Miskolci Egyetem . . . és Informatikai Kara

Végh János

Bevezetés a számítógépes rendszerekbe – programozóknak

A kód működés optimalizálása

V0.11@2015.02.16

Ez a segédlet a *Bevezetés a számítógépes rendszerekbe* tárgy tanulásához igyekszik segítséget nyújtani. Nagyrészt az irodalomjegyzékben felsorolt Bryant-O'Hallaron könyv részeinek fordítása, itt-ott kiegészítve és/vagy lerövidítve, néha más jó tankönyvek illeszkedő anyagaival kombinálva. A képzés elején, még a számítógépekkel való ismerkedés fázisában kerül sorra, amikor előbukkannak a különféle addig ismeretlen fogalmak, és megpróbál segíteni eligazodni azok között. Alapvetően a számítógépeket egyfajta rendszerként tekinti és olyan absztrakciókat vezet be, amelyek megkönnyítik a kezdeti megértést.

Ez az anyag még erőteljesen fejlesztés alatt van, akár hibákat, ismétléseket, következetlenségeket is tartalmazhat. Ha ilyet talál, jelezze a fenti címen. Az eredményes tanuláshoz szükség van az irodalomjegyzékben hivatkozott forrásokra, és az órai jegyzetekre, ottani magyarázatokra is.

Tartalomjegyzék


Optimize Performance

FEJEZET

1

A működés optimalizálása

Egy program írásának elsődleges célja, hogy az minden lehetséges esetben helyesen működjön. Egy gyorsan futó, de helytelen eredményt adó programnak nem sok értelme van. A programozóknak nem csak azért kell világos és tömör programokat írni, hogy azok számukra értelmesek legyenek, hanem azért is, hogy mások el tudják olvasni és megérteni, például kód áttekintés és későbbi módosítások alkalmával. Másrészt viszont

vannak olyan esetek is, amikor egy program gyorsítása is fontos szempont. Ha egy programnak video frameket vagy hálózati csomagokat kell valós időben feldolgozni, egy lassan futó programmal nem érünk célt. Amikor egy számítási feladat olyan mennyiségű számítást igényel, hogy napokig vagy hetekig tart annak a végrehajtása, akár 20% gyorsításnak is komoly hatása lehet. Ebben a fejezetben azt tárjuk fel, hogy tehetjük programjainkat gyorsabbá különféle optimalizálási eljárásokkal.

Hatékony program írásához sokféle típusú aktivitásra van szükség. Először is, ki kell választani a megfelelő algoritmusokat és adat szerkezeteket. Másodszor, olyan forráskódot kell írnunk, amelyet a fordítóprogram hatékonyan tud optimalizálni és abból hatékony végrehajtható kódot készíteni. Ehhez a második részhez fontos megértenünk az optimalizáló fordítóprogramok képességeit és korlátait. A program írásakor látszólagosan kis változtatásokkal is jelentős eltérést érünk el abban, milyen jól tudja a fordítóprogram azt optimalizálni. Néhány programnyelv könnyebben optimalizálható, mint a többi. A C néhány olyan tulajdonsága, mint a pointer aritmetikára és típuskényszerítésre való képesség, komoly kihívást jelent egy optimalizáló fordítóprogram számára. A programozók viszont gyakran meg tudják írni programjaikat úgy is, hogy abból a fordítóprogramok könnyebben tudjanak hatékony kódot előállítani. Egy har-

madik technika a különösen számítás igényes feladatok megoldására, hogy a feladatot olyan részekre osztjuk, amelyeket párhuzamosan is lehet számolni, több processzor vagy több mag valamilyen kombinációjával. Még ha használjuk is a párhuzamosságot, akkor is fontos, hogy a párhuzamos szálak maximális hatékonysággal fussanak, így a fejezet anyaga mindenképpen releváns.

Program fejlesztésről és optimalizálásról beszélve, mindenképpen tekintetbe kell vennünk, hogyan fogják a kódot használni és milyen kritikus faktorok befolyásolják azt. Általában véve, a programfejlesztőnek kompromisszumot kell kötnie, hogy milyen könnyű a programot implementálni és karbantartani, valamint hogy az milyen gyorsan fut. Algoritmikus szinten, egy egyszerű beszúrásos rendezés percek alatt programozható, egy nagyon hatékony rendező rutin elkészítése és optimalizálása egy vagy több napot is igénybe vehet. A kódolási szinten, az alacsony szintű optimalizálások rontják a kód olvashatóságát és modularitását, növelik a hibák lehetőségét és megnehezítik a hibakeresést és a bővítést. Olyan kódok esetén, amelyeket sokszor és teljesítménykritikus környezetben használnak, a kiterjedt optimalizálás nagyon is helyén való lehet. Ilyenkor komoly kihívást jelent bizonyos fokú eleganciát és olvashatóságot megőrizni, a komoly átalakítások ellenére.

Több, a kód hatékonyságot javító technikát is bemutatunk. Ideális esetben a fordítóprogram bármilyen, általunk írott kódot elfogad, és abból a lehető leghatékonyabb, a megadott módon viselkedő gépi kódú programot állítja elő. A modern fordítóprogramok kifinomult analizálási és optimalizálási formákat alkalmaznak, és folyamatosan javulnak. Azonban, még a legjobb fordítóprogramokat is akadályozzák az optimalizálás blokkolók, a program viselkedésének olyan vonatkozásai, amelyek erősen függenek a végrehajtási környezettől. A programozóknak olyan kód írásával kell segíteni a fordítóprogramot, hogy az könnyen optimalizálható legyen.

Egy program optimalizálásakor az első lépés a felesleges munka kiküszöbölése, ami lehetővé teszi a kód számára, hogy a szándékolt feladatot a lehető leghatékonyabban oldja meg. Ez tartalmazza a szükségtelen függvényhívások, feltétel vizsgálatok és memória hivatkozások kiiktatását. Ezek az optimalizálások nem függenek a cél számítógép specifikus tulajdonságaitól.

Egy program hatékonyságának maximalizálásához a programozónak és a fordítóprogramnak egyaránt szüksége van a cél számítógép egyfajta modelljére, ami megadja, hogy az utasítások hogyan hajtódnak végre és a műveleteknek milyen idő jellemzőik vannak. Például, a fordítóprogramnak ismernie kell az utasítások végrehajtási idejét, hogy el

tudja dönteni, hogy szorzó utasítást vagy eltolások és összeadások megfelelő kombinációját érdemes használni. A modern számítógépek kifinomult technikákat használnak a gépi kódú programok végrehajtására, sok utasítást párhuzamosan, esetleg azokat a programban való megjelenésétől eltérő sorrendben hajtva végre. A programozóknak ismerniük kell ezeknek a processzoroknak a működését, hogy programjaikat maximális sebességűre tudják hangolni. Főként az Intel és az AMD processzorok tervei alapján, bemutatunk egy magas-szintű számítógép modellt. Javaslunk az utasítások processzoron való végrehajtására egy grafikus adatfolyam (dataflow) ábrázolási módot, amelynek használatával megjósolhatjuk a program hatékonyságát.

A processzor működésének ilyen megértése után megtehetjük második lépésünket programunk optimalizálására, a processzor ún. utasítás szintű párhuzamos végrehajtásra való képességének kihasználásával, amikor is több utasítást hajt végre egyidejűleg. Számos program átalakítást hajtunk végre, amelyek csökkentik a számítás különböző részei közötti adat függőségeket és növelik a párhuzamosság elérhető mértékét.

A fejezetet nagy programok optimalizálásával kapcsolatos problémák tárgyalásával zárjuk. Ismertetjük az ún. kód profil meghatározókat, amely eszközök a program különböző részeinek hatékonyságát mérik. Az ilyen vizsgálatok segítenek a kód kevéssé

hatékony részeinek megtalálásában és felhívják figyelmünket a kód azon részeire, amelyekre érdemes optimalizálási erőfeszítéseinket koncentrálni. Végezetül bemutatuk egy fontos megfigyelést, amit Amdahl-törvényként ismernek, ami mennyiségileg is megfogalmazza a rendszer valamely részének optimalizálása által kifejtett hatást.

A fejezetben az optimalizálást úgy mutatjuk be, mint egy egyszerű lineáris folyamatot, amelyik során meghatározott sorrendben egyszerű lineáris transzformációkat alkalmazunk a kódra. Valójában azonban a feladat nem ennyire egyszerű, sok "próbaszerencse" kísérletezés is szükséges hozzá. Ez egyre inkább igazzá válik a későbbi optimalizálási fázisok felé haladva, amikor is látszólagosan kis változások a hatékonyság nagyon nagy változását eredményezik, az ígéretes technikák meg hatástalannak bizonyulnak. Mint azt a példákban látni fogjuk, elég nehéz megmondani, hogy egy bizonyos kód sorozatnak miért annyi a végrehajtási ideje. A hatékonyság a processzor számos olyan megvalósítási részletétől függ, amelyről nagyon kevés dokumentáció áll rendelkezésre és amelynek működését csak kevésssé értjük. Ez a másik oka, hogy a technikák számos variációját és azok kombinációját is kipróbáljuk.

Egy program assembly ábrázolásának tanulmányozása az egyik leghatékonyabb eszköz arra, hogy egy fordítóprogram működését és/vagy a generált kód futását

megértsük. Jó stratégia, ha a ciklusok belsejének tanulmányozásával kezdjük a kód vizsgálatát, és megkeressük az olyan hatékonyság csökkentő tényezőket, mint a túlzott memória hivatkozások vagy a regiszterek csekély használata. Az assembly kódból kiindulva, előre jelezhetjük, milyen utasításokat lehet párhuzamosan elvégezni, és azok mennyire jól használják a processzor erőforrásait. Mint látni fogjuk, gyakran meg tudjuk határozni egy ciklus végrehajtási idejét (vagy legalább egy alsó korlátot tudunk adni rá) a kritikus utak meghatározásával, a ciklusok ismételt végrehajtásakor keletkező az adatfüggőségi láncok felderítésével. Ezután visszatérhetünk a forráskódhoz, és annak módosításával a fordítóprogramot hatékonyabb implementáció készítése felé irányíthatjuk.

