Konstrukce překladačů

HW scheduling X **scheduling překladačem**

HW scheduling:
- podvody → když tak pak rollback
- jedna posl. instr. → bez větvení !! (skokové instr. tam jsou, ale jen se jimi ověřuje, zda predikce byla správná)
- ale nemůžou tam být úplně komplikované schedulovací alg. (NP-úplný problém) → stále má smysl scheduling překladačem – sofistikovanější alg. + větší okno instrukcí (HW ~ 100 instr.) ← μ-op.

Scheduling překladačem:
- jen přípustné permutace
- smyčky, podmínky !! → nemáme prostředky, jak HW dokáže dělat spekulativní vykonávání
- neznáme skutečné závislosti → musíme předp., že tam jsou (!)
- známe závislosti přístupů do paměti !!

blended code = optimalizovaný jen pro nějakou generaci procesorů ("průměr"), ne pro konkrétní procesor
- "průměr" – neodpovídá skutečnému konkrétnímu HW

link-time optimization = některé části kompilace se udělají až při linkování (např. aby se procedura nekompilovala 1x víckrát, ale jen jednou, nebo třeba pro inter-procedural optimalizace, kdy potřebujeme vidět celý program)

profile-driven optimization = nejdřív kompilace s instrumentací, pustíme a pak znova kompilace s profilem
⇒ víme, co je důležité
⇒ optimalizace !!

"DAG" = data-flow DAG nebo DAG závislostí (v jednom BB)

call graph X **call tree**
- call graph → statický – možná volání mezi procedurami
- call tree → program jsme pustili a v tomhle je, jak se to volalo, kterou cestu lze najít v call graphu

basic block (BB)
- bez větvení – ale exceptions jo
- skok na konci; >2 větví: switch, catche

↓

control flow graph procedury
- vrcholy jsou BB

Závislosti

= povinnost provést jednu operaci po jiné

důvody:
- datová závislost (dependence)
- antidependence
  = závisí to na sobě, ale nepředává si to data
  - R-W
  - W-W – někdy i nevíme, jestli vedou do stejného místa – např. přes pointer → ale mohlo by ⇒ je to antidependence
  - jiné
- control dependence
  - Provedení operace jenom v jedné větvi větvení

JCC → LOAD, STORE

NELZE! – nešlo by zrollbackovat

NELZE! nepřípustné ⇒ může způsobit výjimku ⇒ nemůžeme přesunout, protože by mohlo k té výjimce dojít, ale taková větev by v tom ifu byla not taken

Proměnná bez aliasu
- lokální
- pozor: int x[3]; int a; x[3] = ... → alias a ⇒ UB (nekorektní, "nemyslíme se to")

Alias
- když jsou dvě přístupové cesty k jednomu paměťovému místu
- např. dva pointery, přístupy do pole a[i], a[j]
- alg. nerozhodnutelné (i != j)

Paměťové místo
- část proměnné nebo dyn. alok. paměti
- překladač neumí pojmenovat, jen ví, že to někde je – typicky přes pointer

Typy

- **skalární / jednoduchý / (atomický) typ (scalar)**
  = typ, se kterým umí pracovat cílový procesor
- **Složený typ (aggregate)** – HW nezná
  - pole, structy, stringy → jednotlivé skalární instr.
  - překladač ví, k jaké položce přistupujeme (u structu) a u polí, která indexujeme konstantou nebo "pevně" daným cyklem s konstantami ⇒ unroll
    → optimalizace ☺
    (rozbití na elementy ⇒ jde pak třeba dávat do registrů)

Zarovnání
- řeší linker ← moduly
- řeší linker ← segmenty procesu
- zarovnání na stránky
- stat. prom. – vůči začátku modulu → řeší back-end
- lok. prom. → řeší back-end (např. SP, offset vůči SP)
- prvky struktur → řeší front-end
- požadavky dané HW ⇒ padding nebo permutace prvků struktury (v C se nesmí)
  - striktní – např. Arm
  - optimalizační – např. Intel, AMD

Proměnné

- statické/globální → jediná instance, umisťuje překladač
- lokální – proměnné a parametry + pomocné proměnné
- thread-local → jedna instance per vlákno

Live range = místa v proceduře, kdy je proměnná zapotřebí

– zkoumá se u skalárních lok. prom. bez aliasu

⇓

do registrů!! + víme, že zápis modifikuje celou proměnnou

... = x
⋮ } tady nepotřebujeme skladovat x → místo něj tam může být něco jiného s disj. rozsahem platnosti (regist. či zás.)
x = ...

