ELEKTRONIK TIDNINGEN

Anders Holmberg
Produktchef för säkerhetskritiska tillämpningar
IAR Systems AB
Anders.Holmberg@iar.se
018-16 78 00

Så överlistar du en trasig kompilator

Ett fel upptäcks i kompilatorn i slutskedet av ett projekt. Istället för att revidera all kod kan du leta automatiskt efter kod som kan ha påverkats. Då slipper du omfattande kodrevideringar. Anders Holmberg visar hur.

Redaktör
Jan Tångring
jan@etn.se
0734-17 13 09

EMBEDDED EXPERT

EXPERTARTIKEL

Så överlistar du en trasig kompilator

Ett nytt sätt att hantera dilemmat ”uppdatera verktyget eller källkoden?”

Av Anders Holmberg, IAR Systems

Anders Holmberg är produktchef för säkerhetskritiska tillämpningar på IAR Systems där han jobbat i olika roller sedan år 2000. Tidigare var han konsult på Enator och TietoEnator. Under sin första fem yrkesår i början av 90-talet undervisade han i programmering på sin egen alma mater, Uppsala Universitet.
Anders Holmbergs kunnande omfattar även smartmobilutveckling och vetenskapliga beräkningar på parallelldatorer.

När man arbetar med kodgenereringsverktyg, som kompilatorer, händer det ibland att verktyget uppför sig på ett felaktigt sätt. Eller så kan det vara så att det upptäcks att produktionskoden förlitar sig på ej dokumenterat eller ej definierat beteende i kompilatorn. I detta fall finns det i praktiken bara två möjligheter för användaren: att uppdatera till en korrekt version av verktyget eller att ändra källkoden och på så sätt ta sig runt det oönskade beteendet.

Här följer ett möjligt tillvägagångssätt.

Att göra programändringar mycket sent i ett projekt är nästan aldrig särskilt bra. Trots att rättning av felet kan vara avgörande för korrekt och säker användning av slutprodukten ger det också ett antal oönskade sidoeffekter.

Sett utifrån hela utvecklingsprocessen så innebär att ändra kod och bygga om applikationen, att man tvingar fram omstart av en eller flera test- och QA-aktiviteter i projektet. Att minimera test- och QA-aktiviteter utan att äventyra säkerheten och integriteten vid kodändringar kan därmed vara avgörande för tiden till marknaden.

Ju senare i processen ett problem påträffas, desto fler steg i processen måste tas om för att säkerställa kvalitet och spårbarhet. I värsta fall kan en ny fullständig extern godkännandeprocess eller certifieringsbedömning krävas.

Det finns också ett kodperspektiv. Att ändra kod för att rätta till underligheter innebär alltid risk för att nya oönskade beteenden introduceras. Detta leder ibland till beslutet att låta problemet kvarstå i produkten och tydligt dokumentera problemets konsekvenser. Normerande säkerhetsramverk gör ofta det hela än svårare genom att de kräver omfattande analys av ändringarnas inverkan innan de utförs.

Det finns också ett goodwillperspektiv. Många eller stora kodändringar i ett projekts slutstadier kan göra beställarna osäkra på slutproduktens kvalitet. Om produkten redan har nått marknaden kan situationen bli rena rama mardrömmen.

Så, det är inte alltid möjligt att undvika kodändringar, men metoder och verktyg som förhindrar eller minimerar ändringars inverkan kan vara till stor hjälp.

Det finns tre huvudsakliga anledningar till sena kodändringar.

”Källkodsbuggen” beror antingen på misstag eller missförstånd från programmerarens sida när det gäller implementeringen, eller på en tvetydig eller ofullständig funktionsspecifikation som lämnar mycket öppet för tolkning. Även om detta ofta förekommer, kommer det inte att diskuteras vidare i artikeln.

”Den dolda non-ANSI C/C++-källkodsbuggen”. ANSI C-standarden har några mörka hörn där uppträdandet antingen är implementeringsdefinierat eller odefinierat. Om delar av källkoden har implementerats för att vara beroende av hur en viss kompilator uppträder för dessa hörnfall av standarden, kan man förvänta sig problem i framtiden. Men eftersom denna slags dolda bugg framför allt är en process- och kunskapsfråga kommer den inte att diskuteras vidare.

