Abstrakte Maschinen
(Virtuelle Maschinen)

M. Anton Ertl

Was ist eine virtuelle Maschine?

Programmiermodell einer realen Maschine

Aus „Pentium Processor User's Manual, Vol. 3“, Seite 3-1:

- Memory Organization
- Data Types
- Registers
- Instruction format
- Operand selection
- Interrupts and exceptions

entworfen für Implementierung in Hardware
Befehle im RAM für Ausführung
Befehle in ausführbaren Dateien für Speicherung und Übertragung
Befehle in Objektdateien und Libraries zum Linken

Was ist eine virtuelle Maschine? Beispiel: Java VM

- Speicher: Menge von Arrays und Objekten, mit Garbage Collection
- Stack, Locals
- Datentypen
  Stack:
  1 Slot: `int` (i32), `float` (f32), `address`
  2 Slots: `long` (i64), `double` (f64)
  Nur im Speicher: `byte` (i8), `short` (i16), `char` (u16).
  statisches Typechecking (nicht Java Typechecking)
- Befehle: Bytes (bytecode), Immediate-Operanden als Folgebytes
- Exceptions

Zweck von virtuellen Maschinen

- Modulare Programmiersprachenimplementierung
- Schnelle Interpretation (WAM, Java VM anfangs)
  im Gegensatz zu anderen Zwischendarstellungen
  oder Übersetzung in realen Maschinencode (Webassembly)
  Ist LLVM eine VM? Textdarstellung, keine Übersetzung zur Laufzeit
- Übertragungsformat (Java VM, Webassembly, Smalltalk/Squeak)
  Image (Smalltalk) entspricht ausführbarer Datei
  oder Pakete (Java .class) entsprechen DLLs
- oder nur zur Laufzeit (CPython)

Ist eine Virtuelle Maschine nicht sowas wie VirtualBox?

- Im Betriebssystembereich
  Simuliert reale Maschine
  Meist der selbe Befehlssatz
  Betriebssystem für reale Maschine kann darauf laufen
  Hypervisor
  nicht Thema dieser LVA
- Im Programmiersprachenbereich (auch: Abstrakte Maschine, Bytecode)
  Zwischenschicht für die Implementierung
  gewisse Ähnlichkeiten zu realer Maschine
  optimiert für Implementierung in Software
  oder als Zwischenstufe zur Übersetzung
  soll auf verschiedener Hardware effizient sein
  Thema dieser LVA
- Überlappungen
  Sandboxing
  Programmiersprachen-VM als Basis für Betriebssystem (UCSD p-System)

Java VM

- Befehle im Speicher: Interpretation, JIT Compilation
- Befehle in `.class`-Dateien: für dynamisches Linken
- Dauerhaft stabiles Format
  viele Compiler erzeugen es
  viele Implementierungen
  viele Werkzeuge
  maschinenunabhängig
- Unterschiedliche Anforderungen:
  Linken: Referenzen über constant pool
  Interpretation: fixe offsets
  Quickening:
  ersetze z.B. `getfield` mit Referenz auf constant pool
  durch `getfield_quick` mit offset

```
class Example {
 static int square(int num) {
     return num * num;
 }

 static int sumsq(int a, int b) {
     int c=square(b);
     return square(a)+c;
 }
}
```

```
javac Example.java
javap -v Example.class
```

```
square:
   0: iload_0
   1: iload_0
   2: imul
   3: ireturn

sumsq:
   0: iload_1
   1: invokestatic #7
   4: istore_2
   5: iload_0
   6: invokestatic #7
   9: iload_2
  10: iadd
  11: ireturn
```

```
  #1 = Methodref #2.#3
  #2 = Class #4
  #3 = NameAndType #5:#6
  #4 = Utf8 java/lang/Object
  #5 = Utf8 <init>
  #6 = Utf8 ()V
  #7 = Methodref #8.#9
  #8 = Class #10
  #9 = NameAndType #11:#12
 #10 = Utf8 Example
 #11 = Utf8 square
 #12 = Utf8 (I)I
 #13 = Utf8 Code
 #14 = Utf8 LineNumberTable
 #15 = Utf8 sumsq
 #16 = Utf8 (II)I
 #17 = Utf8 SourceFile
 #18 = Utf8 Example.java
```

```
class Example2 {
    float x;
    long a[];
    void inca(long inc) {
        for (int i=0; i<a.length; i++) {
            a[i] += inc;
        }
    }
    void inc1() {
        inca(1);
    }
}
```

```
 0: iconst_0
 1: istore_3
 2: iload_3
 3: aload_0
 4: getfield    #7
 7: arraylength
 8: if_icmpge   27
11: aload_0
12: getfield    #7
15: iload_3
16: dup2
17: laload
18: lload_1
19: ladd
20: lastore
21: iinc        3, 1
24: goto        2
27: return
```

```
 0: aload_0
 1: lconst_1
 2: invokevirtual #13
 5: return
```

Befehlsübersicht Java VM, Teil 1

- ca. 200 Befehle, codiert als bytes mit immediate-Operanden (big-endian)
- Push Constants: z.B. `bipush`, `ldc1`, `ldc2w`
- Load local, z.B. `iload`, `iload_3`, `aload`
  `wide` prefix-Befehl für locals ab 256
- Store local, z.B. `lstore`, `dstore_3`
- Arrays, z.B. `anewarray`, `caload`, `lastore`
- Fields, z.B. `getfield`, `putstatic`
- Objekte: `new`, `checkcast`, `instanceof`
- Stack-Befehle, z.B. `pop`, `dup`, `dup2_x2`

Befehlsübersicht Java VM, Teil 2

- Arithmetik, z.B. `iadd`, `fmul`, `lrem`
- Bitweise, z.B. `ishr`, `lushr`, `ixor`
- Umwandlung, z.B. `i2l`, `d2f`, `int2char`
- Control transfer, z.B. `ifeq`, `if_icmpeq`, `lcmp`, `goto`
- switch, z.B. `tableswitch`, `lookupswitch`
- Exceptions: `athrow` (try-Block-Information out-of-band)
- Method invocation, z.B. `invokevirtual`, `invokeinterface`
- Return, z.B. `ireturn`
- Monitors: `monitorenter`, `monitorexit`
- Quick-Varianten: z.B. `getfield_quick`, `putfield2_quick`

Beispiel: Gforth

- Speicher: mit Adressarithmetik, keine Abstraktion
- Datenstack, Return-Stack, FP stack, locals stack
- Auf dem Daten- und Returnstack: Maschinenwörter („Zellen“)

Beispiel: Gforth ohne locals

```
: square ( num -- n )
  dup * ;

: sumsq ( a b -- n )
  square swap square + ;
```

```
<square>      dup
<square+$8>   *
<square+$10>  ;s

<sumsq>       call
<sumsq+$8>    square
<sumsq+$10>   swap
<sumsq+$18>   call
<sumsq+$20>   square
<sumsq+$28>   +
<sumsq+$30>   ;s
```

Beispiel: Gforth mit locals

```
: square {: num -- n :}
    num num * ;

: sumsq {: a b -- n :}
    b square {: c :}
    a square c + ;
```

```
<square>      >l
<square+$8>   @local0
<square+$10>  @local0
<square+$18>  *
<square+$20>  lp+
<square+$28>  ;s
```

```
<sumsq>       >l
<sumsq+$8>    >l
<sumsq+$10>   @local1
<sumsq+$18>   call
<sumsq+$20>   square
<sumsq+$28>   >l
<sumsq+$30>   @local1
<sumsq+$38>   call
<sumsq+$40>   square
<sumsq+$48>   @local0
<sumsq+$50>   +
<sumsq+$58>   lit
<sumsq+$60>   #24
<sumsq+$68>   lp+!
<sumsq+$70>   ;s
```

Gforth: Zweck der VM

- Quellcode ist dauerhaftes Format
- VM ursprünglich für Interpretation
  VM-Code eigentlich nur flüchtig im RAM
- aber: Gforth's Compiler ist in Gforth geschrieben
  VM in GNU C, Rest in Forth
- ⇒ Image für Bootstrapping
  Image (Speicherdump): viel einfacher als `.class` files
- Gforth Images:
  abhängig von der Gforth-Version
  abhängig von byte order und Zellengröße
  unabhängig von Basisadresse (wegen ASLR): Adressen vs. andere Daten
  unabhängig von Engine (`gforth`, `gforth-fast`, ...): Zahlen für VM-Befehle