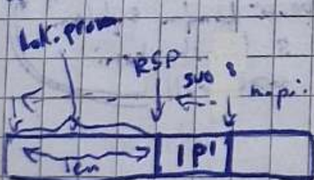
– variable splitting/renaming = rozdělení proměnné s nesouv. rozsahem platnosti

Alokace

- statická/globální → pevná adresa (překladač rozhoduje, kde to bude)
  – pokročilé překladače analyzují, která vlákna používají které statické proměnné, aby je umístily dál od sebe pro zabránění false sharingu
- dynamická
  – knihovní volání – překladač neřeší
- registrová
  – skalární lok. proměnné bez aliasu – **nesmí na to být namířen pointer**
- zásobníková → složené či aliasované lok. prom. + prom., které se nevešly do reg.
  – zásobník dán procesorem, nebo emulován pomocí GLOBAL reg, addr – branch and link instr. {reg:=IP, IP:=addr} + BR reg ≈ ret
  (↳ volání funkcí)

– před reg. alokací je to (strojovém) v mezikódu ve virt. registrech – optimisticky by instr. pracují s prom., jako by byly v registru

– spill kód = kód, který se tam musel přidat navíc ⟸ když se zjistí, že se to nevejde, tak se to musí předělat → potenciálně se na to pak musí znova pustit nějaké optimalizace



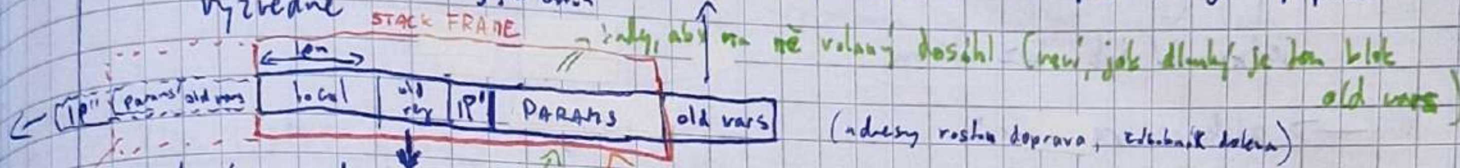
např. I. akt.: IP → CALL addr { sub rsp, 8 ; mov [rsp], IP' ; jmp addr } – po konci té call instr. IP := addr; IP' → Konst.

RET { add rsp, 8 ; jmp [rsp-8] }

není přístupný → nejde takhle emulovat

Volací konvence

- povinnost zachovat obsah registrů (všech/některých/žádných)
  - volající u těch reg., které potřebuje zachovat a nemá záruku, že budou zachovány, pomocí pushů dá před callem na zásobník a pak si je popem vyzvedne



(adresy rostou doprava, zásobník doleva)

  - volaná procedura musí zachovat hodnoty reg., u kterých má volaný zaručeno, že budou zachovány → buď na ně nesahá, nebo je ona musí pushnout na zás. a před returnem zase vrátit
    (nebo si zvětší prostor len a nastaví to tam normálními mov)
  - občas se takhle na zás. ukládá něco zbytečně – aby se to vyvážilo, tak se to rozdělení reg., které se musí a nemusí zachovat, dělá půl na půl

(když se to dělá takhle)

- umístění parametrů – reg. + zás.
  (stavový automat, kterému se řeknou typy těch parametrů a on řekne, do jakých reg. je dávat)
  - bývají jen vstupní – prostě se po návratu z callu jenom přes ně posune zás.
- odpovědnost za úklid zás.
- umístění návratové hodnoty
  - hodnotově – v té části PARAMS – z toho si to volaný pak překopíruje do lok. prom.
  - pointerem – do PARAMS se dá pointer na data, která jsou v lok. prom.
  - reg.
- úprava jména procedury kvůli linkeru – mangling
  - např. namespacy, třídy, ...