Detta leder oss till den här artikelns huvudfokus, nämligen buggen som beror på verktyget för objektkodsgenerering. Vi kommer att begränsa diskussionen till buggar i byggkedjan, det vill säga kompilatorn, assemblatorn och länkeditorn

Följande exempel kommer att användas i resten av artikeln: En bugg du har hittat beror på att kompilatorn gör felaktiga antaganden om registerallokering och stackallokering av variabler som är lokalt kopplade till en funktion. Buggen visar sig när många variabler konkurrerar om tillgängliga CPU-register och några variabler temporärt måste flyttas till stacken. Du har hittat buggen i en stor funktion med mängder med aritmetiska beräkningar, men det finns ingen garanti för att buggen endast visar sig i stora funktioner med många beräkningar.

Så vi hamnar vid frågan om vi ska övertala kompilatorleverantören att ta fram en fix eller använda oss av en eller flera ”workarounds” i hela kodmassan med alla de konsekvenser för projektet som vi angett ovan.

För högsäkerhetsprojekt bör kompileringskedjan och leverantören noggrant granskas innan val görs. Och det typiska scenariot är att när en viss kompilator och version har valts behålls den genom hela projektet. Vissa ramverk för säkerhetskritiska projekt kräver till och med att verktygsvalet underkastas en formaliserad process där verktygen

har förhandsgranskats eller är validerade enligt vissa speciella kriterier.

Vi ska nu titta på en speciell teknik som du kan använda om du har ett nära samarbete med din kompilatorleverantör. Vi tänker oss en 32-bitars CPU-arkitektur . Många 32-bitars CPU-kärnor inbegriper någon slags instruktionspipeline för att utöka prestanda genom att dela komplexa instruktioner i 1-cykeldelar som exekveras i sina egna pipelinestadier. På detta sätt kan ett genomflöde av en instruktion per klockcykel uppnås under idealiska förhållanden. Det är dock väldigt lätt att bryta detta flöde om på varandra följande instruktioner konkurrerar om samma resurs. Ett exempel är om den första instruktionen skriver till ett visst register och den direkt därpå följande instruktionen läser från samma register. I många arkitekturer med pipelines orsakar detta en så kallad pipeline stall, vilket innebär att encykelsbearbetningen avbryts medan den andra instruktionen väntar på att den första instruktionen slutför sin skrivning till registret.

En bra kompilator för en sådan CPU-arkitektur kommer att försöka arrangera om eller lägga upp instruktioner så att avståndet mellan instruktioner som använder samma CPU-resurs på ett pipelineblockerande sätt maximeras.

För att utföra detta måste kompilatorn ställa upp en eller flera beroendegrafer för den grupp instruktioner som ska ordnas för att avgöra om det är säkert att flytta en instruktion bakåt eller framåt i instruktionsströmmen. Kompilatorn använder en uppsättning funktioner för att avgöra om två instruktioner är oberoende, vilket betyder att de inte använder resurser på ett sätt som innebär konflikter och att deras ordningsföljd kan bytas.

Låt oss ta oss en titt på en funktion som kompilatorn skulle kunna använda för att avgöra om två MOV-instruktioner är oberoende.

```
Bool AreIndependent(Instr inst1, Instr inst2)
{
  if (IndependentSourceAndDest(Inst1, Inst2))
     return true;
  else
     return false;
}
```

Kodexempel 1

```
void ChangeMOVOrder(Instr inst1, Instr inst2)
{
  // Do other processing first
  ...
  if (AreIndependent(inst1, inst2)) {
     ChangeOrderHelper(inst1, inst2);
  }
}
```

Kodexempel 2

```
void ChangeMOVOrder(Instr inst1, Instr inst2)
{
  // Do other processing first
  ...
  // Bug detection code
  if (IsVolatile(inst1) || IsVolatile(inst2)) {
     ReportSourceStatement(inst1);
     ReportSourceStatement(inst2);
     return;
  }
  if (AreIndependent(inst1, inst2)) {
     ChangeOrderHelper(inst1, inst2);
  }
}
```

Kodexempel 3

Funktionen i kodexempel 1 ser oskyldig nog ut. Den skiftar i stort sett frågan om oberoende till en hjälpfunktion som avgör om källregister och målregister för MOV-instruktionerna används oberoende av varandra.