Externe Formate: Von Image bis `.class`-Dateien

- Image: Abbild eines Teils des Speichers
  Kann einfach in den Speicher geladen werden (an dieselbe Adresse)
- ähnlich ausführbaren Dateien (besonders im `a.out`-Format)
- Adressunabhängige Images
  Basisadresse zur Laufzeit addieren (Win32Forth, entspricht PIC)
  Information, was Addresse ist, Basisadresse beim Laden addieren (Gforth)
  Zwei Images an verschiedenen Adressen erzeugen und vergleichen
- `.class`-Dateien entsprechen `.o`, `.so`, `.dll`
- Linken mehrerer vom Compiler erzeugten VM-Dateien
  symbolische Referenzen auf Dinge in anderen Dateien
- mehr symbolisch ⇒ mehr Auflösung zur Laufzeit
  `.o` `.so`: offsets von Feldern sind Zahlen
  `.class`: offsets von Feldern sind symbolisch

Implementierung

Ziele

- schnelle Laufzeit
- schnelle Compilation
- geringer Speicherverbrauch
- Portabilität
- Implementationsaufwand
- Debugging

Wege

- Interpreter
  `switch`-basiert
  Threaded code
- Code-copying Hybrid
- Compiler
  Copy-and-patch compiler
  handgeschrieben
- mehrstufiger Compiler
  Granularität: Methode oder Trace

VM Interpreter

- Dispatch
  hole nächsten VM-Befehl und starte ihn
- Operandenzugriff
  hole Operanden, speichere Resultate
- Nutzlast
  der für die Semantik der implementierten Sprache relevante Teil

switch-basierter Interpreter

```
void vm(long *ip, long *sp)
{
  for (;;) {
    switch (*ip++) {
    case ...:
      ...
    case plus: /* + */
      sp[1] += sp[0];
      sp++;
      break;
    case ...:
      ...
    }
  }
}
```

```
    # %rdx: Tabellenbasis
    # %rdi: ip; %rsi: sp
10: add    $8,%rdi
14: cmpq   $0xd,-8(%rdi)
19: ja     10
1b: mov    -8(%rdi),%rax
1f: movslq (%rdx,%rax,4),%rax
23: add    %rdx,%rax
26: jmp    *%rax

70: mov    (%rsi),%rax
73: add    %rax,8(%rsi)
77: add    $8,%rsi
7b: jmp    10
```

- Standard C
  maximale Portabilität
- VM-Register in
  realen Registern

"Call threading"

```
typedef void (* inst_t)();
inst_t *ip;
long *sp;
void vm()
{
  for (;;) {
    inst_t inst=*ip++;
    inst();
  }
}
void add()
{
      sp[1] += sp[0];
      sp++;
}
```

```
 0: sub   $8,%rsp
 4: mov   ip(%rip),%rdx
 b: lea   8(%rdx),%rax
 f: mov   %rax,ip(%rip)
16: mov   $0,%eax
1b: callq *(%rdx)
1d: jmp   4

1f: mov   sp(%rip),%rax
26: mov   (%rax),%rdx
29: add   %rdx,8(%rax)
2d: add   $8,%rax
31: mov   %rax,sp(%rip)
38: retq
```

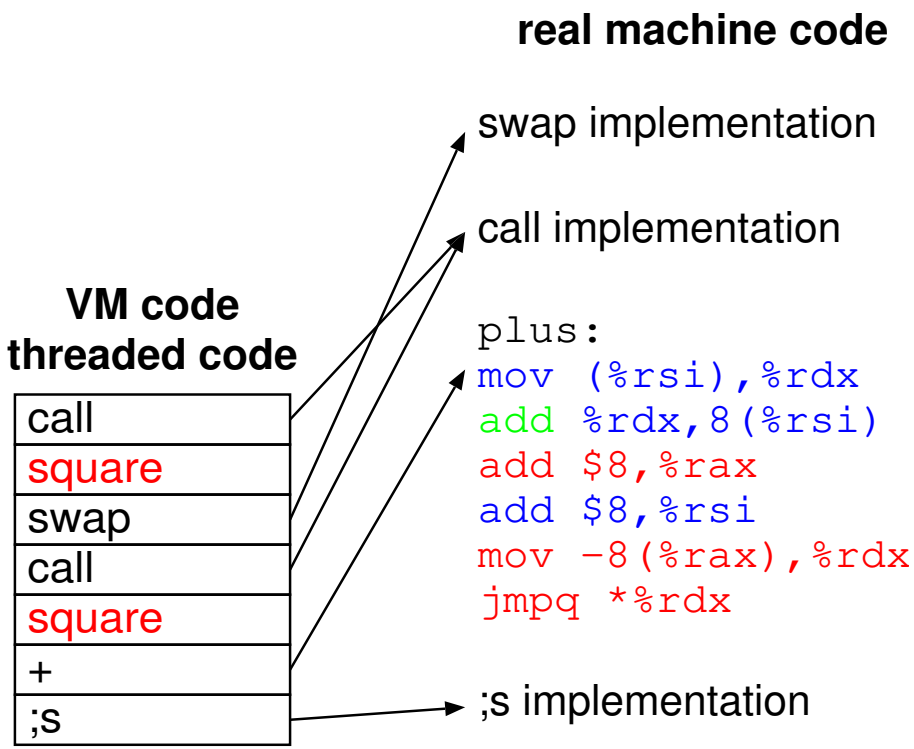
- Standard C
  maximale Portabilität
- Modularität
- VM-Register im Speicher
- viel Kontrollfluß

Direct-threaded code (Labels as values)

```
typedef void *inst_t;
inst_t *vm(inst_t *ip,
           long *sp)
{
  static inst_t insts[] =
      {... &&plus, ...};
 ...
 plus: /* + */
  sp[1] += sp[0];
  sp++;
  inst_t next = *ip;
  ip++;
  goto *next;
 ...
}
```

```
mov  (%rsi),%rdx
add  %rdx,8(%rsi)
add  $8,%rax
add  $8,%rsi
mov  -8(%rax),%rdx
jmpq *%rdx
```

- nicht-standard C
  gcc, clang, icc, tcc
  gcc seit 1992
- VM-Register in
  realen Registern



Direct-threaded code (tail-call optimization)

```
typedef void (*inst_t)(void **ip,
                       long *sp)
 __attribute__((regcall));

__attribute__((regcall))
void plus(void **ip, long *sp)
{
  sp[1] += sp[0];
  sp++;
  inst_t next = *ip;
  ip++;
  __attribute__((musttail))
  return next(ip,sp);
}
```

```
# clang-14 -O2
mov (%rsi),%rax
add %rax,8(%rsi)
add $8,%rsi
mov (%rdi),%rax
add $8,%rdi
jmp *%rax
```

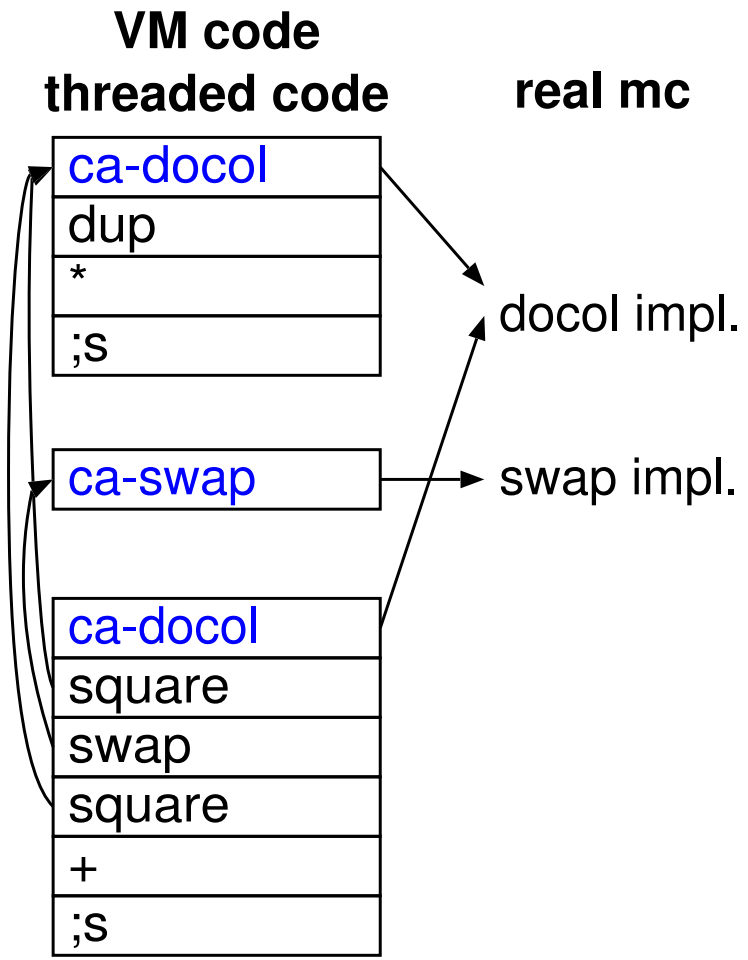
- tail-call Optimierung
  gcc-9 und clang-11 führen sie durch
  mit `musttail` garantiert
  `musttail` ab clang-13 und gcc-15
- Register nach Calling Convention
  AMD64 System V ABI: 6 GPRs
  gcc: explizite Registervariablen
  clang-11:
    Calling Convention `regcall`
    11 Parameter in Registern
    nur AMD64
  ARM A64, RISC-V: 8 GPRs