A legtöbb nagy fordítóprogramot, beleértve a **gcc**-t is, folyamatosan fejlesztik és javítják, különösen azok optimalizálási lehetőségeit. Nagyon jól bevált stratégia, hogy csak a szükséges legkisebb mértékben írjuk át a programot annak eléréséhez, hogy a fordítóprogram hatékony kódot tudjon generálni. Ilyen módon el tudjuk kerülni az olvashatóság, a modularitás és a hordozhatóság elrontását, mintha csak egy nagyon rossz képességű fordítóprogrammal dolgoznánk. Itt ismét csak iteratív módon módosítjuk a kódot és elemzzük annak hatékonyságát, mérésekkel vagy az assembly

kód analízisével.

Kezdő programozók számára furcsának tűnhet, hogy a forráskódot módosítjuk azzal a céllal, hogy a fordítóprogram hatékonyabb kódot állítson elő, de ténylegesen így készítenek nagy hatékonyságú programokat. Az assembly nyelven való programozáshoz képest ez a közvetett megközelítési mód azzal az előnnyel jár, hogy az eredményül kapott kód más számítógépeken is futni fog, bár nem a csúcs hatásfokkal.

1.1. A fordítóprogramok lehetőségei és korlátai

A modern fordítóprogramok kifinomult algoritmusokat használnak annak felderítésére, hogy a programban milyen értékeket számolunk ki és azokat hogyan használjuk. Ezek kihasználják a kifejezések egyszerűsítésének lehetőségeit, egyszer elvégzett számítás több helyen való felhasználását, és hogy egy adott számítást kevesebb alkalommal kelljen elvégezni. A legtöbb fordítóprogram (a `gcc` is) lehetőséget biztosít a felhasználónak annak szabályozására, hogy milyen optimalizálási lehetőségeket használjon. A legegyszerűbb szabályozás az optimalizálási szint kiválasztása. Például, a **gcc** fordítót a **-O1** kapcsolóval meghívva, az csak az alap szintű optimalizálást használja. A **gcc** fordítót a **-O2** vagy **-O3** kapcsolóval használva, az sokkal kiterjedtebb optimalizálást végez. Ezek javítják ugyan a program hatékonyságát, de növelik a program méretét és megnehezítik a hibák felderítését a szokásos hibakereső eszközökkel. Az alábbiakban főként az 1 szintű optimalizálással végzett fordításokat tárgyaljuk, bár a **gcc** felhasználói körében a 2. optimalizálási szint használata az elfogadott. Szándékosan korlátozzuk az optimalizálás szintjét annak bemutatása érdekében, hogy a C nyelven egy függvény megírásának különböző módjai hogyan befolyásolják a fordítóprogram által generált

kód hatékonyságát. Látni fogjuk, hogy tudunk olyan C kódot írni, amelyet csak 1. optimalizálási szinttel fordítunk és mégis jelentősen túlszárnyalja azt a naivabb módon megírt változatot, amelyet a legmagasabb optimalizálási szinttel fordítunk le.

A fordítóprogramoknak figyelniük kell arra, hogy csak biztonságos optimalizálásokat alkalmazzanak a programokban, úgy értve, hogy az eredményül kapott program pontosan úgy viselkedjék az összes elképzelhető helyzetben, mint a nem optimalizált változat, a C nyelvi szabvány által biztosított határok között. Ha biztosítjuk azt, hogy a fordítóprogram csak biztonságos optimalizálást végezzen, kiküszöböljük a nemkívánatos futási időbeli viselkedést, de ez azt is jelenti, hogy a programozónak több erőt kell kifejteni olyan irányban, hogy úgy írja meg a programot, amit aztán a fordítóprogram hatékony gépi kóddá tud alakítani. Hogy megértsük azt a kihívást, amit annak eldöntése jelent, hogy egy átalakítás biztonságos-e vagy sem, tekintsük a 1.1 listán látható két eljárást.

Első pillantásra a két eljárás azonos viselkedést mutat: az **xp** mutató által kijelölt helyen levő értékhez hozzáadja az **yp** mutató által kijelölt helyen tárolt érték kétszeresét. Második pillantásra látjuk, hogy a **twiddle2** függvény hatékonyabb: csak három memória hivatkozásra van szüksége (***xp** olvasás, ***yp** olvasás, ***xp** írás), míg a **twiddle1**

Programlista 1.1: Példa a "memória álnév" optimalizálási problémára

```
void twiddle1(int *xp, int *yp)
{
   *xp += *yp;
   *xp += *yp;
}

void twiddle2(int *xp, int *yp)
{
   *xp += 2* *yp;
}
```

függvénynek hatra (***xp** olvasás kétszer, ***yp** olvasás kétszer, és ***xp** írás kétszer) Azaz, azt hihetnénk, hogy ha a fordítóprogramnak a **twiddle1** eljárást kell lefordítani, akkor hatékonyabb kódot generálhat, ha **twiddle2** módon végzi a számítást.

Nézzük meg, mi történik, ha **xp** és **yp** egyenlők. Ebben az esetben a **twiddle1** a következő számítást végzi:

```
*xp += *xp;
```

ami megkétszerezi az **xp** által kijelölt helyen levő értéket. Mivel kétszer hívjuk, az érték az eredeti négyszeresére változik. A **twiddle2** a következő számítást végzi:

```
*xp += 2* *xp;
```

azaz az érték a háromszorosára növekszik. A fordítóprogram nem tudhatja, hogyan fogjuk **twiddle1** függvényt hívni, azaz fel kell tételeznie, hogy **xp** és **yp** egyenlők is lehetnek. Ezért nem generálhatja a **twiddle2** stílusú optimalizált kódot.

Ez az eset, amikor két mutató ugyanazt a memóriahelyet jelöli, memória álnév (memory aliasing) néven ismeretes. Amikor biztonságosan akar optimalizálni, a fordítóprogramnak azt kell feltételezni, hogy a különböző pointerek memória álnevek is lehetnek.

Másik példaként vegyünk egy olyan programot, amelyben szerepelnek a `p` és `q` program változók, és tekintsük a következő kódrészletet:

```
x = 1000; y = 3000;
*q = y; /*3000 */
*p = x; /*1000 */
t1 = *q; /*1000 or 3000 */
```

A `t1` kiszámított értéke attól függ, hogy `p` és `q` egymás álnevei vagy sem: ha nem, akkor az érték 3000, ha igen, akkor 1000. Ez elvezet bennünket az egyik fő optimalizálás gátlóhoz, amik olyan program tulajdonságok, amelyek nagyon erősen korlátozzák a fordítóprogramot optimális kód előállításában. Ha egy fordítóprogram nem tudja kitalálni, hogy két mutató egymás álnevei vagy sem, azt kell feltételezni, hogy bármelyik

eset lehetséges, ami korlátozza az optimalizálási lehetőségek számát.

Egy másik gátló tényező lehet a függvényhívás. Példaként tekintsük a következő két eljárást:

```
int f();
int func1(){
return f()+ f()+ f() + f();
}

int func2(){
return 4*f();
}
```

Első pillantásra úgy tűnik, a két eljárás ugyanazt számolja, de **func2** csak egyszer hívja **f**-et, míg **func1** négyszer. Akár a **func2** stílusában is generálhatnánk kódot, amikor **func1**-et

kell lefordítani. Tekintsük azonban a következő **f** kódot:

```
int counter = 0;
int f(){
return counter++;
}
```

Ennek a függvénynek van egy mellékhatása: módosítja a program globális állapotát. Megváltoztatva, hogy hányszor hívjuk a függvényt, változik a program viselkedése. Nevezetesen, **func1** visszatérési értéke **0 + 1 + 2 + 3 = 6** lenne, míg **func2** visszatérési értéke **4 . 0 = 0**, feltéve, hogy mindkét esetben a `counter` globális változó kezdetben 0 értékű volt. A legtöbb fordítóprogram nem próbálja kitalálni, hogy nincs-e a függvénynek mellékhatása, és így lehet-e alkalmazni a **func2** stílusú optimalizálást. Ehelyett, a fordító program felkészül a legrosszabb esetre és változatlanul hagyja a függvény hívást.

A **GCC** jó, de nem kivételes fordító, ami optimalizálási képességeit illeti. Elvégzi az alapvető optimalizálásokat, de nem végez olyan radikális program átalakításokat, mint amilyeneket "agresszívebb" fordítók. Ennek következtében, a **GCC** felhasználói több erőfeszítést fordítanak arra, hogy olyan módon írják meg programjaikat, amelyik

egyszerűbbé teszi a fordító számára hatékony kód előállítását.

1.2. A program hatékonyságának kifejezése

Bevezetjük a ciklus per elem metrikát (cycles per element, CPE), amivel kifejezhetjük a program hatékonyságát, és ami később kalauzul szolgál a kód javításakor. A CPE mérések segítenek annak részletes megértésében, hogy mennyire hatékony egy iteratív program program ciklusának működése. Ez nagyon hasznos olyan programok esetén, amelyek valamilyen ismétlődő számítást végeznek, például egy képben pixeleket dolgoznak fel vagy egy mátrix szorzás elemeit számolják.

Egy processzor tevékenységeinek sorba rendezését egy olyan óra vezérli, amely meghatározott frekvenciájú jelet szolgáltat, tipikusan gigahertz-ben (GHz) kifejezve. Például, amikor egy rendszerről azt írják, hogy az egy "4 GHz" processzorral rendelkezik, az azt jelenti, hogy a processzor másodpercenként 4.0×10^9 ciklust hajt végre. Az egy órajel végrehajtásához szükséges időt az órajel frekvencia reciproka adja meg. Ezt tipikusan nanoszekundum egységben adják meg (1 nanoszekundum 10^{-9} szekundum), vagy pikoszekundumban (1 pikoszekundum 10^{-12} szekundum). Például, egy 4 GHz frekvenciájú óra periódusideje 0.25 nanoszekundum vagy 250 pikoszekundum. Egy programozó szempontjából sokkal intuitívebb a mérés eredményét órajel ciklusokban

kifejezni, mint nanoszekundumban vagy pikoszekundumban. Ilyen módon a mérés azt fejezi ki, hány utasítást kell végrehajtani, és nem azt, hogy milyen gyorsan fut az óra.

