”, Addison-Wesley, 1986.

[2] ANSI Standard Programming Language C, Committee Draft ANSI, January 1999.

[3] D. Bernstein, M. Rodeh, M. Sagiv, “Proving safety of speculative load instructions at compile-time”, 4^{th} European Symposium on Programming, 1992.

[4] J. Bharadwaj, W. Y. Chen, W. Chuang, G. Hoflehner, K. Menezes, K. Muthukumar, J. Pierce, “The Intel IA-64 Compiler Code Generator”, IEEE MICRO, pp. 44-52, Sept/Oct 2000.

[5] R.J.Blainey, “Instruction scheduling in the TOBEY compiler”, IBM J. Res. Develop. Vol. 38 No 5, September 1994.

[6] G. Chaitin, “Register Allocation and Spilling via Graph Coloring”, Proc. of the SIGPLAN ’82 Symp. on Compiler Construction, Vol. 17, No. 6, June 1982.

[7] F. Chow, S. Chan, R. Kennedy, S-M. Liu, R. Lo, P. Tu,“A New Algorithm for Partial Redundancy Elimination based on SSA Form”, ACM SIGPLAN ’97 Conference on Programming Language Design and Implementation, June 1997.

[8] D. Gries, “The Science Of Programming”, Springer, 1978.

[9] Intel Corporation, “The Intel® Itanium® Software Developers’ Manual, Vol 1-3”, http://developer.intel.com/design/itanium/downloads/245317.htm, October 2002.

[10] Intel Corporation, “Intel® Itanium®2 Processor Reference Manual”, ftp://download.intel.com/design/Itanium2/manuals/25111001.pdf, June 2002.

[11] R. Krishnaiyer, D. Kulkarni, D. Lavery, W. Li, C. Lim, J. Ng and D. Sehr, "An Advanced Optimizer for the IA-64 Architecture", IEEE Micro, Vol20, No 6, Nov 2000, pp60-68.

[12] S. McConnel, "Code Complete", Microsoft Press, 1993.

[13] E. Morel, C. Renvoise, "Global Optimization by Suppression of Partial Redundancies", Comm. of the ACM, February 1979.

[14] S. Muchnick, Advanced Compiler Design and Implementation, Published by Morgan Kaufman, 1997.

[15] B.R.Rau, M. Lee, P.P.Tirumalai, M.S. Schlansker, "Register Allocation for Software Pipelined loops", ACM PLDI, 1992, pp 283-299.

[16] R. Zahir, J. Ross, D. Morris, D. Hess, "OS and Compiler Considerations in the Design of the IA-64 Architecture", ASPLOS-IX Proceedings on Architectural Support for Programming Languages and Operating Systems, November 2000.