Speichersparen mit indirect threaded code

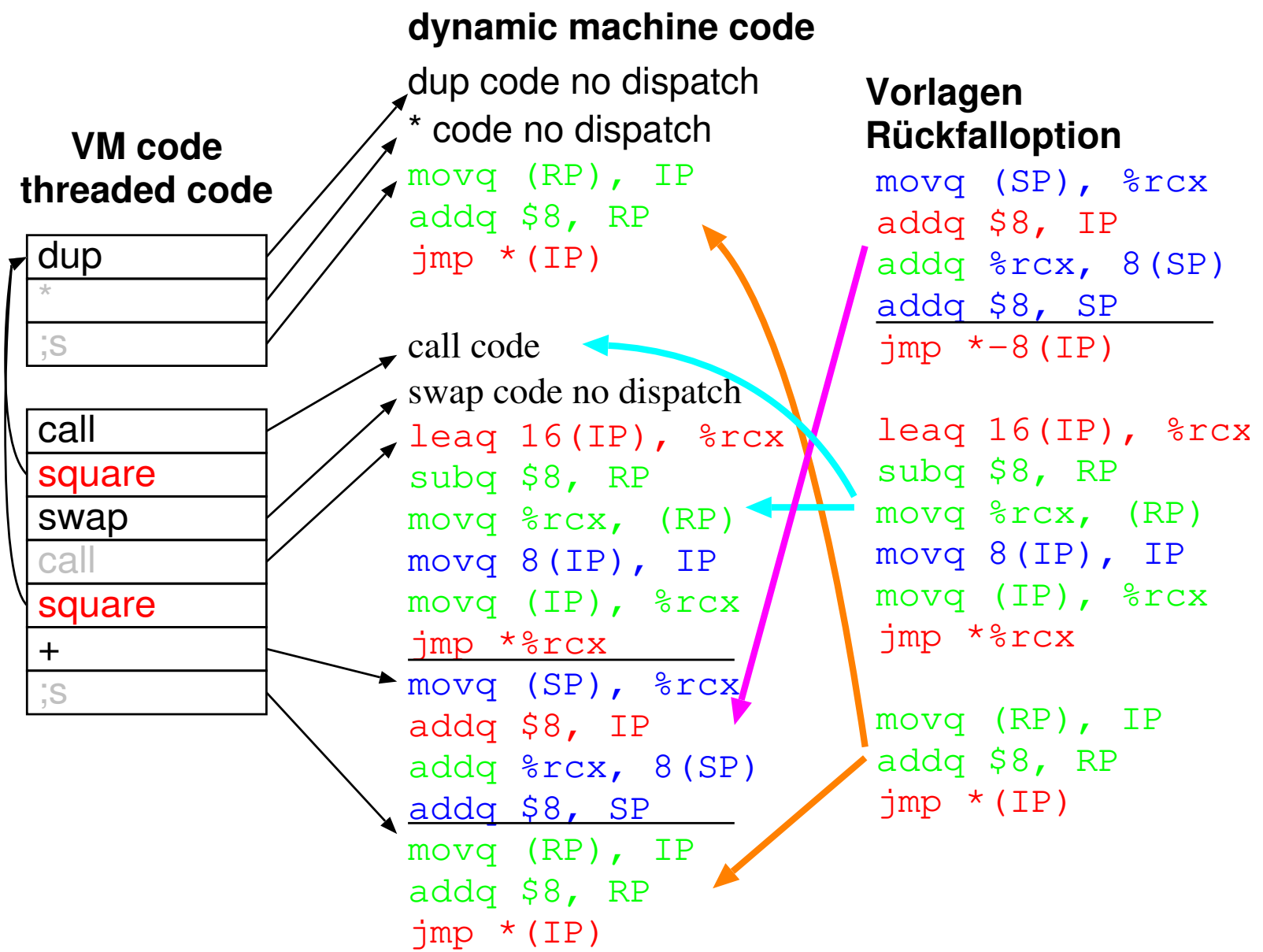
```
typedef void *Label;
typedef Label *inst_t;
Label *vm(inst_t *ip,
          long *sp)
{
 plus: /* + */
  sp[1] += sp[0];
  sp++;
  Label next = **ip;
  ip++;
  goto *next;
  ...
}
```

```
mov (%rsi),%rdx
add %rdx,8(%rsi)
add $8,%rax
add $8,%rsi
mov -8(%rax),%rdx
mov (%rdx),%rdx
jmpq *%rdx
```

- sinnvoll bei 16-bit Adressen

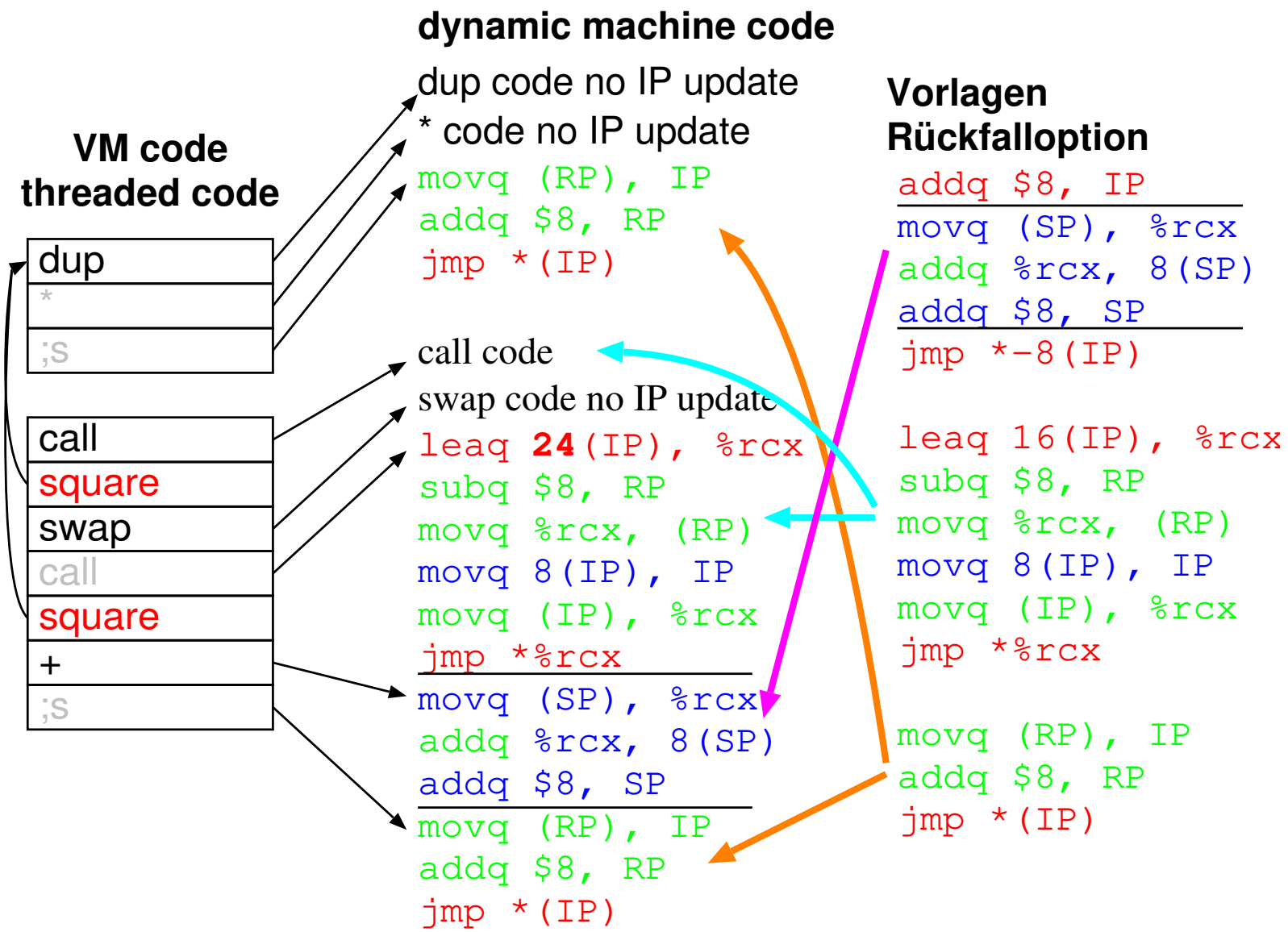


Code-copying (Interpreter/Compiler Hybrid)