Programlista 1.2: Példa a program hatékonyságának kifejezésére

```
/* Compute prefix sum of vector a */
void psum1(float a[],float p[],long int n)
{
   long int i;
  p[0] = a[0];
  for (i = 1; i < n; i++)
     p[i] = p[i-1] + a[i];
}

void psum2(float a[], float p[], long int n)
{
  long int i;
```

Sok eljárás tartalmaz olyan ciklust, amely egy elemkészlet felett iterál. Az

$$\vec{a} =< a_0, a_1, \ldots, a_{n-1} >$$

vektor esetén a megelőző elemek összegének (prefix sum)

$$\vec{p} =< p_0, p_1, \ldots, p_{n-1} >$$

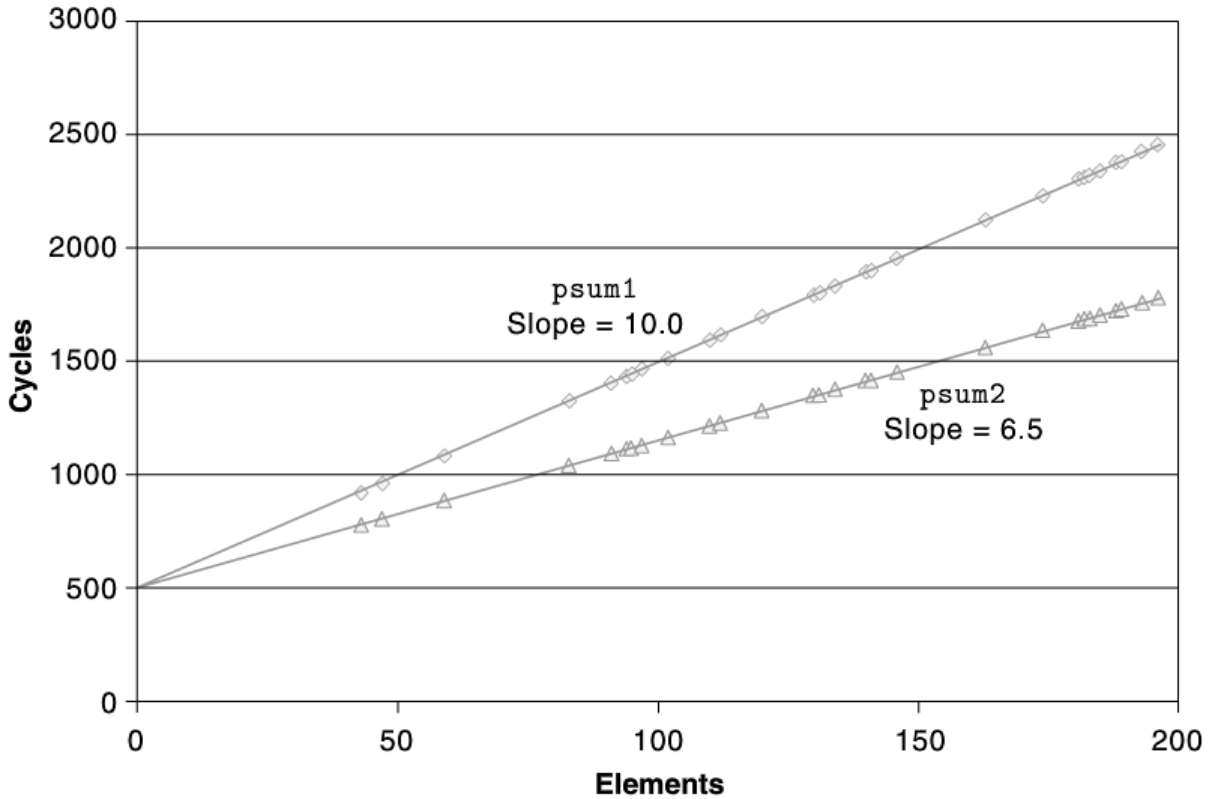
definíciója

$$p_0 = a_0$$

$$p_i = p_{i-1} + a_i; 1 \leq i < n$$

A **psum1** és **psum2** függvények mindegyike az n hosszúságú vektor megelőző elemeinek összegét számítja ki. A **psum1** iterációnként egy eredmény elemet számít ki. A **psum2** függvény a ciklus kisimítás (loop unrolling) néven ismert technikát használja arra, hogy iterációnként két elemet számítson ki. Meg fogjuk látni az utóbbi módszer előnyeit.

Az ilyen eljárások végrehajtásához szükséges időt egy konstans és a feldolgozott elemek számával arányos tényező adja meg. Az 1.1 ábra azt mutatja, hogyan függ a két függvény esetén az n elemszámtól az órajel ciklusok száma. Egy legkisebb négyzetek módszerű illesztést használva, azt találjuk, hogy a futási idők (órajel ciklusokban mérve) a **psum1** függvény esetén a $496 + 10.0n$, a **psum2** esetén pedig az $500 + 6.5n$ egyenlettel közelíthetők. Ezek az egyenletek egy 496 és 500 órajel ciklus közötti közötti többlet időt jeleznek, ami az időzítő kód végrehajtása továbbá a ciklus és az eljárás inicializálása érdekében szükségesek; valamint egy elemenként 6.5 és 10 ciklus közötti többletet. Nagy (mondjuk 200 fölötti) elemszám esetén a futási időt a lineáris tag határozza meg. Ezeket az együtthatókat tekintjük a tényleges elemenkénti

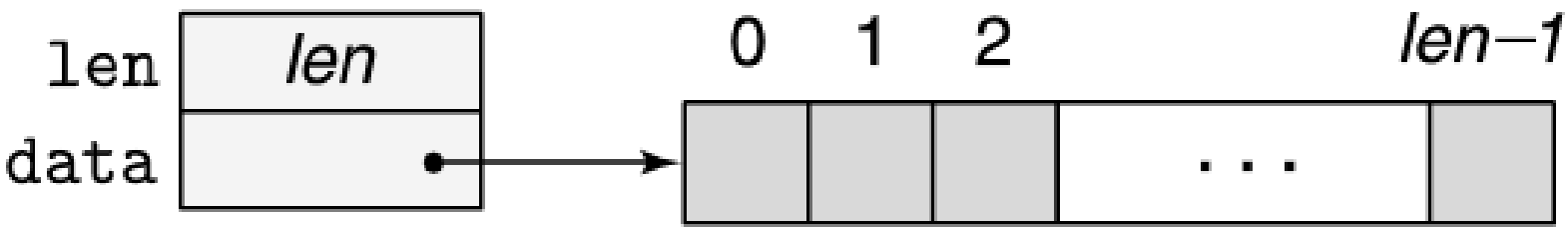


1.1. ábra. A prefix-sum függvények hatékonysága. Az egyenesek meredeksége az elemenkénti órajel ciklus számot (CPE) adja meg.

ciklusszámnak (CPE). Az iterációnkénti ciklus szám helyett az elemenkénti ciklus számot használjuk, mivel a ciklus mag kisimítás és a hasonló technikák kevesebb iterációt használnak, és végső célunk annak meghatározása, hogy egy adott vektor hossz esetén milyen gyorsan fut az eljárás. Arra törekszünk, hogy a számításokra a CPE értéket minimalizáljuk. Ilyen értelemben a **psum2** 6.50 CPE értéke jobb, mint a **psum1** 10.0 CPE értéke.

1.3. A példa program

Nézzük meg egy példán, hogy egy absztrakt vektort kezelő programot hogyan alakíthatunk át szisztematikusan hatékonyabb kóddá. Egy vektort a memória két blokkjában tárolunk: van egy fejzet és maga az adattömb. A fejzet szerkezete a következő, lásd 1.2 ábra és 1.3 programlista:



1.2. ábra. A 'vektor' absztrakt adattípus. A vektor egy header információ és megadott hosszúságú tömb ábrázolja.

A deklaráció a **data_t** adattípust használja az elemek tárolására. Esetünkben a hatékonyságot egész (**integer, C int**), egyszeres pontosságú lebegőpontos (**single-precision**

Programlista 1.3: A 'vektor' absztrakt adattípus fejzete

```
/* Create abstract data type for vector */
typedef struct {
   long int len;
   data_t *data;
} vec_rec, *vec_ptr;
```

floating-point, C float) és kétszeres pontosságú (**double-precision floating-point, C double**) értékekkel mérjük. Ehhez lefordítjuk és futtatjuk a kódot, ilyesféle típus deklarációkkal:

```
typedef int data_t;
```

A vektor elemek tárolására az adatblokkot úgy foglaljuk le, hogy az egy **len** számú objektumból álló **data_t** típusú tömb legyen.

Az 1.4 és az 1.5 programlisták mutatnak a vektor előállításra vonatkozóan pár alap eljárást, amelyek lehetővé teszik a vektor elemek elérését, és megadják a vektor hosszát. Fontos megjegyzés, hogy **get_vec_element** minden egyes vektor hivatkozás esetén ellenőrzi a határokat. Ez a kód hasonló ahhoz, amit több nyelv (pl. Java) használ tömbök megvalósítására. Az index túllépés vizsgálata csökkenti a programhiba lehetőségét, de egyúttal lassítja a végrehajtást.

A továbbiakban a kódon átalakítások sorozatát fogjuk végezni, a kombináló függ-

Programlista 1.4: **Az absztrakt vektor típus implementációja, a new_vec eljárás.** Mostani programunkban az adat típusa int; float vagy double lehet.

```
/* Create vector of specified length */
vec_ptr new_vec(long int len)
{
  /* Allocate header structure */
   vec_ptr result = (vec_ptr) malloc(sizeof(vec_rec));
   if (!result)
    return NULL;  /* Couldn't allocate storage */
  result->len = len;
   /* Allocate array */
   if (len > 0) {
     data_t *data = (data_t *)calloc(len, sizeof(data_t));
     if (!data) {
       free((void *) result);
    return NULL;  /* Couldn't allocate storage */
     }
     result->data = data;
   }
   else
     result->data = NULL;
   return result;
 }
```

vény különböző változatait hozzuk létre. A folyamat jellemzésére egy Intel Core i7 processzort használó számítógépen (referencia számítógép) megmérjük a függvények CPE hatékonyságát. Ezek a mérések azt jellemzik, hogyan futnak egy bizonyos számí-

Programlista 1.5: **Az absztrakt vektor típus implementációja, a get_vec_element eljárás.** Mostani programunkban az adat típusa int; float vagy double lehet.

```
/* Retrieve vector element and store at dest.
 * Return 0 (out of bounds) or 1 (successful) */

int get_vec_element(vec_ptr v, long int index, data_t *dest)
{
   if (index < 0 || index >= v->len)
     return 0;
   *dest = v->data[index];
   return 1;
}

/* Return length of vector */
long int vec_length(vec_ptr v)
{
  return v->len;
}
```

tógépen, azaz semmi garancia nincs arra, hogy másik számítógép vagy fordítóprogram használatával hasonló hatékonyságot kapunk. Ennek ellenére, különféle fordítóprogram/processzor kombinációkat használva, egészen összehasonlítható eredményeket kapunk.

