Analyse der Unterbrechungssperren in FreeBSD durch Symbolische Ausführung des LLVM Zwischencodes

Vortrag zur Masterarbeit
”Estimating Interrupt Latency Bounds Induced by Interrupt Locks in the FreeBSD Kernel for AMD64 by Symbolic Execution of LLVM IR”

7.8.2024

Jonathan Krebs

Agenda

- Kontext
- Präzisierung / Fokus der Aufgabe
- Aufbau der Analyse
- Anwendung
- Fazit

Kontext

Übergeordnetes Ziel
maximale Interrupt-Latenz des universellen Betriebssystems FreeBSD statisch bestimmen

Unterbrechungsverzögerung / Interrupt-Latenz
Zeit zwischen Unterbrechungsanforderung (IRQ) und Beginn der Behandlung durch das Betriebssystem

Interrupt Lock
IRQs werden vom Betriebssystem zur einseitigen Synchronisation verzögert, z.B. während Interrupt-Handlern und in kritischen Abschnitten.

Motivation in PAVE
Kann ein Power-Failure-Interrupt innerhalb des verbleibenden Energiefensters behandelt werden?

Fokus auf Kontrollfluss innerhalb kritischer Abschnitte

Ziel: C-Programm $\rightarrow$ Alle nativen Instruktionsfolgen zwischen cli und sti.

Problem
Interrupt-Zustand nicht klar aus Code-Struktur des Kerns ersichtlich:

```
static inline uint64_t intr_disable(void) {
        uint64_t rflags;
        asm volatile("pushfq; popq %0" : "=r" (rflags));
        asm volatile("cli" : : : "memory");
        return rflags;
}

static inline void intr_restore(uint64_t rflags) {
        asm volatile("pushq %0;  popfq" : : "r" (rflags));
}
```

```
void spinlock_enter(void) {
    if (curthread->spinlock_count == 0) {
        uint64_t flags = intr_disable();
        curthread->spinlock_count = 1;
        curthread->saved_flags = flags;
        critical_enter();
    } else
        curthread->spinlock_count++;
}

void spinlock_exit(void) {
    uint64_t flags = td->saved_flags;
    curthread->spinlock_count--;
    if (curthread->spinlock_count == 0) {
        critical_exit();
        intr_restore(flags);
    }
}
```

```
bool mtx_trylock_spin(atomic uint64_t *mutex) {
  spinlock_enter();
  bool acquired = CAS(mutex, MTX_UNOWNED, curthread);
  if(acquired) return true;
  spinlock_exit();
  return false;
}
```

Fokus auf Kontrollfluss innerhalb kritischer Abschnitte

Ziel: C-Programm $\rightarrow$ Alle nativen Instruktionsfolgen zwischen cli und sti.

Problem
Interrupt-Zustand nicht klar aus Code-Struktur des Kerns ersichtlich:

- `intr_disable` gibt den alten Zustand zurück, `intr_restore` stellt ihn wieder her.
- `spinlock_enter` / `spinlock_exit`: Weitere Verschachtelung durch Zähler
- teilweise dynamisches Interrupt-Sperr-Verhalten: z.B. `mtx_trylock_spin`

Annahme
Interrupt-Sperr-Verhalten einer Funktion kann durch Argumente, Rückgabewert und ausgewählte Variablen erschlossen werden.

Aufbau der Analyse

Bekannte Anfänge von Interruptsperren:
(Instruction, State)

- Start-Markierungen, z.B. `cli`-Instruktionen
- `call`-Instruktionen zu Funktionen, die Interruptsperren anfangen

$\longleftrightarrow$

Simulation auf LLVM-IR-Ebene:
Übersetzen zu Z3-Solver, um alle Variablenbelegungen gleichzeitig zu behandeln.