- Rückfalloption: threaded code
- Relokatible Vorlagen 2× compilieren, vergleichen
- unbenutzte Befehlsslots
- Kontrollfluss: threaded code dispatch braucht IP
- Immediate-Operanden über IP
- Wie: Label vor Dispatch

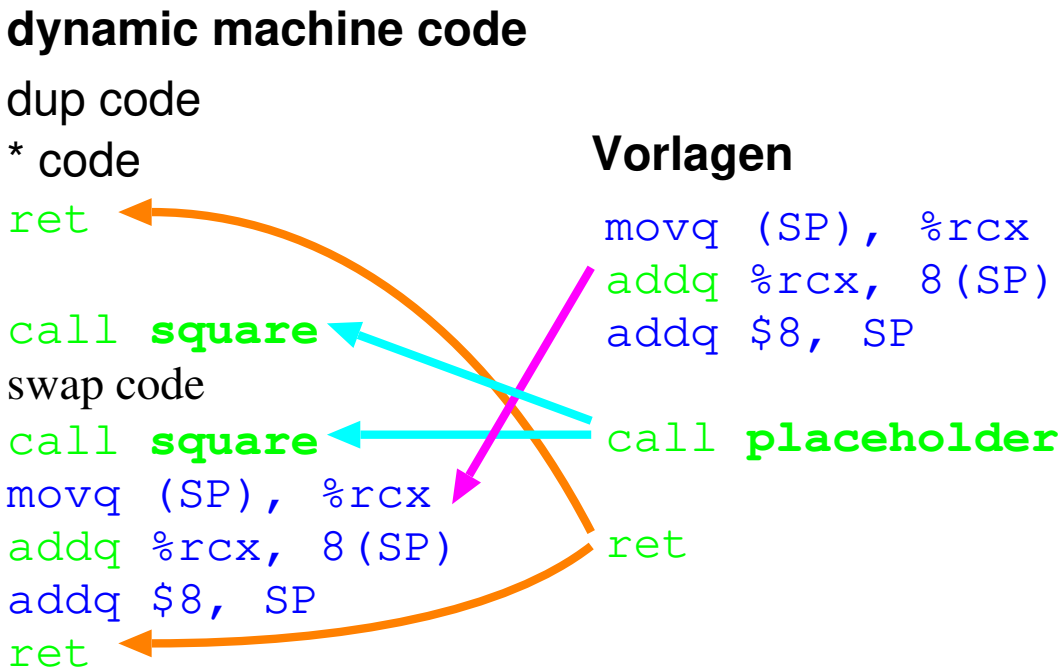
Code-copying: IP-Update-Optimierung



- nur manche Befehle brauchen IP
- nur davor IP update
- reduziert Datenflußabhängigkeiten in Schleifen
  Speedup Faktor 1–3.2
- IP update am Anfang der Vorlage danach Label
- IP-Update-Vorlagen verschiedene Updates
- oder Versionen für verschiedene IP-offsets

Copy-and-patch compilation

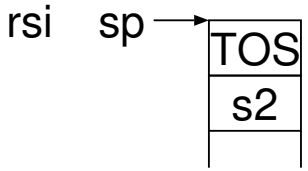
- Statt immediate-Operanden und Sprungziele über IP ...
- Befehle mit immediate-Operanden verwenden und verändern (patch)
- Implementierung: `.o`-Dateien und Relokationsinformation
  oder Vergleich von Vorlagen mit unterschiedlichen immediate-Operanden



- Verwendung von `call` und `ret`
  Probleme mit `n(%rsp)` (AMD64)
  Sicherung des Rücksprungregisters (ARM A64, RISC-V)
- kein Rückfall in Interpreter
  bzw. IP kurzzeitig setzen
- Thunderbird (2000) speedup 1.06–1.49
  PPC7400 (1999) speedup 1.29-1.87
  weniger speedup auf aktuellen Cores?

Stack caching mit einem Zustand

registers memory



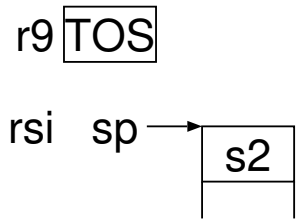
C code

```
sp[1] += sp[0];
sp++;
```

assembly code

```
movq %rsi), %rax
addq %rax, 8(%rsi)
addq $8, %rsi
```

registers memory



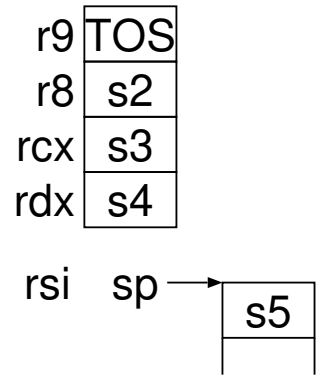
C Code

```
TOS += sp[0];
sp++;
```

assembly code

```
addq (%rsi), %r9
addq $8, %rsi
```

registers memory



C code

```
TOS += s2;
s2 = s3;
s3 = s4;
s4 = sp[0];
sp++;
```

assembly code

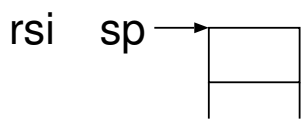
```
movq %rcx, %r10
movq %rdx, %rcx
addq %r8, %r9
movq (%rsi), %rdx
addq $8, %rsi
movq %r10, %r8
```

- Stack-Elemente in Registern
- Viel hilft viel? Eher nicht
- stack pointer updates gleich

Stack caching mit mehr Zuständen

S0: no items in regs

registers memory



C code

```
sa = sp[1]-sp[0];
sp+=2;
```

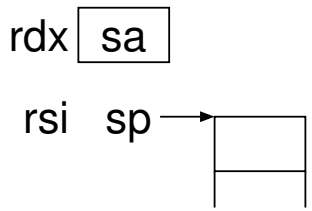
assembly code

```
movq 8(%rsi), %rdx
subq (%rsi), %rdx
addq $16, %rsi
```

after: S1

S1: 1 item in regs

registers memory



C Code

```
sa = sp[0]-sa;
sp++;
```

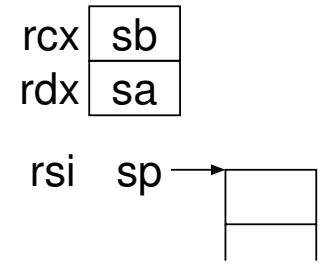
assembly code

```
movq (%rsi), %rax
subq %rdx, %rax
addq $8, %rsi
movq %rax, %rdx
```