Fáze

- instr. selection
  - typicky 1:n
- instr. scheduling
  - NP-úplná úloha
- reg. alloc
  - NP-úplná úloha (barvení grafu)

Reprezentace

- mezikód střední úrovně
  - nesekvenční – uvnitř BB jsou grafy
  - část. sekv. – uvnitř BB jsou pseudoinstr.
  - plně sekv. – "bez BB" – celé je to jeden proud instr.
  - součástí musí být i třeba seznam glob. prom., procedur a další informace nad rámec samotného kódu
- mezikód nízké úrovně
  - ekvivalenty stroj. instr.
  - ve výsledku chceme plně sekv. formu, ale optimalizace se lépe dělají na těch část. sekv.
    → do plně sekv. formy se přechází až v pozdních fázích
- většinou nemá smysl moc používat nějaké komplikované stroj. instr., protože ten procesor si to pak stejně rozkládá na μ-operace + víc menších instr. dává větší prostor pro optimalizace v rámci schedulingu překladačem

Analýza rozsahu platnosti (live range analysis)

DF: Proměnná je v bodě a živá ≡ ∃ cesta v CFG do bodu b t.ž.:
- na cestě a→b není žádný zápis do té proměnné
- v bodě b je tato proměnná čtena

jednoduchá lok. bez aliasu

→ alokace registrů

Alg.: CFG V = BB, E = CF přechody, VAR … proměnné

1) lok. analýza – uvnitř BB
   - W … do proměnné se zapisuje
   - R … tu proměnnou se tam před 1. zápisem čte

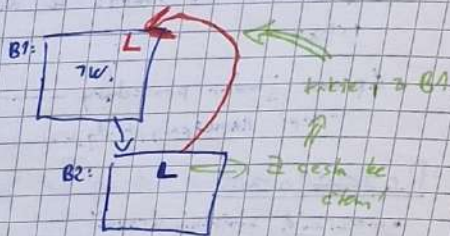
   funkce $BB \times VAR \to \{0,1\}$

2) glob. analýza
   - L … proměnná je na začátku BB živá

   1) $\forall v \in VAR$:
   2) $\forall b \in BB$: $L(b,v) \leftarrow R(b,v)$
   3) Dokud dochází ke změnám:
   4) $\forall (B1, B2) \in CF$:
   5) $L(B1, v) \mathrel{|}= \neg W(B1,v) \,\&\, L(B2,v)$

   $O(|BB| \cdot |CF| \cdot |VAR|)$

3) dopočet
   - zač. a konce BB



Alokace reg.

- vstupem je části. sekv. kód – ekvivalenty stroj. instr. s virt. reg., optimistický
  + máme spočtené rozsahy platnosti prom. – intervaly, kdy je prom. živá

  (jednoduché prom. bez aliasu v reg.)
- výstupem je přiřazení virt. reg. fyzickým

Alg.:
1) vytvoříme matici kolizí C: $c_{ij} := \begin{cases} 1 \dots \text{prom. } v_i \text{ a } v_j \text{ mají nepr. průnik platnosti} \\ 0 \dots \text{jinak} \end{cases}$
2) spočítáme odhad ceny za spill kód – cena za ∀ přístupy k proměnné, když nebude v reg.
3) barvení grafu (VAR, C) n barvami – NP-úplná úloha
   - ↳ # fyz. reg.
   - ⇒ heuristické řešení
     - vrcholy s < n sousedy lze hned obarvit
     - odebereme vrcholy na zásobník
     - pak je postupně ze zás. přidáme
4) doplnění spill kódu
   - no jo, ale ten přece taky může potřebovat nějaké reg.!
   - při složitosti → vyhradíme na to zvláštní reg., které se neúčastní alg. alokace
   - → u ∀ instr. pesimisticky předp., že bude spill kód a na to prostor rezervujeme

Okrajové poznámky:
- pak by s nejmenší cenou za spill kód nebo vrcholy s co největším/nejmenším stupněm
- půjde obarvit → obarvíme, přidáme do grafu
- nepůjde → nebude v reg. (a nevrátíme ho do grafu)
- malá šance, že půjdou obarvit
- příležitost k optimalizacím, kterou z dostupných barev použijeme
- kdyby byly ve nějak seřazený a větší vzdálenost, tak počet potřebných registrů

Optimalizace:

minimalizace přesunových instr.
- ztotožnění virt. reg. spojené přes.-instr. (pokud to jde) → kdyžtak může pomoct duplikace kódu
- úprava alg. – do grafu kolizí přidáme nový typ hrany, která bude říkat, že by bylo dobré, kdyby dané vrcholy měly stejnou barvu
  → řídíme se tím při volbě barvy

Vliv instr. sady – ∀ operand instr. má určenou množinu reg., v nichž může být

- ortogonální ≡ ∀ dvě množiny povolených reg. jsou buď identické, nebo disj. (← ortog. velikosti)
  → alokace reg. pro ∀ množinu zvlášť
- neortogonální
  → do toho grafu kolizí přidáme úplný graf s vrcholy odpovídajícími všem fyz. reg. a spojíme ty VARs s těmi reg., kde nemohou být + ty nové vrcholy dostanou nejvyšší prioritu (nikdy je nevybereme jako oběť)
  + to, který registr odpovídá které barvě, je určeno tím, jakými barvami jsou obarveny ty fyz. reg.