$\downarrow$

Trace als Graph: Knoten = LLVM Instruktionen

Zwischencode als Analysegrundlage

Compilation-Pipeline: $C \overset{\text{Frontend}}{\rightarrow}$ LLVM IR $\overset{\text{Optimierung}}{\rightarrow}$ LLVM IR $\overset{\text{Backend}}{\rightarrow}$ ASM

Vorteile gegenüber C oder ASM:

- Einfacher Aufbau / Kontrollfluss: Funktion $\ni$ Basisblock $\ni$ Instruktion
- Enthält Typen
- Enthält temporäre Werte statt Spilling
- C++-Infrastruktur zum Verarbeiten vorhanden

```c
#include <stdlib.h>
int add_something(int x, _Bool b) {
        int y;
        if(b) y = 4;
        else y = rand();
        return x + y;
}
```

```
add_something:
        pushq   %rbx
        movl    %edi, %ebx
        movl    $4, %eax
        testb   $1, %sil
        jne     .LBB0_2
        callq   rand@PLT
.LBB0_2:
        addl    %ebx, %eax
        popq    %rbx
        retq
```

```
declare i32 @rand()

define i32 @add_something(i32 %0, i1 %1) {
  br i1 %1, label %5, label %3
3:
  %4 = call i32 @rand()
  br label %5
5:
  %6 = phi i32 [ %4, %3 ], [ 4, %2 ]
  %7 = add i32 %6, %0
  ret i32 %7
}
```

Zwischencode als Analysegrundlage

Compilation-Pipeline: $C \overset{\text{Frontend}}{\rightarrow}$ LLVM IR $\overset{\text{Optimierung}}{\rightarrow}$ LLVM IR $\overset{\text{Backend}}{\rightarrow}$ ASM

Vorteile gegenüber C oder ASM:

- Einfacher Aufbau / Kontrollfluss: Funktion $\ni$ Basisblock $\ni$ Instruktion
- Enthält Typen
- Enthält temporäre Werte statt Spilling
- C++-Infrastruktur zum Verarbeiten vorhanden

Nachteile

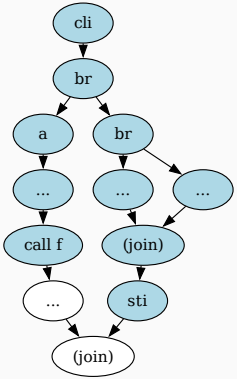
- (unpräzises) Abbilden auf Assembly notwendig
- Assembly-Quellcode wird nicht betrachtet

Symbolische Ausführung

- Idee: Emulation, bis Interrupt-Flag wieder 1 ist, Programmablauf mitschreiben.
- Programmzustand teilweise unbekannt. $\Rightarrow$ Symbolische Variablen
- Ergebnis von Instruktionen kann ein Term mit Variablen sein.
- Verzweigungen: Alle Möglichkeiten betrachten, Bedingungen mitführen.
- Z3-Solver als Bibliothek zum Erstellen und Lösen von symbolischen Ausdrücken
- hier Übersetzung selber implementiert
  - um zu wissen, wann Fehler in welche Richtung auftreten – wir brauchen alle möglichen Ausführungspfade
  - einfache Anbindung
  - Fertige Alternativen: z.B. KLEE, SymCC analysieren auch mit LLVM-IR, anderer Fokus.

Kontrollflussanalyse: DAG

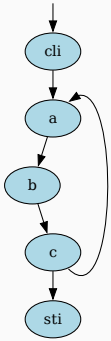
Einfachster Fall: Kontrollfluss im IR ist azyklisch
Gesucht: Subgraph, der mit Interrupt-Flag == 0 ausgeführt wird.



Beobachtungen

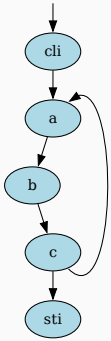
- `br` → „und“, `join` → „oder“
- Instruktionen können topologisch sortiert werden
- Funktionen können rekursiv bearbeitet werden.
  Annahme: keine tiefe/unbeschränkte Rekursion.
- wir wissen erst nach Auswerten einer Instruktion, ob/welchen ausgehenden Kanten gefolgt wird.

Kontrollflussanalyse: Zyklen



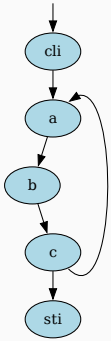
- topologische Sortierung nicht direkt möglich.
  - stattdessen Kondensierung auf starke Zusammenhangskomponenten.
  - Verallgemeinerung des Programmzustands am Einsprungpunkt eines Zykels:
    `phi`-Instruktionen bekommen undefinierte (symbolische) Werte
    Annahme: Interrupt-Sperr-Verhalten schleifeninvariant.
- Zyklen können verschachtelt sein. Zyklen werden rekursiv behandelt, Definition für Subzyklen wie in LLVM nach Havlak (1997).
- Nicht alle Zyklen im IR-Kontrollflussgraph bleiben erhalten:
  Wird die Rückkante genommen?