after: S1

S2: 2 items in regs

registers memory



C code

```
sa -= sb;
```

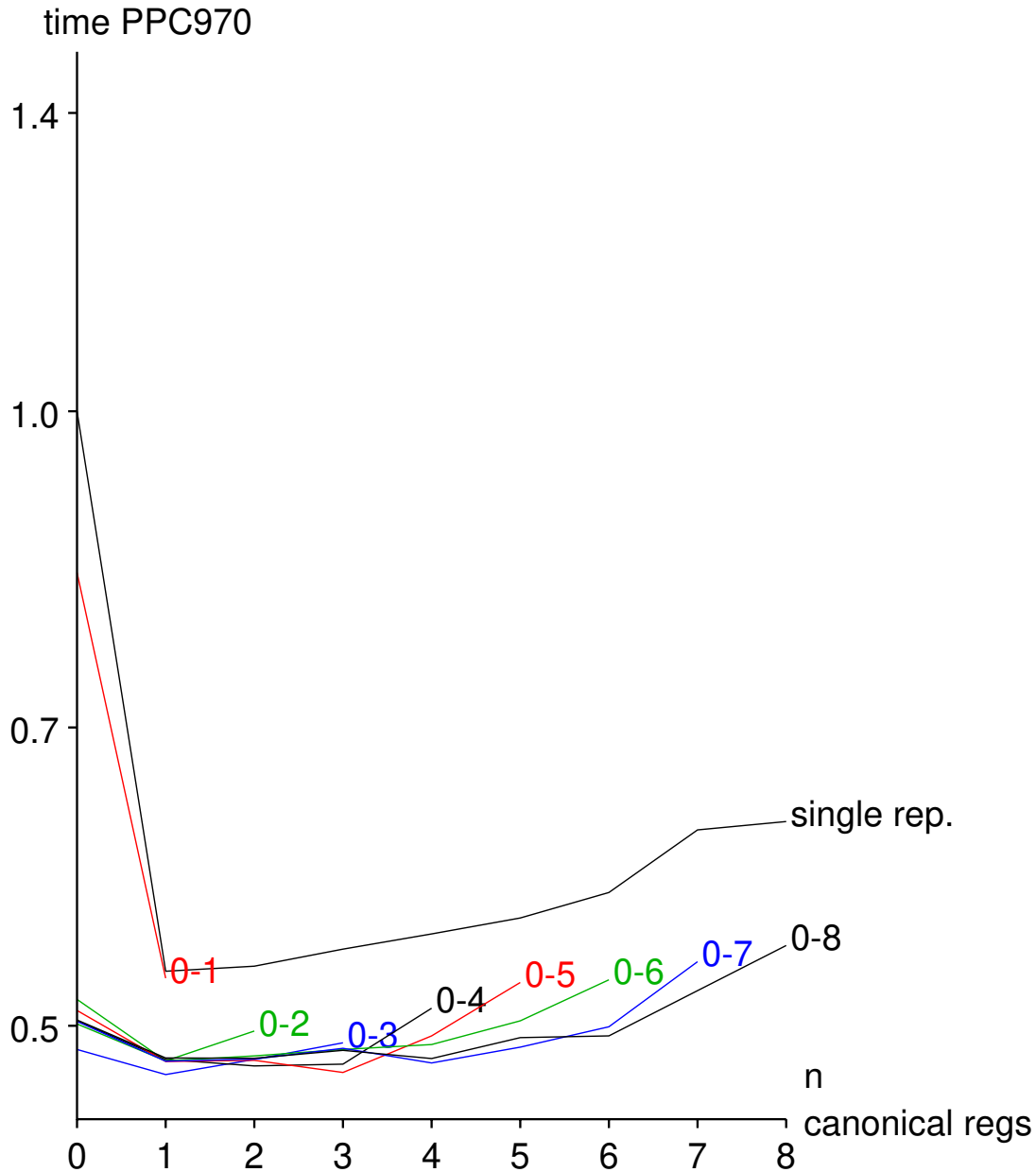
assembly code

```
subq %rcx, %rdx
```

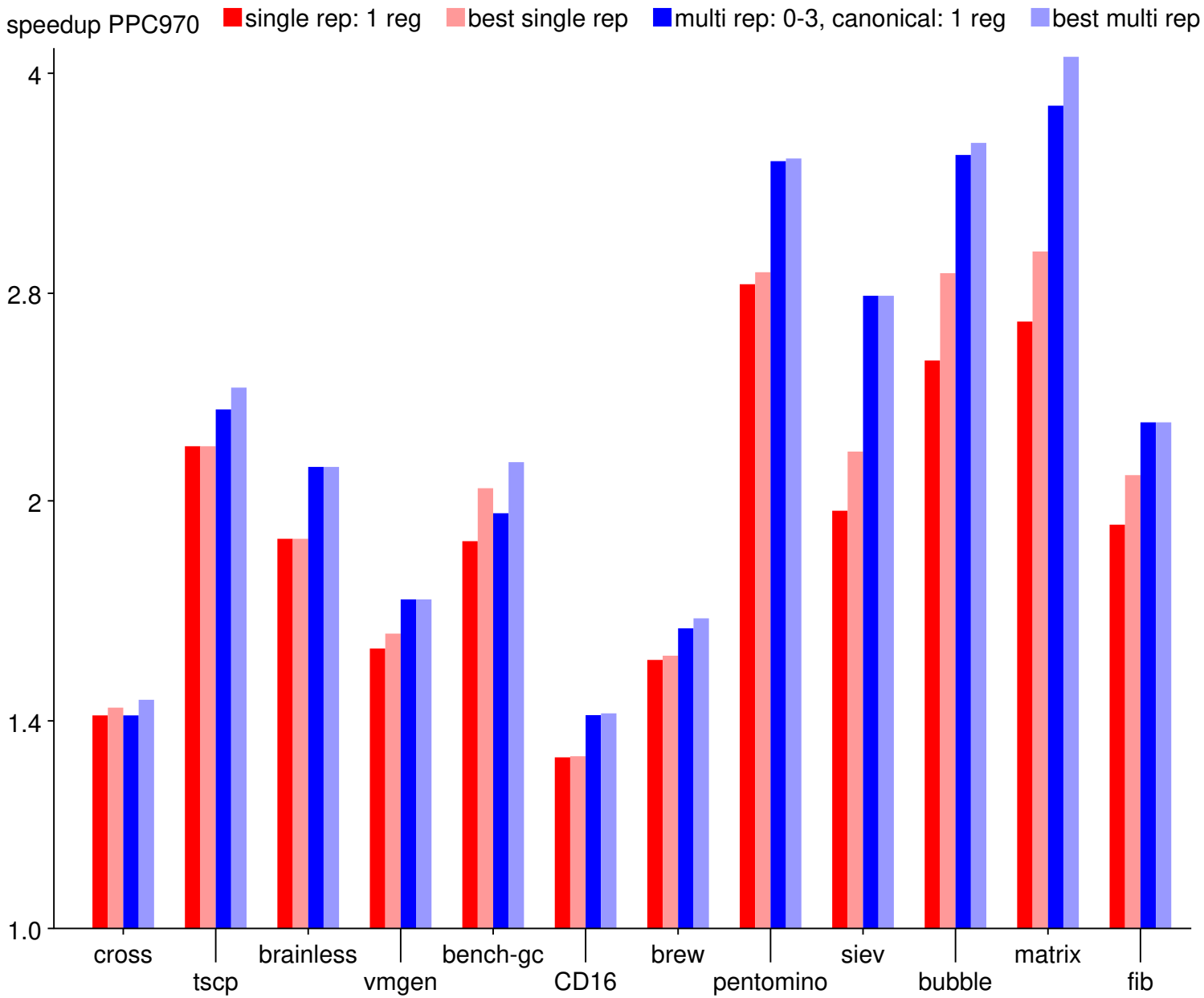
after: S1

- S1 bei Kontrollfluss
- Implementierungen von VM-Befehlen
- + reine Zustandsübergänge
- kürzester Pfad
- weniger stack pointer updates
- weniger loads und stores