- může ale být problém, že vyjde, že je nějaká proměnná spojena se všemi fyz. reg. ⇒ obarvení ∄ ⇒ strčilo by se to do paměti ⇒ FUJ (i když by reg. byl dostatek)
  → upravíme, aby se to přesunulo mezi reg. → kolem každého použití proměnné budou oblasti dohlídající se reg-reg move instrukcemi → tím se životnost té proměnné rozdělí do víc, které se překrývají rozumně → jde použít dokonce jednodušší alg. na barvení grafu (!)

- neortogonalitu taky způsobuje volací konvence, která určuje, v jakých reg. co může být (např. jedna se použije jako 2. parametr nějaké funkce a pak jako 3. parametr jiné funkce)
  → nepůjde přiřadit reg.

Poznámky na okraji:
- třídí, když je nějak seřazený a vždy vybírám ten nejmenší, tak tím minimalizujeme počet potřebných registrů
- překážka k optimalizacím, kterou z dostupných barev použijeme
- malá šance, že půjde obarvit
- pak by s nejmenší cenou za spill kód nebo vrcholy s co největším / nejmenším stupněm → třeba budou nějací sousedé mít stejné barvy → kdyby(!)
- půjde obarvit ⇒ obarvíme a přidáme do grafu
- nepůjde ⇒ nebude v reg. (a nevrátíme to do grafu)

Scheduling – rozvrhování instrukcí

(eliminace závislostí ← → co nejlepší využití pipeline)

- typicky těsně před fází alokace reg.
  + když alokace reg. způsobí vytvoření spill kódu, tak se pak scheduling pustí znovu
- ILP
  - pipelining
  - MIMD – superskalární proc.
    - víc pipeline – v jednu chvíli se provádí stejná fáze několika instr.
  - SIMD
    - neřeší se při schedulingu, ale dřív – při generování instr. a často když je ten kód ještě ve stromové formě
- provádí se na jednom BB a pak se rozšíří mírně dál:
  - sw pipelining – pro BB, které představují cykly
  - trace scheduling – slití posl. BB do jednoho – ale pozor na závislosti, odskoky a přískoky
    - ale pořád v rámci jedné procedury (pokud nebyla inlineována)

- vstupem je dag s uzly = instr. a hranami popisujícími závislosti (délky těch hran)

+ model procesoru, který řekne, jak je určitá permutace instr. dobrá
  - latence – časování závislých dvojic instr. → jako ref. bod se bere okamžik vstupu instr. do výp. jednotky (≠ měřeme doba dekódování instr.)
  - rezervační tabulky – kdy která instr. obléhuje které výp. jednotky
  - + kap. procesoru → #výk. jednotek

- výstup: uzlům toho dagu přiřadíme časy (v jednotkách taktů procesoru)
  (a hlavně určíme topolog. průchod tím grafem)

  skutečné časování se ale může lišit – třeba kvůli rozpracovaným instr. z předchozího BB nebo kvůli nepřítomnosti dat v cachi (překladač předpokládá, že jsou v nejbližší cachi) nebo kvůli době dekódování instr.

- předávání dat v reg. :)
- předávání dat přes paměť :(→ musí se opatrně
- při schedulingu p. alok. reg. přibydou antidependence na prom., které byly dány do stejného reg.
- kontrolní dependence v rámci trace schedulingu a sw pipeliningu
- další technické závislosti – např. manipulace se zás.

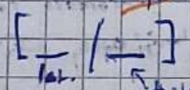
Základní alg. – List scheduling

→ chceme, aby ∀ instr. doběhly co nejdřív

- jdeme postupně – vždy máme nějaké připravené instrukce, z nichž jednu rozvrhneme
  → na základě délky krit. cesty = délka od té instr. do konce BB
  - vezmeme tu připravenou instr. s nejvyšší touhle délkou
  - ∀ instr. je to min. součet délek, kdy je s akt. rozvržením schopna být ukončena – i s latencemi předchozích instr., které už jsme rozvrhli.
- $O(|E| \cdot (\log|V| + |R|))$ ← vel. rezerv. tab.