Amikor sorra vesszük az átalakításokat, azt találjuk, hogy azok némelyike csak mi-

nimális hatékonyság javulást eredményez, míg mások drámai hatással járnak. Annak kiválasztása, hogy milyen átalakítás kombinációt használjunk, már valóban a gyors kód írásának "fekete mágiája" témakörébe tartozik. A mérhető javulást nem eredményező kombinációk némelyike valóban hatástalan, mások viszont azért fontosak, mert a fordítóprogram számára eredményeznek további optimalizálási lehetőséget. Tapasztalatunk szerint a legjobb eredményt a kísérletezés és az analízis együttese adja: ismételten ki kell próbálni különféle megközelítéseket, hatékonysági méréseket és assembly kódbeli ábrázolások vizsgálatát, hogy azonosítani tudjuk a hatékonyságbeli szűk keresztmetszeteket.

Kiinduló pontként vegyük a **combine1** CPE méréseket a referencia számítógépen, különféle adat és művelet típusokat kombinálva. Egyszeres és kétszeres pontosságú lebegőpontos adatok esetén az ezen a számítógépen végzett mérések azonos hatékonysági eredményt adtak összeadások esetén, de különbözőt szorzás esetén. Ezért öt különböző CPE értéket adunk meg: egészek összeadása és szorzása, lebegőpontos összeadás, egyszeres pontosságú szorzás ("F *" címkével) és kétszeres pontosságú szorzás (D *) címkével, lásd 1.4 táblázat.

Mint látjuk, méréseink kicsit pontatlanok. Az egész összeg és szorzat CPE értéke

Programlista 1.6: **Az absztrakt vektor típus implementációja.** Mostani programunkban az adat típusa `int`, `float` vagy `double` lehet.

```
/* Create vector of specified length */
vec_ptr new_vec(long int len)
{
  /* Allocate header structure */
   vec_ptr result = (vec_ptr) malloc(sizeof(vec_rec));
   if (!result)
    return NULL;  /* Couldn't allocate storage */
  result->len = len;
   /* Allocate array */
   if (len > 0) {
     data_t *data = (data_t *)calloc(len, sizeof(data_t));
     if (!data) {
       free((void *) result);
    return NULL;  /* Couldn't allocate storage */
     }
     result->data = data;
   }
   else
     result->data = NULL;
   return result;
 }
```

valószínűleg 29.00 lenne, nem pedig 29.02 vagy 29.21. A számok szépítgetése helyett, az elvégzett mérések tényleges eredményeit tüntettük fel. Sok tényező van, ami egy bizonyos kódsorozat végrehajtásához szükséges órajelek számának pontos mérését

befolyásolja. Lélekben kerekítsük fel vagy le ezeket a számokat néhány század órajellel.

Az optimalizálás nélküli kód közvetlenül fordítódik C nyelvről gépi kódra, néha nyilvánvaló hiányosságokkal. A '**-O1**' utasítássori kapcsoló használatával engedélyezzük az alap-optimalizálásokat. Mint láthatjuk, ez jelentősen –több mint egy kettes szorzóval– javítja a program hatékonyságát, a programozó erőfeszítése nélkül. Általában véve, helyes gyakorlat legalább ezt az optimalizálási szintet engedélyezni. A későbbiekben ezt az optimalizálási szintet (esetleg e felettit) használunk programjaink generálásakor és mérésekor.

Optimalizálási példaként tekintsük a 1.7 listán bemutatott kódot, amely rutin valamilyen művelet felhasználásával a vektor valamennyi elemét egyetlen értékbe kombinálja. A fordítás idején megadott **IDENT** és **OP** konstansok változtatásával újrafordításkor a kód különféle műveleteket tud végezni az adatokon.

Programlista 1.7: **A kombináló művelet kezdeti megvalósítása.** Az IDENT és az OP különböző deklarációival különböző műveleteket mérhetünk meg.

```
/* Implementation with maximum use of data abstraction */
void combine1(vec_ptr v, data_t *dest)
{
   long int i;

   *dest = IDENT;
   for (i = 0; i < vec_length(v); i++) {
   data_t val;
   get_vec_element(v, i, &val);
   *dest = *dest OP val;
   }
}
```

1.1. táblázat. A **combine1** megvalósítás esetén mért időértékek különféle műveletek és adattípusok esetén.

Függvéı	Oldı	Módszer	+	*
combine1	310	Absztrakt, nem optimalizált	29.02	2
combine1	310	Absztrakt, -O1	12.00	1

Egész

1.4. Programhurkok hatékonyságának javítása

Vegyük észre, hogy az 1.7 listán bemutatott **combine1** függvény a **for** programhurok teszt feltételének vizsgálatakor használja a **vec_length** függvényt. Idézzük fel, hogy ezt a vizsgálatot minden egyes iteráció alkalmával el kell végezni. Másrészt viszont, a vektor hossza nem változik a programhurkon belül. Ezért kiszámíthatjuk egyszer a vektor hosszát és a későbbiekben azt használjuk a feltétel vizsgálatára. Az 1.8 lista mutatja a módosított **combine2** változatot, ami a végrehajtás elején hívja meg a **vec_length** függvényt, és annak eredményét a **length** helyi változóhoz rendeli. Ez a változtatás észrevehető hatással bír bizonyos adattípusok és műveletek esetén, és minimálissal mások esetén. Azonban, ez az átalakítás szükséges ahhoz, hogy a későbbiekben kipróbált átalakításokban ne legyen szűk keresztmetszet.

Az optimalizálásnak ez a fajtája a kód áthelyezés (code motion) néven ismert általános optimalizálási osztály egy eleme. Ez az osztály, amelynek elemei azonosítják azokat a kódrészeket, amelyeket a számítás többször is elvégez (például egy hurokban), de a számítás eredménye nem változik. Emiatt ezt a számítást a kód egy korábbi részére helyezhetjük át, ahol az nem számítódik ki olyan sokszor. Ebben az esetben a **vec_length**

Programlista 1.8: Programhurok feltételvizsgálatának javítása. Ha **veclength** hívását kivesszük a programhurok végrehajtásából; nem kell azt minden egyes iterációban végrehajtani.

```
/* Move call to vec_length out of loop */
void combine2(vec_ptr v, data_t *dest)
{
  long int i;
  long int length = vec_length(v);
  *dest = IDENT;
  for (i = 0; i < length; i++) {
    data_t val;
    get_vec_element(v, i, &val);
    *dest = *dest OP val;
  }
}
```

hívását a ciklus elé helyezhetjük át.

Az optimalizáló fordítóprogramok megkísérlik a kód áthelyezést. Sajnos, azonban nagyon óvatosak azonban olyan kód átalakításakor, amelyek megváltoztatják, hogy egy eljárást mikor és hogyan hívódik meg. Nem tudják megbízhatóan detektálni, hogy egy függvénynek vannak-e mellékhatásai, így azt tételezik fel, hogy lehetnek. Ha például a **vec_length** függvénynek lenne mellékhatása, akkor **combine1** és **combine2** másként

1.2. táblázat. A **combine2** megvalósítás (a **veclength** hívás felszámolása) esetén mért időértékek különféle műveletek és adattípusok esetén.

Függvéı	olda	Módszer	Egész +	*
combine1	310	Absztrakt -O1	12.00	1
combine2	323	**vec_length** áthelyezés	8.03	8

viselkedne. A kód hatékonyságának javításához a programozónak gyakran segítenie a kód explicit áthelyezésével.

Programlista 1.9: Az **strlen** függvény egy implementációja

```
/* Sample implementation of library function strlen */
/* Compute length of string */
size_t strlen(const char *s)
{
  int length = 0;

  while (*s != '\0') {
    s++;
    length++;
  }
  return length;
}
```

A programhurok alacsony hatékonyságának egy extrém példájaként tekintsük a **lower1** eljárást az 15 listán. Ez az eljárás hallgatók által hálózat programozási projekt keretén belül készített programból való. Az a célja, hogy a szövegben a nagybetű karaktereket kisbetűvé alakítsa. Az eljárás végig lépeget a sztringen, és minden nagybetűt kisbetűvé alakít, azaz az 'A' ... 'Z' tartományból az 'a' ... 'z' tartományba helyezi át azokat.