Stack caching: Resultate



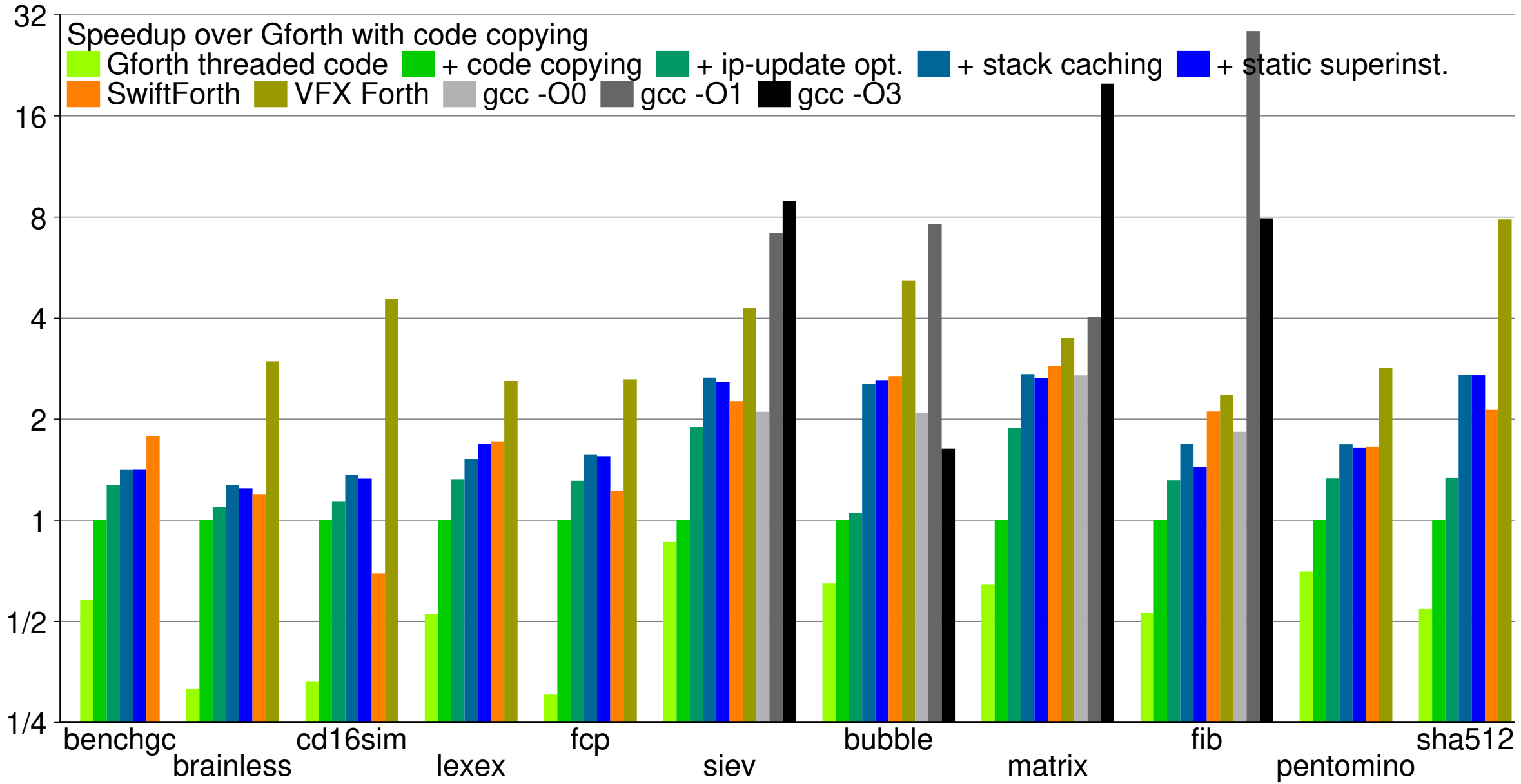
Stack caching: Resultate



VM Superinstructions

- Sequenz von VM-Befehlen als einer implementieren
- Reduziert dispatches
  macht sie vorhersagbarer
- Reduziert Operandenzugriffsoverhead
- Eventuell Optimierungen bei der Nutzlast
- Hilft aber nur dort, wo die Sequenz vorkommt
- Wird in Gforth nur mehr wenig genutzt

Optimierungen in Gforth: Resultate



Register: Dalvik VM

```
class Example {
 static int square(int num) {
     return num * num;
 }

 static int sumsq(int a, int b) {
     int c=square(b);
     return square(a)+c;
 }
}

javac Example.java
javap -v Example.class
```

```
square:
 0: iload_0
 1: iload_0
 2: imul
 3: ireturn

sumsq:
 0: iload_1
 1: invokestatic #7
 4: istore_2
 5: iload_0
 6: invokestatic #7
 9: iload_2
10: iadd
11: ireturn
```

```
square:
 0: mul-int v0, v1, v1
 4: return v0

sumsq:
 0: invoke-static {v3}, square
 6: move-result v0
 8: invoke-static {v2}, square
14: move-result v1
16: add-int/2addr v0, v1
18: return v0

dalvik-exchange --dex \
  --dump-to=Example.dexdump \
  Example.class
# dalvik-exchange alias dx
```

```
class Example2 {
 float x;
 long a[];
 void inca(long inc) {
  for (int i=0; i<a.length; i++) {
   a[i] += inc;
  }
 }
 void inc1() {
  inca(1);
 }
}

inc1:
 0: aload_0
 1: lconst_1
 2: invokevirtual inca
 5: return
```

```
 0: iconst_0
 1: istore_3
 2: iload_3
 3: aload_0
 4: getfield a
 7: arraylength
 8: if_icmpge 27
11: aload_0
12: getfield a
15: iload_3
16: dup2
17: laload
18: lload_1
19: ladd
20: lastore
21: iinc 3, 1
24: goto 2
27: return
```

```
 0: const/4 v0, #int 0
 2: iget-object v1, v5, a
 6: array-length v1, v1
 8: if-ge v0, v1, 32
12: iget-object v1, v5, a
16: aget-wide v2, v1, v0
20: add-long/2addr v2, v6
22: aput-wide v2, v1, v0
26: add-int/lit8 v0, v0, #int 1
30: goto 2
32: return-void

0: const-wide/16 v0, #long 1
4: invoke-virtual {v2, v0, v1},
                        inca
10: return-void
```

Dalvik: Implementierung und Gründe

```
#add-int vAA vBB vCC
movzbl 2(%rdi), %eax
movzbl 3(%rdi), %ecx
movl (%rdx,%rcx,4), %ecx
addl (%rdx,%rax,4), %ecx
movzbl 1(%rdi), %eax
movl %ecx, (%rdx,%rax,4)
movzbl 4(%rdi), %eax
addq $4, %rdi
movq (%rsi,%rax,8), %rax
jmpq *%rax
```

```
#add-int/2addr vA vB
movzbl 1(%rdi), %eax
movl %eax, %ecx
shrl $4, %ecx
movl (%rdx,%rcx,4), %ecx
andl $15, %eax
addl %ecx, (%rdx,%rax,4)
movzbl 2(%rdi), %eax
addq $2, %rdi
movq (%rsi,%rax,8), %rax
jmpq *%rax
```

Anforderungen (2005)

- `mmap()` der VM Dateien mehrere Prozesse
- effiziente Interpretation dispatch pro VM-Befehl weniger VM-Befehle

später JIT, AOT compiler

Register oder Stack?

Register VM

Befehle für Register 0–7

9^3 Varianten von `add`

r0–r2 in realen Registern

```
a = b+c
#add_r3_r4_r5
movq 40(%rsi), %rax
addq 32(%rsi), %rax
movq %rax, 24(%rsi)

#add_r0_r1_r2
leaq (%r8,%rcx), %rdx
```

Gforth

multi-state stack caching

Befehle für locals 0-7

9 `@local`-Varianten

9 `!local`-Varianten

Funktioniert mit +, - ...

```
b c + to a
@local1    1->2
  mov 8(%rbp),%r15
@local2    2->3
  mov 16(%rbp),%r9
+          3->2
  add %r9,%r15
!local0    2->1
  mov %r15,0(%rbp)
```

- Shi et al. 2008
  optimierter register VM code
  nur allgemeine Variante
  register VM ist schneller als
    stack VM ohne stack caching
      viel bei langsamen dispatch
      kaum bei code copying
- auf aktueller Hardware?
  mit spezialisierten Varianten?
  gegen multi-state stack caching?
  mit superinstructions?
- mit realen Registern
  jedenfalls schneller
  Zwischen-VM für Stack-VM

Gleitkommazahlen

- reale Hardware
  RAM enthält ganze Zahlen, Adressen, Gleitkommazahlen
  “General-purpose”-Register für ganze Zahlen und Adressen
  FP/SIMD-Register für Gleitkommazahlen

- Gforth
  Datenstack für ganze Zahlen und Adressen; multi-state stack caching
  FP Stack für Gleitkommazahlen; FP-TOS in FP-Register
  Locals für alles; locals im RAM

- JavaVM
  Stack slots und local slots für alles
  Alles im RAM
  oder kopieren zwischen GPRs und FPRs
  oder Analyse

Datendarstellung

- Oft näher an Programmiersprache als an Hardware
  z.B. JavaVM:
  keine allgemeinen Adressen und Adressarithmetik
  nur Adressen von Objekten
- kleine Typen fixer Größe auf Stack, in Locals, in VM-Registern
- variable Größe, größere Typen oft dynamisch allokiert (Boxing)
- Boxing overhead: long (unboxed) vs. Long (boxed) in Java: Faktor 10
- BigNums mit dynamischer Typprüfung
  häufiger Fall unboxed
  allgemeiner Fall boxed