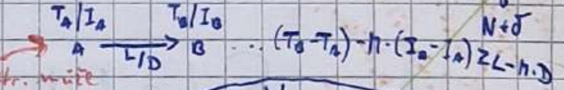
Vylepšení

- branch-and-bound
  - backtrackingem zkusíme, jestli se nám nepovede najít lepší řešení
  - po několika pokusech se zažírne a lepší řešení už se přestane hledat
- SW pipelining → skok na začátek
  - pro cykly – přes hranice BB
  - Unroll-and-compact
    - rozvineme ten cyklus a na to pustíme ten normální alg.
      ⟹ nejdřív hrajeme roli jen hrany s /0, v další iteraci i s /1, ...
    - představujeme si, že je tam ten BB nekonečně-krát
    - řešíme asymptotické chování
      → $T = T_0 + k \cdot N$ (k = #iterací; N = #taktů na iteraci → minimalizujeme)
    - detekujeme, kdy se už zacyklíme
    - kritická smyčka → cyklus v grafu záv. s největším podílem součtu latencí a součtu iterací → dolní odhad, jak to může být prudké (ale může to kolidovat v rez. tab.)



#iterací, přes které je ta závislost

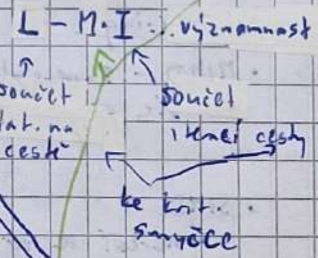
- modulo scheduling
  - snažíme se přímo najít rozvrh s periodou M (když ne, pak zkusíme M+1, M+2, ...)
    → lze také omezit podílem potřebných a dostupných výk. jednotek procesoru
  - rez. tab. používáme modulo M, instr. rozvrhujeme do prostoru čas × iterace
    → předp. $M \in \mathbb{N}$

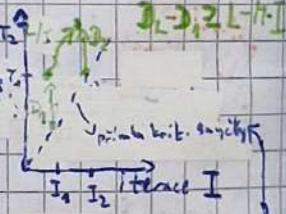


∀ instr. může hrát roli A i B ⟹ příp. i kruhy, kdy musí být rozvržena

vedou z k' první → acyklické :)

- priorita instr. při rozvrhování: $L - M \cdot I$ … významnost





$T_1 - M \cdot I_1 = D_1$

$T_2 - M \cdot I_2 = D_2$

směrnice M

→ instr. rozvrhneme na co nejmenší čas, to pak opravíme přičtením významnosti ⟹ nejvyšší prioritu má ta instr. s nejvyšším tímhle číslem ⟹ rozvrhneme ⇓ opravíme podle rezerv. tab.

- variable expansion
  - pro eliminaci antidependencí
  - pomocí duplikace kódu a přeznačení proměnných
    - nejdřív duplikace a pak rozvrhneme
    - nejdřív eliminace antidependencí (přeznačením) a pak duplikace a pak rozvrhování
  - ale pozor na spill kód

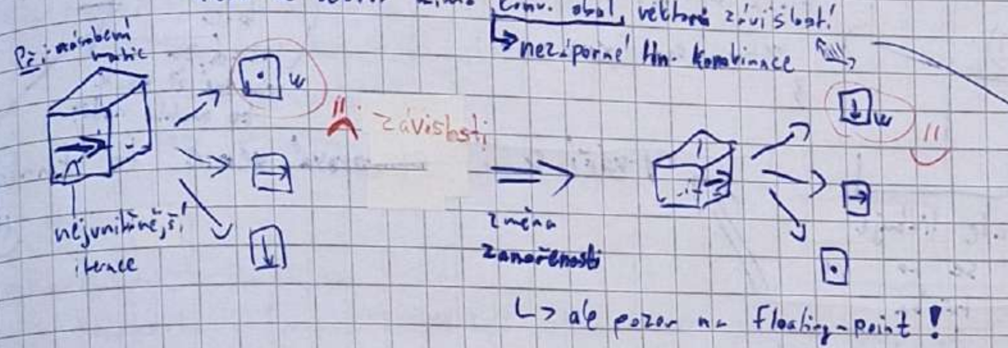
Jemnozrnný paralelismus
- ILP → scheduling
- SIMD