Kontrollflussanalyse: Zyklen



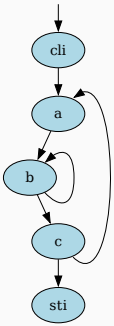
- topologische Sortierung nicht direkt möglich.
  - stattdessen Kondensierung auf starke Zusammenhangskomponenten.
  - Verallgemeinerung des Programmzustands am Einsprungpunkt eines Zykels:
    `phi`-Instruktionen bekommen undefinierte (symbolische) Werte
    Annahme: Interrupt-Sperr-Verhalten schleifeninvariant.
- Zyklen können verschachtelt sein. Zyklen werden rekursiv behandelt, Definition für Subzyklen wie in LLVM nach Havlak (1997).
- Nicht alle Zyklen im IR-Kontrollflussgraph bleiben erhalten:
  Wird die Rückkante genommen?

Kontrollflussanalyse: Zyklen



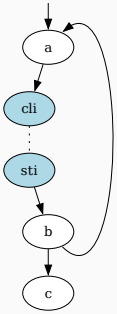
- topologische Sortierung nicht direkt möglich.
  - stattdessen Kondensierung auf starke Zusammenhangskomponenten.
  - Verallgemeinerung des Programmzustands am Einsprungpunkt eines Zykels:
    `phi`-Instruktionen bekommen undefinierte (symbolische) Werte
    Annahme: Interrupt-Sperr-Verhalten schleifeninvariant.
- Zyklen können verschachtelt sein. Zyklen werden rekursiv behandelt, Definition für Subzyklen wie in LLVM nach Havlak (1997).
- Nicht alle Zyklen im IR-Kontrollflussgraph bleiben erhalten:
  Wird die Rückkante genommen?

Kontrollflussanalyse: Zyklen



- topologische Sortierung nicht direkt möglich.
  - stattdessen Kondensierung auf starke Zusammenhangskomponenten.
  - Verallgemeinerung des Programmzustands am Einsprungpunkt eines Zykels:
    `phi`-Instruktionen bekommen undefinierte (symbolische) Werte
    Annahme: Interrupt-Sperr-Verhalten schleifeninvariant.
- Zyklen können verschachtelt sein. Zyklen werden rekursiv behandelt, Definition für Subzyklen wie in LLVM nach Havlak (1997).
- Nicht alle Zyklen im IR-Kontrollflussgraph bleiben erhalten:
  Wird die Rückkante genommen?

Kontrollflussanalyse: Zyklen



- topologische Sortierung nicht direkt möglich.
  - stattdessen Kondensierung auf starke Zusammenhangskomponenten.
  - Verallgemeinerung des Programmzustands am Einsprungpunkt eines Zykels:
    `phi`-Instruktionen bekommen undefinierte (symbolische) Werte
    Annahme: Interrupt-Sperr-Verhalten schleifeninvariant.
- Zyklen können verschachtelt sein. Zyklen werden rekursiv behandelt, Definition für Subzyklen wie in LLVM nach Havlak (1997).
- Nicht alle Zyklen im IR-Kontrollflussgraph bleiben erhalten:
  Wird die Rückkante genommen?

Startpunkte von Interrupt-Sperren

Rekursiv definierte Menge (Instruktion, Bedingung bzw. Zustand):

- Start-Markierungen
  - Interrupt-Flag war vorher 1.
- `call`-Instruktionen zu Funktionen f, die Interruptsperren anfangen.
  - Pfad zum Startpunkt existiert (approximiert)
  - $\wedge$ Pfad zum return existiert (durch Simulation)
  - $\wedge$ f fängt Interruptsperre an

Startpunkte von Interrupt-Sperren

Rekursiv definierte Menge (Instruktion, Bedingung bzw. Zustand):

- Start-Markierungen
  - Interrupt-Flag war vorher 1.
- `call`-Instruktionen zu Funktionen f, die Interruptsperren anfangen.
  - Pfad zum Startpunkt existiert (approximiert)
  - $\wedge$ Pfad zum return existiert (durch Simulation)
  - $\wedge$ f fängt Interruptsperre an

Fixpunkt-Iteration führt nach endlich vielen Iterationen zur Lösung (monotones boolesches Gleichungssystem in Instruktion $\times$ Programmzustand)

Praktikabilität der Fixpunktiteration

Anzahlen aller `call`-Instruktionen sowie aller Programmzustände sind groß.