Typüberprüfung

- keine Typprüfung (Forth)
- statische Typprüfung: keine zur Laufzeit
  Java VM für Basistypen
  Statische Typprüfung auf der Quellsprache und/oder der VM
- statisches Typwissen durch Separierung
  z.B. Datenstack (Integers, Adressen) vs. FP Stack (FP) in Forth
  in Safe Forth zusätzlich Object Stack
- Objekte und Klassen
  oft nur Adressen in Stack/Registern/Locals
  Typinformation am Anfang des Objekts
  Methodendispatch: Typprüfung und Overloading resolution
  jedes Resultat erfordert dynamische Allokation (Boxing)
- dynamische Typprüfung mit Tagging

Tagging

- Tag bestimmt Typ
  z.B. 0 für Integer, 1 für Adresse
- Pointer Tagging
  Grundlage: ganze Zahlen/Adressen
  niederwertigste Bits für Tags: nur ausgerichtete (aligned) Adressen
  höchstwertigste Bits für Tags: begrenzter Adressraum
  z.B. Smalltalk, WAM (Prolog), Lisp-Implementierungen, Ocaml-Interpreter
- NaN-Boxing (Lua, JavaScript)
  Grundlage: Gleitkomma-Zahlen
  64-bit NaN hat Nutzlast von 51 bits: für Adressen
  ganze Zahlen als normale Gleitkommazahlen
  Warum "Boxing"? Bezieht sich auf das Verpacken als NaN

Native-code compilation

- Vor Programmstart: Ahead of time (AOT)
- Beim Programmstart: Load-and-go
- Beim ersten Ausführen eines Teils: Just in time (JIT)
- Mehrstufige JIT-Compilierung
  erste Stufe Interpreter oder schneller Compiler
  zweite Stufe für häufig ausgeführte Codeteile
- Granularität
  Methode (Hotspot)
  oder Trace/Superblock (HotPathVM, TraceMonkey, Pypy)

Webassembly

Ziele

- VM für C/C++
  sprachunabhängig
- sicher
  Sandkiste
  läuft im Browser
- nur Compilation

Design

- Stack und locals
  i32 i64 f32 f64 v128
- Linearer Speicher (Bytes)
  nicht enthalten: alles andere
  für Security
- Globals
- Arrays, Structs, GC
- Strukturierter Kontrollfluss

```
void inca(long a[], size_t n, long inc) {
  for (int i=0; i<n; i++)
    a[i] += inc;
}
void inc1(long a[], size_t n) {
  inca(a,n,1);
}
```

```
block
 local.get 1
 i64.eqz
 br_if 0
 loop
  local.get 0
  local.get 0
  i64.load 3 0
  local.get 2
  i64.add
  i64.store 3 0
  local.get 0
  i64.const 8
  i64.add
  local.set 0
  local.get 1
  i64.const -1
  i64.add
  local.tee 1
  i64.eqz
  i32.eqz
  br_if 0
 end
end
end

local.get 0
local.get 1
i64.const 1
call 0 <inca>
end
```

Prolog

Prolog = Imperative Sprache
+ Unifikation
+ Backtracking

```
append([], L, L).
append([X|L1], L2, [X|L3]) :-
          append(L1, L2, L3).
?- append([a,b],[c,d],L).
?- append(L1,L2,[a,b,c]).
```

Prolog: Warren Abstract Machine (WAM)

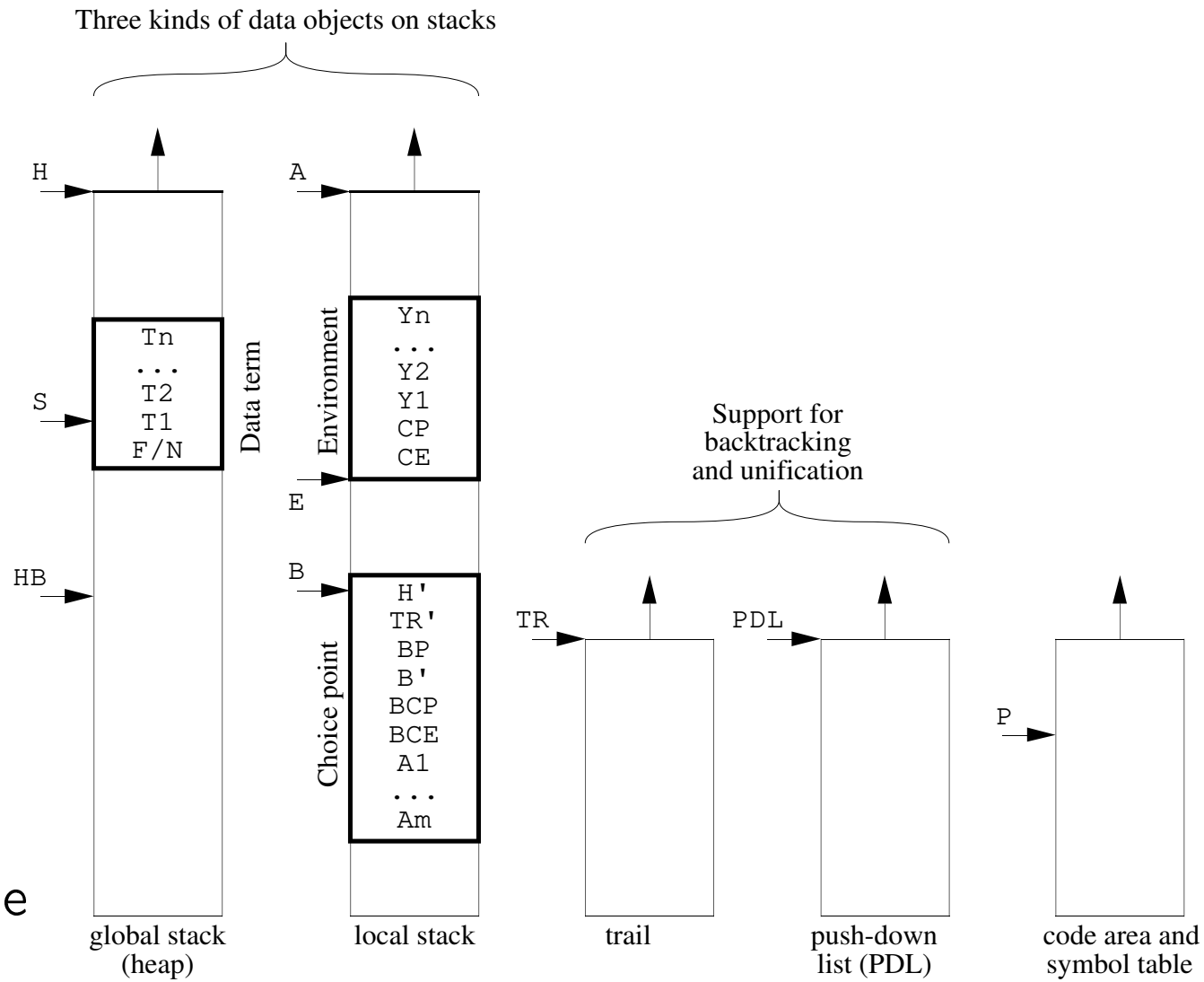
WAM = sequential control (call/return/jump)
+ unification (get/put/unify)
+ backtracking (try/retry/trust)
+ optimizations

```
append([], L, L).
append([X|L1], L2, [X|L3]) :-
  append(L1, L2, L3).
```

append/3:	switch_on_term V1, C1, C2, fail	Jump if variable, constant, list, structure
V1:	try_me_else V2	Create choice point if A1 is variable
C1:	get_nil A1	Unify A1 with nil
	get_value A2, A3	Unify A2 and A3
	proceed	Return to caller
V2:	trust_me_else fail	Remove choice point
C2:	get_list A1	Start unification of list in A1
	unify_variable X4	Unify head: move head into X4
	unify_variable A1	Unify tail: move tail into A1
	get_list A3	Start unification of list in A3
	unify_value X4	Unify head: unify head with X4
	unify_variable A3	Unify tail: move tail into A3
	execute append/3	Jump to beginning

Prolog: Warren Abstract Machine (WAM)

- dynamic typing
- global stack
  structures
- local stack
  local variables
  choice points
  Unifikation von structs
- trail stack
  Variablen
- permanent memory
  code area
  symbol table
- VM-Register
  P, CP, E, B, A, TR, H, HB, S, Mode
  A1, A2...; X1, X2, ...