Hrubozrnný paralelismus
- vlákna
- SMP
  - hyper-threading
  - multicore, multisocket
- NUMA
- Cluster

strip-mining – dělení na nezáv. bloky

Vektorizace – jemnozrnný paralelismus

- překladač si musí být jistý, že ta transf. bude korektní!
- cykly musí mít předvídatelný # iterací
- cykly s afinním chováním – +/*/CMP
  - → afinní transformace:
    - loop reversal – výměna zanoření ↗ užitečné nejen pro vektorizaci, ale i ILP
    - loop skewing – změna souřadného systému v iter. prost.
      - hledá se vektor mimo konv. obal vektorů závislostí!
        - → nezáporné lin. kombinace



Presburgerova aritmetika

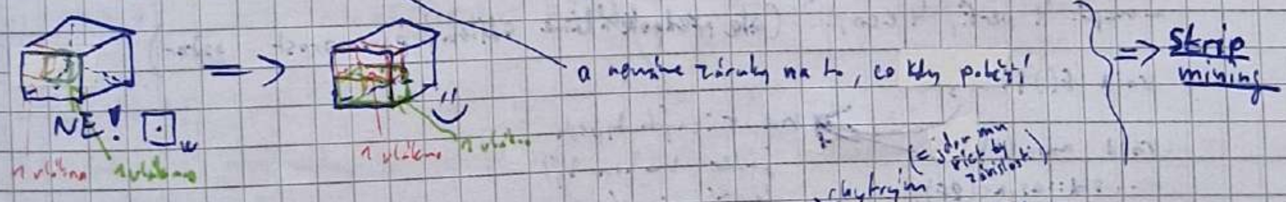
└> ale pozor na floating-point!

- meze musí být konstanty (ne nutně známé za překladu, ale prostě se nesmí měnit)
- polyhedrální kompilace – ten prostor se natrhají na n-dim. mnohostěny
  - loop skewing je speciální případ tohohle
  - musí se zapnout!
  - např. POLLY v LLVM

Loop fusion
= spojení smyček, které jsou za sebou
for(...
for(...
mohlo třeba vzniknout inlinováním nějakých funkcí

Hrubozrnná paralelizace

- typicky ne u nejvnitřnější smyčky, ale naopak u nejvnějšnější ⇒ nezávislé :) bloky + dostatečně velké, aby se to vyplatilo
- trumíme nějakou knihovnu – OS, framework, ...
  - → neumíme řídit závislosti ⇒ potřebujeme mít zcela nezáv. bloky



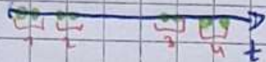
⇒ Strip mining

- → lepší je, když máme k dispozici taskový paralelismus s runtime schedulingem (chytrým) (a task stealingem :)) (= jde mu říct ty závislosti)
  - ⇒ řídíme si to, jak to trumíme :) ⇒ synchronizace v tom scheduleru k řešení závislostí
- chceme nahoře mít ty nezáv. bloky a dole taky, abychom mohli dělat jemnozrnnou paralelizaci, závislosti mohou být v rohách mezi (třeba 2-rozměrný cyklus tohle neudělá)
  - ⇒ typicky ale máme málo vrstev (třeba 2-rozměrný cyklus tohle neudělá)
  - ⇒ typicky ale máme málo vrstev a už to vyjde (a může to pomoct cachím)
  - ⇒ rozdělíme na víc bloků
    - → a nad nimi děláme strip mining

Optimalizace pro cache

→ prostorová lokalita

→ časová lokalita
→ ideálně přístupy hned vedle sebe, ale tím zase rozbijeme přístupy jiné
⇒ z-křivka → nepravidelné přístupy :(
⇒ Kompromis



Problémy s linkováním

a.c | b.c

```
for(...)
  a[i] = f(b[i], c[i]);
```

```
float f(float x, float y){
  return sqrt(x*x + y*y);
}
```

ale když se to linkuje dynamicky, tak to nepůjde inlinovat a vektorizovat

když ne, tak to linker udělá v rámci link-time optimizations

⇒ vytváří se i vektorová verze té funkce f :)

když překladač toho b.c uzná za vhodné, že by se to mohlo hodit

třeba je u toho nějaká #pragma

Heterogenní jádra

– např. 4 perf., 4 eco, ... nebo dokonce GPU (ale předpokládáme sdílený paměťový prostor)

```
void f(){...}
void main(){
  .. std::async(&f);
}
```

thread pool

→ na různých typech jader mohou být lepší různé optimalizace a nebo taky dokonce mohou mít jinou instr. sadu

⇒ je potřeba mít víc verzí f()
a rozšířit ten koncept pointeru na funkci

→ tabulka ?
→ víc pointerů v jednom

Analýza aliasů (points-to analýza)

→ kam pointery míří (v Javě escape analýza = jestli pointer vrácený z new z dané funkce může utéct ⇒ když ne, tak se může destruovat na konci té funkce ☺)

↳ jestli 2 výrazy označují stejnou paměťovou lokaci

- indexy !! ☹
- překryvy pointerů !!