Det är helt OK för kompilatorn att lämna instruktionerna tillsammans i samma ordning. För att få ihop pusslet med maximala prestanda som resultat kan den till exempel ibland helt enkelt skapa en ny så kallad pipeline stall genom att flytta två instruktioner för att undvika ett annat avbrott.

Låt oss återvända till funktionen ovan och kompilatorn som använder denna funktion. När en kund kompilerar ett visst program med den här kompilatorn fungerar den felfritt, förutom att han noterar att två minnesskrivningar görs i fel ordning i förhållande till hur de specificeras i programmet. Vanligtvis är detta den princip som schemaläggaren är beroende av för att kunna utföra sina trick, men i detta fall är det inte OK eftersom användaren har angett att båda variablerna som påverkas av MOV-instruktionerna är volatila, vilket innebär att ordningen mellan skrivningarna inte får ändras. Om minnesskrivningarna exempelvis ska sätta igång någon extern periferi-enhet kan detta vara extremt viktigt.

Funktionen `AreIndependent()` ignorerar de båda instruktionernas volatila attribut och rapporterar därmed att det är OK att arrangera om dem.

Som angetts ovan kan schemaläggaren naturligtvis välja att lämna två oberoende instruktioner som de är, men för den här kunden är det lätt att se att han har åtminstone en plats som påverkas av den här buggen, men har han fler andra platser som uppvisar problemet? Att ta reda på detta på ett effektivt sätt innebär att all kod måste gås igenom på jakt efter läsningar och skrivningar till volatila variabler och att man måste undersöka den genererade koden. Så vi är tillbaka vid det centrala temat i den här artikeln: Hur kan kundens hantering av problemet bli effektivare?

Det finns en möjlig åtgärd för att avhjälpa situationen: en specialversion av den ursprungliga kompilatorn som kan försöka identifiera all kod i användarens samlade kodbas som faktiskt påverkas av buggen.

Här ska jag visa hur kompilatorn kan omvandlas till en så kallad ”buggdetektor”. Funktionen ovan används i funktionen i **kodexempel 2** som ändrar de två instruktionernas ordning när den funktionen har bestämt att det är förmånligt att göra så.

Denna funktion kan ändras så att den upptäcker buggfallet, det vill säga när `ChangeMOVOrder()`-funktionen använder fel information för att fatta ett beslut.

Den extra koden i rött i **kodexempel 3** letar efter situationen som ställer till problem, och när en sådan situation uppkommer rapporterar den det påverkade källäget. Lägg märke till hur koden i rött också rättar till buggen eftersom den klassificerar alla MOV-instruktioner med det volatila attributet som beroende. Men det är av största vikt att förstå att vi inte kunde ha placerat detektorkoden i den buggiga funktionen! Om vi hade gjort så skulle den ha rapporterat varje förekomst av möjliga felaktiga MOV-instruktioner.

Även detta förenklade exempel visade oss en av fallgroparna när man skapar en buggdetektor av produktionskvalitet. Det kan vara ganska enkelt att isolera upphovet till buggen, men väldigt komplicerat att avgöra när denna bugg faktiskt resulterar i generering av felaktig kod. Vi skulle till exempel kunna ha ett antal olika funktioner av varierande komplexitet som är beroende av `AreIndependent()`-funktionen.

Men om det är praktiskt möjligt att skapa buggdetektorn kan den nu användas för att noggrant ange exakt var kod som påverkas av den ursprungliga buggen finns. På detta sätt kan vi undvika att gå igenom all objektkod för hand för att leta efter möjliga förekomster av detta problem. ■