Prolog: Vienna Abstract Machine (VAM)

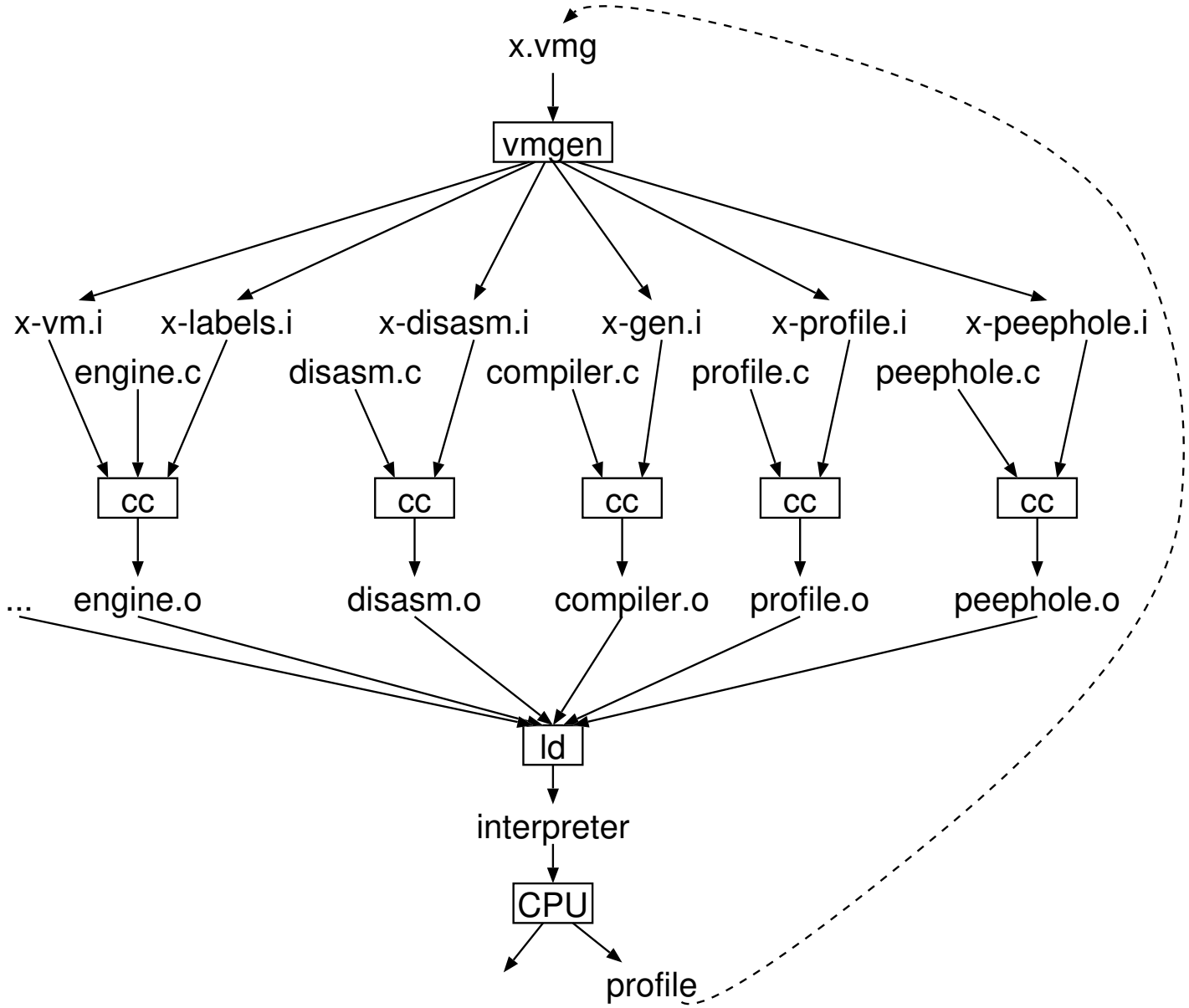
- $\text{VAM}_{2\text{P}}$: Zwei instruction pointer
  Einer für goal (Aufruf)
  Einer für head (Aufgerufener)
- Kombination von goal+head ergibt Maschinencodeadresse
- $\text{VAM}_{1\text{P}}$ kombiniert zur Compilezeit
  für native-code compilation

```
member(T, [T|_]).          [member/2,h-fsttmp, 0, h-list, h-nxttmp, 0, h-void, c-nogoal]
member(X, [_|Y]) :-        [member/2,h-fstvar, 1, h-list, h-void, h-fstvar, 2,
      member(X, Y).                c-goal, member/2, g-nxtvar, 1, g-nxtvar, 2, c-lastcall]
?- member(b,[a,b,c]);
?- member(X,[a,b,c]);
```

VMgen: input

```
\ stack definitions:
\E stack data-stack sp Cell

\ stack prefix definitions:
\E inst-stream  stack-prefix #

\ type prefix definitions:
\E s" int" single data-stack type-prefix i

\ simple instruction definitions:
iadd  ( i1 i2 -- i )
i = i1+i2;

ipush ( #i -- i )

\ superinstruction definitions:
ipush_iadd = ipush iadd
```

VMgen: files and processing



VMgen: C code für `iadd` für VM interpreter

```
I_iadd: {                   /* label */
int i1;                     /* declarations of stack items */
int i2;
int i;
NEXT_P0;                    /* dispatch next instruction (part 0) */
i1 = vm_Cell2i(sp[1]); /* fetch argument stack items */
i2 = vm_Cell2i(spTOS);
sp += 1;                    /* stack pointer updates */
{                           /* user-provided C code */
#line 6 "mini.vmg"
i = i1+i2;
}
NEXT_P1;                    /* dispatch next instruction (part 1) */
spTOS = vm_i2Cell(i);  /* store result stack item(s) */
NEXT_P2;                    /* dispatch next instruction (part 2) */
}
```

VMgen: C code für ipush_iadd für VM interpreter

```
I_ipush_iadd:
{
Cell _IP0;
Cell _sp0;
Cell _sp1;
NEXT_P0;
_IP0 = vm_Cell2Cell(IPTOS);
_sp0 = vm_Cell2Cell(spTOS);
INC_IP(1);
/* ipush ( #i -- i ) */
{
  int i;
  i = vm_Cell2i(_IP0);
  {
  }
  _sp1 = vm_i2Cell(i);
}
```

```
/* iadd ( i1 i2 -- i ) */
{
  int i1;
  int i2;
  int i;
  i1 = vm_Cell2i(_sp0);
  i2 = vm_Cell2i(_sp1);
  {
  i = i1+i2;
  }
  _sp0 = vm_i2Cell(i);
}
NEXT_P1;
spTOS = vm_Cell2Cell(_sp0);
NEXT_P2;
}
```

VMgen: Alpha code für VM Interpreter

```
iadd:
ldl     t0,8(s3)  ;i1 = vm_Cell2i(sp[1]);
ldq     s2,0(s1)  ;load next VM instruction
addq    s3,0x8,s3 ;sp += 1;
addq    s1,0x8,s1 ;increment VM instruction pointer
addl    t0,s4,s4  ;i = i1+i2;
jmp     (s2)      ;jump to next VM instruction

ipush_iadd:
ldq     t0,0(s1)   ;_IP0 = vm_Cell2Cell(IPTOS)
ldq     s2,8(s1)   ;load next VM instruction
addq    s1,0x10,s1 ;increment VM instruction pointer
addl    t0,s4,s4   ;i = i1+i2;
jmp     (s2)       ;jump to next VM instruction
```

VMgen: Unterstützung für VM-Codeerzeugnung

```
/* generated code for gen_ipush */
void gen_ipush(Inst **ctp, int i)
{
  gen_inst(ctp, vm_inst[1]);
  genarg_i(ctp, i);
}
```

```
/* Im handgeschriebenen Code: */
/* Verwendung von gen_... Routinen */
expr: num            { gen_ipush(&p, $1); }
    | expr '+' expr { gen_iadd(&p); }
```

VMgen: Optimierungen

- Instruction Scheduling
- Top-of-stack caching
- Stack stores eliminieren `dup ( i - i i )`
- Separate Sprünge für Sprungvorhersage

Filegrößen (Gforth)

```
 3673 prim                          source lines
  423 peeprules.vmg
 4096 total
66141 engine/prim-fast.i            generated lines
25583 engine/prim.i
139696 total
39215 engine/engine-fast-ll-reg.o   text size bytes
23967 engine/engine-ll-reg.o
```

VMgen: Anwendungen

- Gforth (erweitertes VMgen)
- Cacao interpreter (JavaVM)
- Unladen Swallow (Python)
- mindestens zwei weitere, ähnliche Generatoren