A kilépés vizsgálat részeként a **lower1** hurokban hívja a **strlen** könyvtári függvényt. Bár a **strlen** függvényt általában speciális x86 sztring kezelő utasításokkal implementálják, annak végrehajtása hasonló az 1.9 listán is bemutatott egyszerű verzióhoz. Mivel a C a

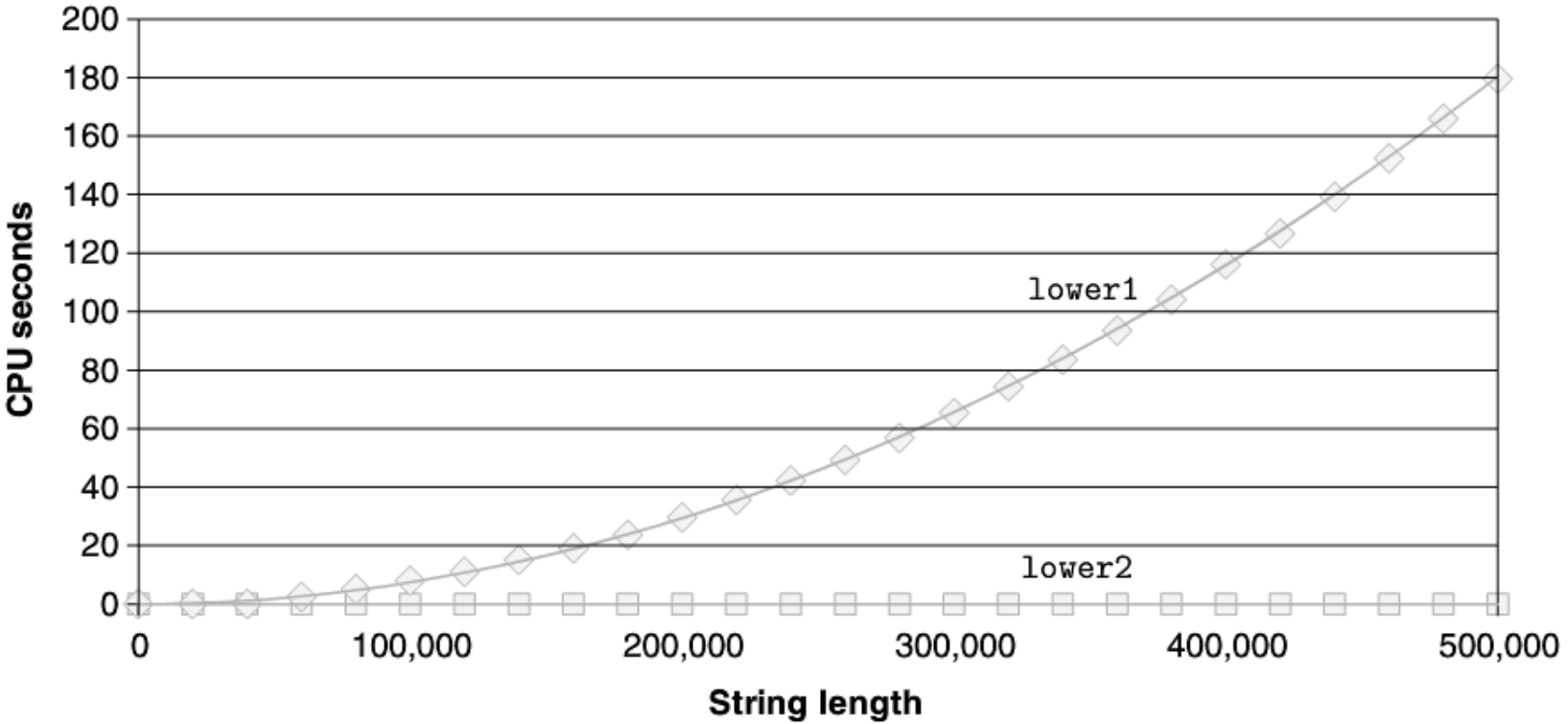
Programlista 1.10: A két kisbetűssé alakító rutin összehasonlítása

```
/* Convert string to lowercase: slow */
void lower1(char *s)
{
    int i;

    for (i = 0; i < strlen(s); i++)
    if (s[i] >= 'A' && s[i] <= 'Z')
     s[i] -= ('A' - 'a');
}
```

```
/* Convert string to lowercase: faster */
void lower2(char *s)
{
  int i;
  int len = strlen(s);
  for (i = 0; i < len; i++)

    if (s[i] >= 'A' && s[i] <= 'Z')
      s[i] -= ('A' - 'a');
}
```

sztringeket nullával határolt karakter sorozatként implementálja, **strlen** csak úgy tudja meghatározni a sztring hosszát, ha ellépeget a szöveg végén található null karakterhez. Egy n hosszúságú sztring esetén **strlen** végrehajtásának ideje n-nel arányos. Mivel **strlen** meghívódik **lower1** mind az n iterációjában, a teljes futási idő a sztring hosszának négyzetével (n^2) arányos.

Az előbbi analízist megerősítik a különböző hosszúságú sztringeken (a könyvtári **strlen** változat használatával) végzett mérések, mint az 1.3 ábrán látható. A **lower1** grafikonja meredeken emelkedik a sztring hosszával. Az 1.3 néhány további mérési adatot mutat,



1.3. ábra. A kisbetűvé alakító rutinok összehasonlítása. Az eredeti **lower1** kód futási ideje kvadratikus viselkedésű a nem-hatékony ciklus szerkezet miatt. A módosított **lower2** kód futási ideje lineáris viselkedésű.

1.3. táblázat. A kisbetűvé alakító rutinok futási idejének összehasonlítása különböző méretű sztringek esetén.

Függvé	16,384	32,768	65,536	13
lower1	0.19	0.77	3.08	12
lower2	0.0000	0.0000	0.0001	0.0

amelyekben a sztring hossza mindenütt 2 hatványa. Figyeljük meg, hogy **lower1** esetén a sztring hosszának megduplázása a futási idő négyszereződéséhez vezet. Ez valóban a négyzetes futási idő függést mutatja. Az 1,048,576 sztring hosszúság esetén a **lower1**

futási ideje már 13 perc CPU idő.

A **lower2** függvény megegyezik a **lower1** függvénnyel, kivéve, hogy **strlen** hívását a ciklusból kivittük. A hatékonyság drámaian javult. Az 1,048,576 sztring hosszúság esetén a függvény végrehajtásához csak 1.5 milliszekundum szükséges, több mint 500.000-szer gyorsabb, mint **lower1**. A sztring hosszának duplázása a futási idő duplázódásához vezet, a lineáris futási idő világos bizonyítékaként. Hosszabb sztringek esetén a futási idő javulása még szembetűnőbb.

Egy ideális világban a fordítóprogram észrevenné, hogy a ciklus kilépési vizsgálatában **strlen** ugyanazt az eredményt adja vissza, így azt ki lehet vinni a ciklusból. Ehhez azonban eléggé kifinomult analízis szükséges, mivel **strlen** a sztring elemeit vizsgálja és ezek az értékek megváltoztatják, hogy **lower1** hogyan halad tovább. A fordítóprogramnak azt kellene észrevennie, hogy bár a sztringen belül a karakterek változnak, egyik sem cserélődik ki nem-nulláról nullára vagy fordítva. Egy ilyen vizsgálat jóval meghaladja a legkifinomultabb fordítóprogramok képességeit is, így maguknak a programozóknak kell ez ilyen fajta átalakításokat elvégezni.

Ez a példa a program készítés egyik gyakori problémáját illusztrálja, amelyben egy látszólag triviális kód darab rejtett aszimptotikus hatékonysági problémát tartalmaz. Úgy

általában nem várnánk, hogy egy kisbetűvé alakító rutin a program hatékonyságának korlátja legyen. A programokat általában kis adat készletekkel teszteljük, amelyekre **lower1** hatékonysága megfelelő. Amikor viszont a végén a programot alkalmazzák, előfordulhat, hogy egymillió karaktert túllépő hosszúságú sztringekre is használják. Amikor hirtelen ez a kis ártalmatlan kód darab a lesz a legfontosabb hatékonysági szűk keresztmetszet. Ezzel szemben, **string2** alkalmas bármilyen hosszúságú szöveg kezelésére. Egy igazán hozzáértő programozó munkájának az is része, hogy elkerülje ilyen aszimptotikus hatékonysági problémák beépítését.

1.5. Az eljárás hívások számának csökkentése

Mint láttuk, az eljárás hívások komoly többlet munkát jelentenek, és a legtöbb optimalizálási formát még blokkolják is. Az 1.8 listán látjuk, hogy a **get_vec_element** minden egyes iterációban meghívódik. Ez a függvény viszont összehasonlítja az *i* indexet a ciklus hatásokkal minden egyes vektor hivatkozás esetén, szóval itt is van hatékonysági tartalék. A határok vizsgálata nagyon hasznos tulajdonság, amikor önkényes tömb hozzáféréssel dolgozunk, de a **combine2** egyszerű vizsgálata azt mutatja, hogy valamennyi hivatkozás érvényes lesz.

Tegyük fel, hogy absztrakt adattípusunkhoz hozzáadunk egy **get_vec_start** függvényt, ami az adattömb kezdőcímét adja vissza, lásd 1.11 lista. Ezután már megírhatjuk a **combine3** függvényt, ami nem tartalmaz a belső hurokban függvény hívásokat. Ahelyett, hogy az egyes elemeket függvény hívással venné elő, a tömböt közvetlenül éri el. A puristák mondhatják, hogy ez az átalakítás komolyan sérti a program modularitását. Elvben, az absztrakt vektor típus felhasználójának nem is kell tudnia, hogy a vektor elemeit egy tömbben tároljuk, mondjuk egy láncolt lista helyett. A pragmatikusabb programozó azt mondaná, hogy ez az átalakítás egy szükséges lépés a nagyobb

Programlista 1.11: **Függvényhívások kiküszöbölése a cikluson belül.** Az eredményül kapott kód sokkal gyorsabban fut, de kissé romlik a modularitása.

```
data_t *get_vec_start(vec_ptr v)
{
  return v->data;
}
/* Direct access to vector data */
void combine3(vec_ptr v, data_t *dest)
{
  long int i;
  long int length = vec_length(v);
  data_t *data = get_vec_start(v);
  *dest = IDENT;
  for (i = 0; i < length; i++) {
    *dest = *dest OP data[i];
  }
}
```

hatékonyság felé vezető úton.

A hatékonyság csak meglepően keveset javult, és azt is csak az egész összegzés esetén. Később azonban ez a hatékonyság vesztés lenne a szűk keresztmetszet a további optimalizálás során. Később még visszatérünk ehhez a függvényhez, hogy megértsük, **combine2**-ben az ismételt határ vizsgálat miért nem teszi még rosszabbá a hatékonyságot. Azokban az alkalmazásokban, amelyekben a hatékonyság fontos

1.4. táblázat. A combine3 megvalósítás esetén mért időértékek különféle műveletek és adattípusok esetén.

			Integer		F
Függvéı	Oldı	Módszer	+	*	+
combine3	33	**vec_length** áthelyezés	8.03	8.09	10
combin	42	Közvetlen adatelérés	6.01	8.01	10

szempont, gyakran kell kompromisszumot kötni a sebesség érdekében a modularitás és az absztrakció között. Érdemes dokumentálni a használt átalakításokat, meg az

ahhoz vezető megfontolásokat, ha később módosítani kellene a kódot.

