

EXAMENSARBETE A Tracing JIT Compiler for Erlang using LLVM

STUDENT Johan Fänge

HANDLEDARE Jörn Janneck (LTH), Haitao Li (Ericsson, Shanghai R&D)

EXAMINATOR Görel Hedin (LTH)

En tracing JIT-kompilator för Erlang

POPULÄRVETENSKAPLIG SAMMANFATTNING Johan Fänge

Nästan alla moderna programspråk använder en interpretator – en flexibel och praktisk om än långsam lösning. Vi prövar ett enkelt sätt att kraftigt öka prestandan på Erlangs interpretator.

Det är vanligt att programspråk använder en interpretator för att köra ett program – exekvera programkod – istället för att kompilera direkt till maskinkod. Interpretatorn är ett litet program som läser programkoden i något bekvämt format, t.ex. som ren text, och sen utför det som står där direkt. Det finns flera fördelar med en interpretator:

- Lätt att skriva och underhålla.
- Lätt att göra dynamisk och flexibel.
- Behövs ingen komplicerad kompilering, kodgenerering eller länkning.
- Samma kod kan fungera på flera plattformar och processorer.

En interpretator är dock ofta långsam, och ett sätt att öka prestandan utan att förlora fördelar som plattformsoberoende är att kompilera till bytekod. Detta är enkla instruktioner, inte helt olik en processors assemblerkod. Detta används av exempelvis Java, C#, Python och även Erlang. Det är lite som att baka klart en fryst pizza i ugnen – snabbt och smidigt och resultatet blir hyfsat, men att göra det från grunden i en riktig pizzaugn ger bättre kvalitet.

Programspråk som vanligtvis använder en interpretator utnyttjar ofta de möjligheter till flexibilitet det ger, vilket kan göra det nästintill omöjligt att kompilera ett sådant program direkt till maskinkod för att öka prestandan. Typexemplet på en besvärlig funktion är `eval`, som kan exekvera godtycklig programkod utifrån en textsträng.

En annan lösning för att nå liknande prestanda är då en Just-In-Time(JIT)-

kompilator. Istället för att kompilera hela programmet till maskinkod i förväg, så väntar man med detta steg tills programmet exekverar och man t.ex. vet vilken plattform programmet kör på, eller vilken data det anropats med. Nu uppstår dock ett annat problem: det tar tid att kompilera och optimera kod. Typiskt börjar man därför i interpretatorn, och kompilerar först till maskinkod när en kodsnuett körs tillräckligt ofta för att det ska löna sig.

Ett lätt sätt att utöka en bytekodsinterpretator är genom en tracing JIT-kompilator. Grundprincipen är att man kör koden som vanligt i interpretatorn tills man stöter på en het loop, en kodsnuett som exekveras många gånger på rad, varpå man fortsätter att använda interpretatorn, men samtidigt också skriver ned vilka instruktioner man exekverar tills man kommer tillbaka till början av loopen. Detta bygger på insikten att den statistiskt vanligaste kodvägen typiskt också är den som med störst sannolikhet observeras.

Resultatet blir en trace, dvs en rak loopiteration som kan optimeras lätt och effektivt. Där programflödet kan avvika från den nedskrivna läggs speciella guard-instruktioner in som återgår till interpretatorn om dess villkor inte är uppfyllt.

I examensrapporten beskrivs våra erfarenheter av att ha utvecklat en tracing JIT-kompilator för programspråket Erlang, ett språk som används mycket på Ericsson där exjobbet påbörjades. I en utvärdering av prestandan på ett antal mindre benchmarks lyckades vi i vissa fall fyrdubbla exekveringshastigheten.

```
# original
names_by_id = [(23, "Kalle"),
               (1, "Eric"), (7, "Bo"), ...]

def find_name(wanted_id):
    for (id, name) in names_by_id:
        if id == wanted_id:
            return name
    return None

# trace
:trace_start
x = names_by_id.get_element(i)
if failed:
    goto :trace_abort
id = get_fst_tuple(x)
if id == wanted_id:
    goto :trace_abort
i = i + 1
goto :trace_start

:trace_abort
...
```

Exempel på en (påhittad) trace för Python