- myšlenkově dynamicky ⇒ obrovský graf V = prom. + dyn. alok. bloky, E ⇔ points-to + čas. razítka, kdy to platilo
- → staticky
  - grafy vzniklé různými běhy složeny do jednoho (ekvivalence nad uzly tak, aby šlo)
  - potřebujeme umět dokázat, že nějaké pointery **ne**míří nikdy na stejný objekt – horní odhad dyn. analýzy
  - > hrany označeny místem, kde k tomu ukazování může dojít = **Kontext** (třeba i celý stack trace)
  - každá proměnná má svou skupinu, dyn. alok. objekty mají skupinu podle místa alokace – new

nebo se místo na haldě může vytvořit na zásobníku

- bezkontextová analýza
  - neřešíme ten kontext
    - → informace společná pro celý program

- kontextová analýza
  - points-to informace může být relativní vůči pozici v programu (IP), nebo s call stackem (aspoň částečným)

Andersenova metoda (bezkontextová ← nejpřesnější možná)

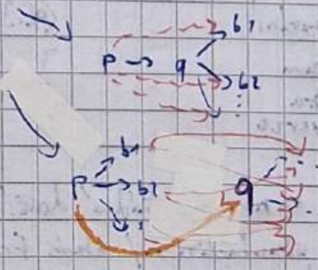
- u ∀ prom. p si pamatujeme množinu (σ) míst, kam může ukazovat
  - ⇒ soustava množinových nerovnic: ⇒ vyřeší se metodou nejm. pevného bodu

$p = \&x;$ (nebo dyn. alok.) ⇒ $\{x\} \subseteq \sigma(p)$ ← ①

$p = q;$ ⇒ $\sigma(q) \subseteq \sigma(p)$ ← ② a opakujeme, dokud se to nezastaví → nejm. pevný bod

$p = *q;$ ⇒ $\forall b \in \sigma(q): \sigma(b) \subseteq \sigma(p)$

$*p = q;$ ⇒ $\forall b \in \sigma(p): \sigma(q) \subseteq \sigma(b)$



ale mohou to být struktury a prvky mohou být ukazatele na další prvky... – ale my to máme celé v jednom

⇒ $\sigma(p, m)$:= místa, kam může ukazovat položka m objektu p

↳ $p.m = \&a;$

kvadratická složitost !! ☹

Steensgaardova metoda (bezkontextová, méně přesná, ale rychlejší)

$T(p)$ ... "typ" proměnné p, na začátku $\forall p$: $T(p)$ navzájem různé, přiřazení ⇒ ztotožnění typů (ROVNOST (x Andersen – nerovnost)) ⇒ třídy ekvivalence

- zase můžeme rozlišovat ty položky ($T(q, m)$ + pravidlo $T(x) == T(y) \Rightarrow \forall m: T(x, m) = T(y, m)$)

Částečné vyhodnocení

- konst. podvýrazy, sparse conditional const. prop. – sledování, kdy jsme do proměnné uložili konstantu
  - ↳ z různých míst, kudy tam přijdeme, to mohou být různé konstanty nebo runtimové hodnoty

Algebraické úpravy

- machine idioms – použití instrukcí, které udělají víc věcí najednou (např. FMA ↔ musíme spojit víc operací do jedné)
- odstranění zbytečných kontrol mezí
- distributivita ... atd.
- conditional move → ušetření podm. skoků – ale můžeme něco dělat zbytečně!! (může pomoct profiling)

Odstranění redundance

- copy propagation – pamatujeme si, že proměnná je kopií jiné, napojeno na analýzu rozsahu platnosti prom.
- eliminace spol. podvýrazů – velmi dobře se detekuje v SSA mezikódu
  - lokální – uvnitř BB, třeba pomocí hashování!
  - globální – uvnitř procedury
  - } je na to potřeba pomocná proměnná → zátěž na registry → nemusí být vždy výhodné
- přesun invariantního kódu z cyklu – např. for(.. i < k.size() ..) ⟹ s = k.size() for(.. i < s ..)
- partial-redundancy elimination, lazy code motion
  - ↳ redundance, ale jen někdy (v ifu) ⟹ řešení např. duplikace BB s tím výpočtem a bez
  - ↳ něco spočítáme, až když je to potřeba, abychom to nedělali zbytečně (přesuneme to až do posledního okamžiku, kam to jde)
  - bariéry: • návraty z funkcí • podmínky v control-flow • ne-loop-invariant výpočty
  - rovnou eliminuje mrtvý kód