1.6. Szükségtelen memória hivatkozások

A **combine3** a kombináló művelet által éppen számítás alatt levő értéket a **dest** mutató által kijelölt helyen gyűjti össze. Ezt a tulajdonságot a lefordított ciklusból generált assembly kód vizsgálatából láthatjuk. Alább mutatjuk a generált x86-64 kódot, amit **float** típus és szorzás művelet esetén kapunk

Programlista 1.12: **A combine3 kódbeli ciklus assembly változata.** Az eredményül kapott kód sokkal gyorsabban fut, de kissé romlik a modularitása.

```
  ;combine3:  data_t  =  float,  OP  =  *
  ;i  in  %rdx,  data  in  %rax,  dest  in  %rbp
.L498:   loop:
   movss   (%rbp), %xmm0        ;Read  product  from  dest
   mulss   (%rax,%rdx,4), %xmm0   ;Multiply  product  by  data[i]
   movss   %xmm0, (%rbp)        ;Store  product  at  dest
   addq   $1, %rdx              ;Increment  i
   cmpq   %rdx, %r12            ;Compare  i:limit
   jg   .L498                   ; If  >,  goto  loop
```

Ebben a ciklus kódban azt látjuk, hogy a **dest** mutatónak megfelelő regiszter az **%rbp** regiszterben található (az IA32-től eltérően, ahol **%ebp** a keret mutató szerepét tölti be, a 64-bites megfelelőjében **%rbp** bármilyen adatot tárolhat). Az **i**-edik iterációban a

program az értéket erre a helyre olvassa be, megszorozza **data[i]**-vel, majd az eredményt visszaírja a **dest** helyre. Ez az írás és olvasás felesleges, mivel aze egyes iterációk kezdetén beolvasott értéknek meg kell egyeznie az előző iteráció végén beírt értékkel.

Ezt a szükségtelen írás/olvasást kiküszöbölhetjük, ha a kódot a az 1.13 lista szerinti módon írjuk meg. Bevezetünk egy **acc** időleges változót, amit a ciklusban használunk a kiszámított érték összegyűjtésére. Az eredményt csak akkor tároljuk el a **dest** helyen, amikor a ciklus befejeződött. Ebben az esetben a fordítóprogram az `%xmm0` regisztert használhatja az értékek gyűjtésére.

A **combine3** ciklushoz képest, az iterációnkénti memória műveleteket két olvasásról és egy írásról egyetlen olvasásra csökkentettük, lásd 1.12 lista.

A program hatékonyságában jelentős javulást tapasztalunk, lásd 1.5 táblázat. Valamennyi érték legalább egy 2.4-szeres faktorral javult, az egész összeadás esetén elemenként két órajel ciklusra esett.

Ismét csak azt gondolhatnánk, hogy a fordítóprogramnak képesnek kellene lennie arra, hogy a **combine3** kódot, lásd 1.11 lista, úgy alakítsa át, hogy az az értéket egy regiszterben gyűjtse, amint azt a **combine4** is teszi, lásd 1.13 lista. Valójában azonban a két függvény a memória álnevek következtében különböző viselkedésű. Tekintsük

Programlista 1.13: **Az eredmény gyűjtése időleges változóban.** Ha az összegyűjtött eredményt az acc (az "accumulator" rövidítése) helyi változóban tartjuk akkor nem kell minden iterációban a memóriából elővenni és oda visszaírni.

```
/* Accumulate result in local variable */
void combine4(vec_ptr v, data_t *dest)
{
   long int i;
   long int length = vec_length(v);
   data_t *data = get_vec_start(v);
   data_t acc = IDENT;

   for (i = 0; i < length; i++) {
    acc = acc OP data[i];
   }
   *dest = acc;
}
```

például azt az esetet, amikor a művelet egészek szorzása és az azonossági elem 1. Legyen **v = [2, 3, 5]** egy három elemű vektor, és tekintsük a következő két függvény hívást:

```
combine3(v, get_vec_start(v)+ 2);
combine4(v, get_vec_start(v)+ 2);
```

Azaz, létrehozunk egy álnevet a vektor utolsó eleme és az eredmény tárolórekesze

Programlista 1.14: **Az eredmény gyűjtése időleges változóban.** A **combine4.c** assembly nyelvű változata.

```
;combine4:  data_t  =  float,  OP  =  *
;i  in  %rdx,  data  in  %rax,  limit  in  %rbp,  acc  in  %xmm0

.L488:   loop:

    mulss   (%rax,%rdx,4), %xmm0   ;Multiply  acc  by  data[i]
    addq    $1, %rdx   ;Increment  i
    cmpq    %rdx, %rbp   ;Compare  limit:i
    jg    .L488   If  >,   ;goto  loop
```

között. Ekkor a két függvény végrehajtásának eredménye, amint az 1.6 táblázatban látható:

Amint előzőleg mondtuk, **combine3** az eredményt a cél memóriahelyen gyűjti össze, ami ebben az esetben a vektor utolsó eleme. Kezdetben ezt az értéket ezért 1-re állítjuk, majd $2 * 1 = 2$, végül $3 * 2 = 6$. Az iteráció utolsó lépéseként ezt az értéket megszorozzuk önmagával, ami a 36 végeredményt adja. A **combine4** esetében a vektor egészen a végéig változatlan marad, amikor is az utolsó elem értékként a kiszámított $1 * 2 * 3 * 5 = 30$ eredményt kapja meg.

Természetesen a **combine3** és **combine4** közötti különbség bemutatására készített

1.5. táblázat. A **combine4** megvalósítás esetén mért időértékek különféle műveletek és adattípusok esetén.

			Integer		Floating	
Funkció	Old	Módszer	+	*	+	F *
combine3	42	Közvetlen adat elérés	6.01	8.01	10.01	11.
combine4	47	Összegyűjtés időleges változóban	2.00	3.00	3.00	4.0

példa eléggé mesterkélt. Mondhatjuk azt is, hogy **combine4** viselkedése közelebb áll a függvény leírásában szereplő szándékhoz. Sajnos, a fordítóprogram nem dönthet a

1.6. táblázat. A "memória álnév" hatása a **combine4** végrehajtására

Függvén	Kezdet	A ciklus előtt	i=0	i = 1	i = 2
combine3	[2,3,5]	[2,3,1]	[2,3,2]	[2,3,6]	[2,3
combine4	[2,3,5]	[2,3,5]	[2,3,5]	[2,3,5]	[2,3

függvény használatának feltételeiről, és nem tudhatja a programozó szándékait sem. Helyette, ha a **combine3** lefordítása a feladat, a konzervatív megközelítést választja, azaz írja és olvassa a memóriát, bár az kevésbé hatékony.

1.7. Értsük meg a modern processzorokat

Mindeddig olyan optimalizálást használtunk, aminek nem sok köze volt a tényleges cél-számítógéphez. Egyszerűen csökkentettük a felesleges eljáráshívások számát és kiküszöböltük az optimalizálást blokkoló tényezőket, amelyek nehézséget jelentenek az optimalizáló fordítóprogramoknak. Ha további hatékonyságjavítást is akarunk, akkor akkor a processzor mikroarchitektúra által használt optimalizálást is figyelembe kell vennünk, azaz azokat a rendszer tervezési elveket, amelyek alapján a processzor végrehajtja az utasításokat. Hogy a hatékonyság utolsó cseppjeit is kifacsarjuk, részletesen meg kell értenünk a program működését, éppúgy, mint a cél processzorra hangolt kód generálási módját. Alkalmazhatunk továbbá bizonyos alap optimalizálást, amelyik a processzorok egy nagy osztályán eredményez hatékonyság javulást. Lehet, hogy az előállított hatékonysági eredmények nem érvényesek más gépekre, de a működés és az optimalizálás általános elvei igen.

Hogy a hatékonyságot javítani tudjuk, legalább alapszinten meg kell értenünk a modern processzorok mikroarchitektúráját. A manapság egyetlen áramköri tokba integrált tranzisztorok száma lehetővé teszi, hogy a modern processzorok a hatékony-

ságot maximalizáló komplex hardvert alkalmazzanak. Ennek egyik eredménye, hogy tényleges működésük messze van attól, amit a gépi kódú programok vizsgálata alapján elképzelünk. A kód szintjén úgy tűnik, hogy az utasítások egymás után hajtódnak végre, és az egyes utasítások értéket olvasnak be regiszterből vagy a memóriából, elvégeznek egy műveletet, és elmentik az eredményt regiszterbe vagy a memóriába. Egy valódi processzorban több utasítás végrehajtás folyik egyidejűleg, ezt nevezik utasítás-szintű párhuzamosságnak (instruction-level parallelism). Bizonyos megvalósítások esetén akár 100 utasítás is lehet egyidejűleg folyamatban. Bonyolult mechanizmusokat használnak arra, hogy ez a párhuzamos végrehajtás úgy viselkedjék, ahogyan azt a gépi kódú program soros szemantikus modellje alapján elvárjuk. A modern processzorok egyik figyelemre méltó finomsága, hogy összetett és egzotikus mikroarchitektúrát használnak, amelyekben az utasítások párhuzamosan hajtódnak végre, míg kifelé az egyszerű soros utasítás végrehajtás látszatát keltik.