Praktikabilität der Fixpunktiteration

Anzahlen aller `call`-Instruktionen sowie aller Programmzustände sind groß.

- Iteration nur innerhalb von Zyklen im Aufrufgraph nötig.

Praktikabilität der Fixpunktiteration

Anzahlen aller `call`-Instruktionen sowie aller Programmzustände sind groß.

- Iteration nur innerhalb von Zyklen im Aufrufgraph nötig.
- Problem:

```
void f(uint32_t x) {
        if (x == 0) { cli; }
        else { f(x-1); }
}
```

- Beobachtung: *n*. Iteration findet Sperren auf durch *n* Funktionsaufrufe.

Praktikabilität der Fixpunktiteration

Anzahlen aller `call`-Instruktionen sowie aller Programmzustände sind groß.

- Iteration nur innerhalb von Zyklen im Aufrufgraph nötig.
- Problem:

```
void f(uint32_t x) {
        if (x == 0) { cli; }
        else { f(x-1); }
}
```

- Beobachtung: *n*. Iteration findet Sperren auf durch *n* Funktionsaufrufe.
- *Annahme:* Sperr-Graph (Subgraph vom Aufrufgraph) nicht zyklisch, d.h. Sperren nur auf äußerer Rekursionsebene
- Dadurch ist die Iterationszahl durch die Pfadlänge begrenzt.
- In analysiertem Code: 2 Zyklen im Aufrufgraph, max. 4 Iterationen.

Anwendung: Anpassen von Annahmen und Realität

Interrupt-Sperr-Verhalten $\leftrightarrow$ Argumente und Rückgabewert

- $\lightning$ Spinlock-Counter
- $\Rightarrow$ nötige Variablen gehören zum Interrupt-Sperr-Verhalten

Interrupt-Sperr-Verhalten ist schleifeninvariant

- $\lightning$ `kern/subr_turnstile.c`: `propagate_priority`
- $\Rightarrow$ Code angepasst

keine indirekten Funktionsaufrufe:

- $\lightning$ IRQ-Handler
- $\lightning$ KTRACE (deaktiviert)
- ! Annahme: beeinflusst nicht das Interrupt-Sperr-Verhalten

Kontrollfluss nicht unterbrochen

- ✓ Exceptions kommen nicht vor
- ! Kontextwechsel: Markieren. Nur 1 pro Sperre gefunden.

Rekursion sehr einfach beschränkt

- $\lightning$ Work-Stealing bei SMP ($\Rightarrow$ deaktiviert)

Funktion	# Instruktionen	# Schleifen	# Kontextwechsel
(critical_exit_preempt)	811	5	1
cpu_new_callout	1321	6	1
callout_reset_sbt_on	1624	6	1
_callout_stop_safe	3244	15	1
cnputsn	1236	7	1
msgbuf_addstr	1660	11	1
__mtx_lock_sleep	2954	17	1
__mtx_unlock_sleep	3020	14	1
_sleep	6923	33	1
wakeup	2502	10	1

Maximale Anzahl an Instruktionen, Schleifen und Kontextwechseln in einem ununterbrechbaren Abschnitt, Gruppiert nach enthaltender Funktion.
Schleifenrücksprünge wurden ignoriert, d.h. Schleifen werden mit 0 oder 1 Iteration gezählt.
Kernel wurde ohne SMP-Unterstützung kompiliert.

Fazit

- Werkzeug, um ununterbrechbaren Code zu finden und den möglichen Kontrollfluss auszuwerten
- Notwendige Verbesserungen an der Analyse:
  - Schleifenobergrenzen
  - automatisch Annahmen prüfen / Fehler suchen
  - Funktionszeiger behandeln, insbesondere für ISRs.
- Interrupt-Sperren in FreeBSD sind komplexer als erwartet
- Anpassung notwendig: Unbeschränkte Schleifen in kritischen Abschnitten vermeiden (z.B. bei mehreren Spinlocks, Prioritätsvererbung)

Demo