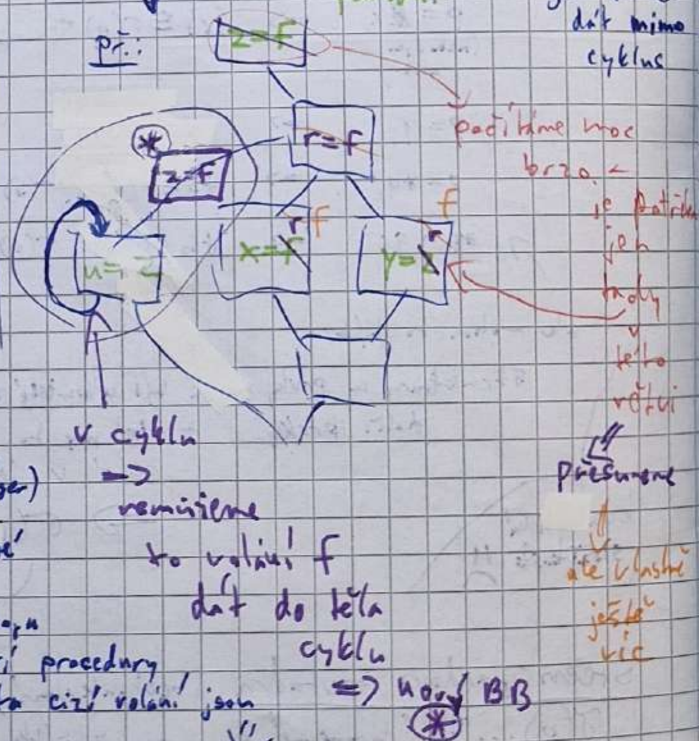
Odstranění neužitečného kódu

- mrtvý kód – něco dělá, ale zbytečně
- nedosaž. kód { nejčastěji v důsledku předchozích optimalizací
- optimalizace skoků – skoky na skoky ⟹ skok

nebo třeba

```
for (i ..) {    // sem je taky if(0<n)
    a[i][k]
}
```

→ programátor nemá hezké prostředky, jak tohle dát mimo cyklus



Integrace procedur (inline-expansion)

⟹ duplikace kódu
+ problémy v jazycích, kde je oddělený překlad od linkování
  ↳ musí se tam nechat i ta původní verze

např.

```
x = a+b
for(i..){
    p[i] = x+i
}
```

- listové procedury = nevolají žádné další
  - ⟹ není potřeba standardizovaný stack frame (třeba pro debugger)
    - ⟹ lze zjednodušit – např. vypustit ukládání některých registrů v prologu a obnovu v epilogu; nebudou tam nečekané výjimky
- shrink wrapping – ve větvích, které nevolají další procedury
  - ⟹ prolog a epilog dáme jen do větví, kde ta cizí volání jsou
- tail merging – u procedur se stejným koncem (detekce např. pomocí hashování)
  - ⟹ spojíme přes skoky ⟹ menší kód (ale problémy s debuggerem – neví, kudy se tam skočilo)

Specializace procedur

- když se procedura volá s konst. parametry
  - => pro ty varianty naklonujeme a přizpůsobíme je okolnostem (=> např. eliminace podmínek)
- nebo taky pro řešení aliasingu -> verze pro jo i ne
  - ↳ optimalizovanější verze
  - ↳ opatrnější verze
  - + if (jo) => jo() else => ne()
- nebo třeba F(x+1)
  - => F(x) a to +1 až někde v těle (může jít slepit s jiným +1)

Interproc. alokace reg.

- => úprava volací konvence - např. nemusíme ukládat nějaké registry
- oproti integraci procedur tu upravenou funkci můžeme volat z víc míst

Nápovědy pro procesor

- branch prediction
- prefetching do cache (ignoruje výjimky!! xx "hard load předem")
- instruction prefetching
  - procesory instrukce načítají po balících (např. 16B u Intelu)
    - => rychlejší (líp krmíme dekodéry)
  - => překladač se ty instrukce (BB) snaží na balíky zarovnat
    - ale tím se zase plýtvá kódová cache
    - => zarovnávají se třeba jen začátky BB v cyklech