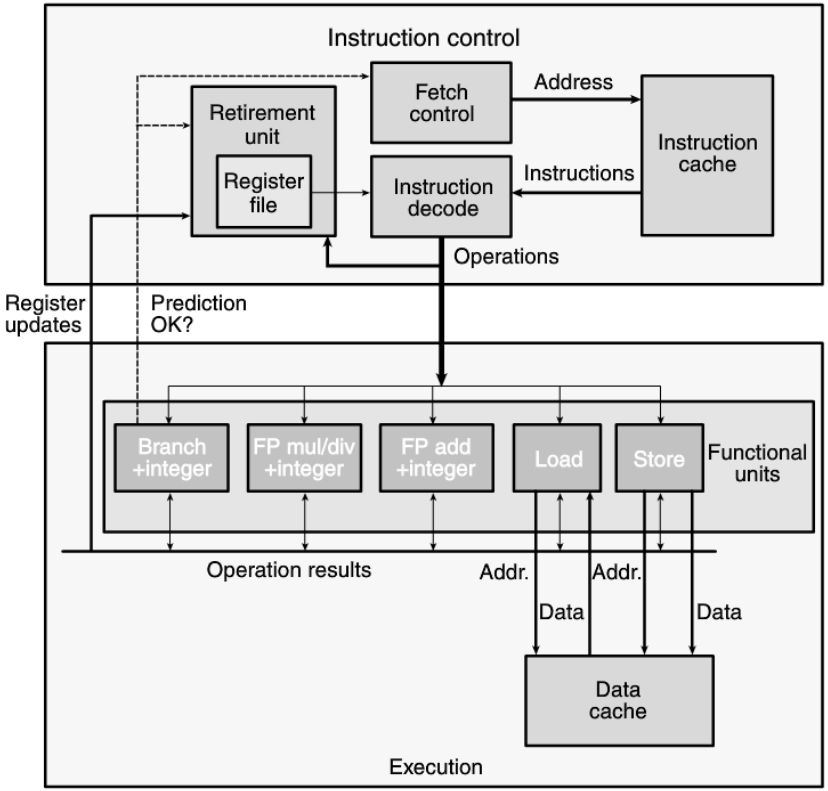
Bár a modern processzorok tervezésének részletei nem férnek bele a kurzus kereteibe, az általános működési elvek ismerete elegendő ahhoz, hogy megértsük, hogyan érik el az utasítás szintű párhuzamosságot. Látni fogjuk, hogy két korlát szabja meg a program maximális hatékonyságát. A látencia korlát (latency bound) akkor fordul elő, amikor

műveletek sorozatát szigorú sorrendben kell végrehajtani, mivel az egyik művelet eredményének a másik elkezdése előtt rendelkezésre kell állni. Ez a tényező határolhatja a hatékonyságot, amikor a kódban az adat függőségek nem teszik lehetővé az utasításszintű párhuzamosítás használatát. Az átbocsátó képesség (throughput bound) jellemzi a processzor funkcionális egységeinek számítási kapacitását. Ez utóbbi tényező lesz a program hatékonyságának végső korlátja.

1.7.1. Általános működés

Az 1.4 ábra egy modern mikroprocesszor erősen egyszerűsített blokkdiagramját mutatja. Hipotetikus processzorunk terve lazán emlékeztet az Intel Core I7 (Nehalem) processzorra. Ez a mikroarchitektúra jellemző a különböző gyártók által az 1990 vége óta gyártott nagyteljesítményű (high-end) processzoroknál. Ezt a szerkezetet illetik olyan jelzőkkel, mint szuperskaláris (superskalar), ami azt jelenti, hogy több műveletet tud végezni minden egyes órajel alatt, és a nem sorrendi (out-of-order), ami azt jelenti, hogy az utasítások végrehajtási sorrendje nem feltétlenül egyezik meg a gépi kódú utasítások sorrendjével. A legmagasabb szinten két fő részre oszlik a terv: az instruction control unit (ICU), ami a felelős az utasítások elővételéért a memóriából, valamint a program adatokon elemi műveletek elvégzéséért, és az execution unit (EU), ami végrehajtja ezeket a műveleteket. Az egyszerű sorrendi csővezetékezett processzorokhoz képest a nem-sorrendi végrehajtású processzorok sokkal nagyobb és összetettebb hardvert követelnek, de nagyobb fokú utasítás-szintű párhuzamosságot érnek el.

Az ICU az utasításokat egy utasítás gyorsítótárból (instruction cache) olvassák be, ami egy speciális nagy sebességű memória, ami az utoljára elért utasításokat tartalmazza.



1.4. ábra. Egy modern processzor blokk diagramja. Az utasítás vezérlő egység (instruction control unit) felelős az utasítások beolvasásáért a memóriából és a primitív műveletek sorozatának előállításáért. A végrehajtó egység (execution unit) végzi a műveleteket és jelzi, hogy az elágazások előrejelzése helyes volt-e.

Általában az ICU jó előre előveszi az utasításokat, így van elég ideje dekódolni azokat leküldeni a műveleteket az EU-ba. Az egyik probléma, hogy amikor a program elágazáshoz[1] ér, két lehetséges irányban haladhat tovább. Bekövetkezhet az ugró utasítás, ami új helyre adja át a vezérlést. A másik lehetőség, hogy nincs ugrás, azaz az utasítássorozat a következő utasítással folytatódik. A modern processzorok használnak egy elágazás előrejelzés (branch prediction) nevű technikát, amelyben megpróbálják kitalálni, hogy bekövetkezik-e az ugrás és megjósolják az új cél címet. A spekulatív végrehajtásnak (speculative execution) nevezett technika használatával a processzor elkezdi az utasításokat elővenni és dekódolni azon a helyen, amelyen az előrejelzés szerint a program folytatódni fog; sőt, elkezdi végrehajtani ezeket a műveleteket, még mielőtt ismert lenne, hogy az elágazás előrejelzés helyes volt-e. Amennyiben később az derül ki, hogy az elágazás előrejelzés helytelen volt, visszamegy az elágazás helyére és megkezdi az utasításokat elővenni és végrehajtani a másik irányban. A “Fetch control” jelű blokk foglalja magában az elágazás előrejelzést, amelynek az a feladata, hogy meghatározza, melyik utasításokat kell elővenni.

[1] Itt az elágazás alatt feltételes ugró utasítást értünk. Más utasítások, amelyek másik helyre viszik át a vezérlést (mint visszatérés szubrutinból vagy indirekt ugrás), hasonló kihívást jelentenek a processzor számára.

Az utasítás dekódoló logika az aktuális program utasításokat primitív műveletek (mikro műveletek) sorozatává alakítja. Ezek a műveletek olyan egyszerű számítási műveleteket végeznek, mint két szám összeadása, adat olvasása a memóriából vagy adat írása a memóriába. Komplex utasítás készletű számítógépek esetén (ilyen az x86 is) egy utasítás változó számú műveletbe dekódolódik. Annak részletei, hogy az utasítások hogyan dekódolódnak primitív műveletek sorozatára, eltérőek a különböző számítógépek esetén, és az eléggé céges tulajdonú információ. Szerencsére azonban programjainkat egy bizonyos gépi implementáció alacsony szintű ismerete nélkül is tudjuk optimalizálni.

Egy tipikus x86 implementáció esetén, egy

```
addl %eax,%edx
```

jellegű utasítás, amelyik csak regiszterekkel operál, egyetlen műveletté konvertálódik. Másrészt viszont az olyan utasítás, amelyik egy vagy több memória hivatkozást tartalmaz, mint pl.

```
addl %eax,4(%edx)
```

több műveletté alakítódik, ahol a memória műveleteket elválasztják az aritmetikai műveletektől. Ez az utasítás három műveletté dekódolódik: egy ahhoz szükséges, hogy

az értéket a memóriából a processzorba töltsük, egy ahhoz, hogy az %eax regiszterben levő értékhez hozzáadjuk a betöltött értéket, és egy pedig ahhoz, hogy az eredményt visszaírjuk e memóriába. Az ilyen fajta dekódolás részekre osztja az utasításokat és lehetővé teszi, hogy a munkát megosszuk különböző dedikált hardver egységek között. Ezek az egységek aztán több utasítás különböző részeit párhuzamosan tudják végrehajtani.

Az EU a műveleteket az utasítás beolvasó egységtől kapja; tipikusan egy órajel alatt többet is. Ezeket a műveleteket kiosztja a rendelkezésre álló funkcionális egységeknek, amelyek aztán elvégzik az aktuális műveletet. Ezek a funkcionális egységek egy bizonyos típusú művelet végrehajtására specializálódtak. 1.4 ábránk néhány ilyen tipikus funkcionális egységet mutat, az Intel Core i7 alapján. Azt látjuk, hogy három funkcionális egység a számításhoz van rendelve, a maradék kettő pedig a memória olvasás (read) és írás (write) funkciókhoz. Az egyes számítási egységek több különféle műveletet tudnak elvégezni: mindegyikük képes az alapvető egész műveletek elvégzésére, mint például az összeadás és a bitenkénti logikai műveletek. A lebegőpontos műveletek összetettebb hardvert követelnek, így ezekhez speciális funkcionális egységek szükségesek.

A memória írását és olvasását a load and store egységek végzik. A load egység kezeli azokat a műveleteket, amelyek adatot töltenek a memóriából a processzorba. Ez az egység összeadót is tartalmaz, a címek kiszámítására. Hasonlóképpen, a store egység kezeli a processzorból a memóriába adatot író műveleteket. Ebben is van a címek számítására összeadó. Mint az ábrán látható, a load és a store egységek a memóriát legutoljára használt értékeket tartalmazó nagy sebességű gyorsítótáron (cache) keresztül érik el.

Spekulatív végrehajtást használva, a műveletek kiértékelődnek, de a végeredmények nem kerülnek be a program regiszterekbe és a memóriába, amíg a processzor biztos nem lesz abban, hogy melyik utasítást is kellett valójában végrehajtani. Az elágazási utasítások is elküldődnek az EU-nak, de nem azzal a céllal, hogy meghatározza, hová történik az elágazás, hanem hogy megállapítsa, helyes volt-e az előrejelzés. Ha az előrejelzés helytelen volt, akkor az EU törli az eredményeket, amelyeket az elágazási pont után számolt. Értesíti a branch egységet, hogy az előrejelzés helytelen volt és jelzi a helyes elágazási címet is. Ebben az esetben az elágazási egység az új címről kezdi meg beolvasni az utasításokat. Egy ilyen helytelen előrejelzés (misprediction) komoly veszteséget jelent a hatékonyságban. Egy darabig eltart, amíg az új utasítások

beolvasódnak, dekódolódnak és elküldődnek a végrehajtó egységnek.

Az ICU egységen belül a visszavonási egység (retirement unit] követi az éppen folyó feldolgozást és biztosítja, hogy az betartsa a gépi kódú program szekvenciális szemantikáját. Az ábra szerint a visszavonási egység részeként a regiszter tömb tartalmazza egész, lebegő pontos és újabban SSE regisztereket, mivel ez az egység vezérli a regiszterek frissítését. Miután egy utasítás dekódolódott, az erről szóló információ belekerül egy FIFO (first in-first out) sorba. Ez az információ a sorban két esemény egyikének bekövetkeztéig marad. Először, ha az utasításhoz szükséges műveletek befejeződtek és az ehhez az utasításhoz vezető elágazás előrejelzése helyesnek bizonyult, akkor az utasítás visszavonható, és a program regiszterek frissítését el lehet végezni. Másrészt, ha az ehhez az utasításhoz vezető elágazás előrejelzése helytelennek bizonyul, az utasítást el kell távolítani, az összes általa kiszámított eredménnyel együtt. Ilyen módon a hibás előrejelzések nem módosítják a program állapotát.

Amint fentebb leírtuk, a program regisztereinek frissítése csak akkor történik meg, amikor az utasítások visszavonódnak, ez pedig akkor következik be, ha a processzor biztos lehet abban, hogy az ehhez az utasításhoz vezető elágazások előrejelzései helyesnek bizonyultak. Hogy az egyik utasítás egy másikkal gyorsabban tudjon kom-

munikálni, az információ nagy részét a végrehajtó egységek egymás között kicserélik, amit az ábrán “Operation results" jelöl. Mint azt a nyilak jelzik, a végrehajtó egységek közvetlenül tudnak egymásnak eredményeket küldeni.

A leggyakrabban használt mechanizmus az operandusok végrehajtó egységek közötti kommunikációjának vezérlésére a regiszter átnevezés (register renaming). Amikor dekódolódik egy utasítás, amelyik frissíteni fogja az r regisztert, a művelet eredményéhez egy egyedi t azonosító rendelődik. A regisztert frissítő művelethez egy táblázat tartozik, amelyik az r és t közötti kapcsolatot tartja nyilván, és ehhez hozzáadódik egy (r, t) bejegyzés. Amikor ezután egy olyan utasítás dekódolódik, amelyik az r regisztert használja, a végrehajtó egységnek küldött művelet tartalmazni fogja a t-t az operandus értékének forrásaként. Amikor valamelyik végrehajtó egység befejezi az első műveletet, az eredményt (v, t) formában állítja elő, jelzendő, hogy a t azonosítóval megjelölt művelet a v eredményt produkálta. Azok a műveletek, amelyek a t forrásra várnak, a v értéket fogják használni forrás értékként, ami az adattovábbítás egy formáját jelenti. Ezzel a mechanizmussal az értékek közvetlenül átvihetők egyik műveletből a másikba, ahelyett, hogy a regiszter tömbön keresztül írás és olvasással tennénk azt meg; ez viszont lehetővé teszi, hogy a második művelet közvetlenül azután megkezdődjön,

hogy az első befejeződött. Az átnevező táblázat csak olyan bejegyzéseket tartalmaz, amelyek függő írási művelettel rendelkező regiszterekre vonatkoznak. Amikor egy dekódolt utasításnak szüksége van az *r* regiszterre, és nincs az ehhez a regiszterhez tartozó bejegyzés, az operandust közvetlenül a regiszter tömbből lehet elővenni. A regiszter átnevezés technológiával egész művelet sorozatot lehet spekulatív módon elvégezni, bár a regiszter csak akkor frissítődik, amikor a processzor már biztos az elágazás eredményében.

1.7.2. A funkcionális egységek hatékonysága

A 1.7 táblázat néhány aritmetikai művelet hatékonyságát mutatja Intel Core i7 esetén (az adatok mérésekből és az Intel irodalomból származnak). Ezek az idők más processzorokra is jellemzők. Az egyes műveleteket jellemzi a látencia idő (latency) (az az összes idő, ami a művelet elvégzéséhez szükséges) és a példány idő (issue time) (a minimum órajelek száma két egymás utáni ugyanolyan típusú művelet esetén).

Azt látjuk, hogy a látencia növekszik a szó hosszal (egyszeresről és a dupla pontosságra áttérve), az adat típus bonyolultságával, (egészről lebegőpontosra áttérve) és a művelet bonyolultságával (összeadásról szorzásra áttérve).

Azt is látjuk, hogy az összeadás és a szorzás legtöbb formájánál a példány idő 1, ami azt jelenti, hogy a processzor minden órajelre képes ilyen műveletből egy újat elindítani. Ez a rövid példány idő a csővezetékezés (pipelining) használatával érhető el. Egy csővezetékezett funkcionális egység olyan szakaszokból áll, amelyek mindegyike a művelet egy részét végzi csak el. Például, egy tipikus lebegő pontos összeadó három állapotból áll (és ezért három órajel ciklus a látenciája): egy dolgozza fel a kitevőket, egy összeadja a törtrészt és egy kerekíti az eredményt. Az aritmetikai műveletek a

1.7. táblázat. Az Intel Core i7 aritmetikai műveleteinek látencia és példány idő paraméterei. A látencia idő az aktuális művelet elvégzéséhez szükséges órajelek teljes száma, a példány idő pedig két azonos művelet között szükséges minimális órajel ciklus szám. Az osztás esetében az idők az adattól függenek.

	Egész		Single
Művelet	Látencia	Példány	Látenc
Összeadás	1	0.33	3
Szorzás	3	1	4
Osztás	11-21	5-13	10-15

szakaszokon át szoros egymásutánban hajthatók végre, nem kell megvárni a kezdéssel, amíg az előző befejeződik. Ez a képesség csak akkor használható ki, ha egymás utáni, logikailag független műveleteket kell végezni. Az egy órajel ciklus példány idejű funkcionális egységeket teljesen csővezetékesítettnek nevezik: ezek minden órajelre új műveletet tudnak indítani. Az egész összeadásnál szereplő 0.33 példány idő abból adódik, hogy a hardverben három teljesen csővezetékezett egység van, amelyik egészek összeadását el tudja végezni. A processzorban benne van a lehetőség, hogy három összeadást végezzen minden egyes órajel alatt. Azt is látjuk, hogy az osztó (amit egész és lebegő pontos osztásra, valamint lebegőpontos négyzetgyök vonásra használnak) nem teljesen csővezetékezett: a példány ideje csak pár órajel ciklussal kevesebb, mint a látencia ideje. Ez azt jelenti, hogy az osztónak az utolsó néhány lépés kivételével valamennyi lépést be kell fejeznie, mielőtt egy új műveletet elkezdhet. Az osztás esetén a látencia és a példány idők tartományként vannak megadva, mivel az osztó és az osztandó bizonyos kombinációi több időt igényelnek, mint a többiek. Az osztás viszonylag költséges művelet, hosszú látencia és példány idői miatt.

A példány időt gyakrabban szokták úgy kifejezni, hogy az egység átbocsátó képességét (throughput) adják meg, ami a példány idő reciproka. Egy teljesen csővezetékezett

funkcionális egység maximális átbocsátó képessége órajelenként egy művelet, a magasabb példány idővel rendelkező egységek esetén alacsonyabb érték.

Az áramkör tervezők nagyon széles határok közötti működési jellemzőkkel rendelkező áramköröket tudnak létrehozni. Egy rövid látencia idejű vagy futószalagos egység megvalósítása több hardvert követel, különösen, ha olyan összetett funkcionalitásról van szó, mint szorzás vagy lebegőpontos műveletek. Mivel a mikroprocesszor lapkán csak korlátozott mennyiségű hely áll rendelkezésre, a CPU tervezőknek megfontoltan kell egyensúlyozni a funkcionális egységek száma és azok egyedi hatékonysága között, hogy optimális általános hatékonyságot érjenek el. Ennek érdekében számos különböző benchmark program eredményeit értékelik ki és a legkritikusabb műveletekhez rendelik az erőforrásokat. Amint a 1.7 táblázatból látható, az egész szorzást valamint a lebegőpontos szorzást és összeadást tekintették fontos műveletnek a Core i7 tervezésekor, bár jelentős mennyiségű hardver szükséges az alacsony látencia és nagy fokú futószalagosítás elérésére. Másrészt viszont, az osztás viszonylag ritka művelet, amit nehéz implementálni akár kis látencia idővel vagy teljes futószalagosítással.

Ezen műveleteknek a látencia és példány idői (más kifejezéssel: a maximális átbocsátóképessége) befolyásolhatják kombináló függvényünk hatékonyságát. Ennek hatását

a CPE értékekre vonatkozó két alapvető korláttal fejezhetjük ki:

1.8. táblázat. Az alapvető korlátok hatása az Intel Core i7 esetén

	Integer	
Bound	+	*
Latency	1.00	3.00
Throughput	1.00	1.00

1.8. Some Limiting Factors

a

1.9. Memory Performance

a

1.10. Performance Improvement Techniques

a

1.11. Performance Bottlenecks

a

1.12. Summary

a

Tárgymutató

Táblázatok jegyzéke

Ábrák jegyzéke

Programlisták

Bibliography

[1] Randal E. Bryant and David R. O'Hallaron. *Computer Systems: A Programmer's Perspective*. Pearson, 2014. ISBN: 978-1-292-02584-1.

[2] Irv Englander. *The Architecture of COMPUTER HARDWARE, SYSTEMS SOFTWARE AND NETWORKING An Information Technology Approach*. Fourth. John Wiley & Sons, Inc., 2010. ISBN: 978-0-470-40028-9.

[3] Irv Englander. *The Architecture of COMPUTER HARDWARE, SYSTEMS SOFTWARE AND NETWORKING An Information Technology Approach*. http://www.wiley.com/go/global/englander. 2010.

[4] Neil Matthew and Richard Stones. *Beginning Linux Programming*. http://longfiles.com/fzi3sbsh0lhu/LinuxProgr4th147627.pdf.html. Wrox Press Ltd, 2008. ISBN: 978-0-470-14762-7.

[5] Clive "Max" Maxfield. *DIY Calculator*. http://diycalculator.com/. 2003.

[6] Clive "Max" Maxfield. *How Computers Do Math*. John Wiley & Sons, Inc., 2005. ISBN: 0471732788.

[7] Clive "Max" Maxfield and Alvin Brown. *The Official DIY Calculator Data Book*. John Wiley & Sons, Inc., 2005. ISBN: 0471732788.

[8] Stanley J. Warford. *Computer Systems*. Jones and Bartlett, 2010. ISBN: 0-7637-7144-9.