Optimalizálás a GCC segítségével

Tekintsük át a GCC O kapcsolóinak jelentését, vizsgáljuk meg, hogy bizonyos optimalizálások valójában miért nem azok, aminek gondoljuk őket, valamint hogyan választhatunk alkalmazásainkhoz különleges optimalizálási eljárásokat.

Írásunkban ismertetjük a *GCC* fordító eszközlánc által biztosított optimalizálási szinteket, illetve az ezek által kínált optimalizálási lehetőségeket. Meghatározzuk, hogy mely optimalizálásokat kell explicit módon kiválasztani, ide értve a géptípustól függőket is. Vizsgálatunk elsősorban a *GCC* 3.2.2-es, 2003 februárjában megjelent változatára összpontosít, de megállapításai a jelenlegi, 3.3.2-es változatra is érvényesek.

Optimalizálási szintek

Először nézzük, a *GCC* milyen kategóriákba sorolja az optimalizálásokat, továbbá a fejlesztő hogyan szabályozhatja, hogy mikor melyiket – és sokszor ennél is fontosabb: mikor melyiket nem – kívánja használni. A *GCC* rendkívül sok optimalizálást ismer. Legtöbbjük három szint valamelyikébe tartozik, bár bizonyosak több szinten is elérhetők. Vannak optimalizálások, melyek az eredményként kapott gépi kód méretét csökkentik, mások viszont gyorsabb kódot eredményeznek, akár méretnövekedés árán is. A teljesség kedvéért meg kell említeni a nullás szintet is – explicit módon a `-O` vagy a `-O0` kapcsolóval választhatjuk ki –, melyen semmilyen optimalizálás nem történik.

Az 1. szint (-O1)

Az első optimalizálási szint célja optimalizált kód gyors előállítása. A hangsúly a gyorsaságon van. Az 1. szint két további, sok esetben egymásnak ellentmondó céllal is bír, ezek a kész kód méretének csökkentése és teljesítményének növelése. A `-O1` szint optimalizálásai túlnyomó részt ezeket a célokat szolgálják. Az *1. táblázatban* ezek a `-O1` jelzésű oszlopban szerepelnek. Az első optimalizálási szintet a következő módon engedélyezhetjük:

```
gcc -O1 -o proba proba.c
```

Bármely optimalizálást bármely szinten engedélyezhetünk, ha a `-f` kapcsolóval kísérve megadjuk a nevét:

```
gcc -fdefer-pop -o proba proba.c
```

Megtehetjük azt is, hogy engedélyezzük az első optimalizálási szintet, majd meghatározott elemeit letiltjuk. Erre a `-fno-` előtag szolgál:

```
gcc -O1 -fno-defer-pop -o proba proba.c
```

	Szintek			
Optimalizálás	-O1	-O2	-Os	-O3
defer-pop	●	●	●	●
thread-jumps	●	●	●	●
branch-probabilities	●	●	●	●
cprop-registers	●	●	●	●
guess-branch-probability	●	●	●	●
omit-frame-pointer	●	●	●	●
align-loops	○	●	○	●
align-jumps	○	●	○	●
align-labels	○	●	○	●
align-functions	○	●	○	●
optimize-sibling-calls	○	●	●	●
cse-follow-jumps	○	●	●	●
cse-skip-blocks	○	●	●	●
gcse	○	●	●	●
expensive-optimizations	○	●	●	●
strength-reduce	○	●	●	●
rerun-cse-after-loop	○	●	●	●
rerun-loop-opt	○	●	●	●
caller-saves	○	●	●	●
force-mem	○	●	●	●
peephole2	○	●	●	●
regmove	○	●	●	●
strict-aliasing	○	●	●	●
delete-null-pointer-checks	○	●	●	●
reorder-blocks	○	●	●	●
schedule-insns	○	●	●	●
schedule-insns2	○	●	●	●
inline-functions	○	○	○	●
rename-registers	○	○	○	●

1. táblázat A GCC optimalizálásai és azok a szintek, melyeken engedélyezve vannak

2. táblázat *x86 géptípusok*

Célprocesszor típusa	-march= típus
i386 DX/SX/CX/EX/SL	i386
i486 DX/SX/DX2/SL/SX2/DX4	i486
487	i486
Pentium	pentium
Pentium MMX	pentium-mmx
Pentium Pro	pentiumpro
Pentium II	pentium2
Celeron	pentium2
Pentium III	pentium3
Pentium 4	pentium4
Via C3	c3
Winchip 2	winchip2
Winchip C6-2	winchip-c6
AMD K5	i586
AMD K6	k6
AMD K6 II	k6-2
AMD K6 III	k6-3
AMD Athlon	athlon
AMD Athlon 4	athlon
AMD Athlon XP/MP	athlon
AMD Duron	athlon
AMD Tbird	athlon-tbird

A fenti paranccsal engedélyezzük az első szintet, majd letiltjuk a `defer-pop` optimalizálást.

A 2. szint (-O2)

A második szinten minden olyan az adott géptípuson támogatott optimalizálás megtörténik, amelynél nem kell a sebesség vagy a méret javára dönteni – itt a két szempont kiegyensúlyozottsága jellemző. A hurkok kibontására vagy a függvények helyi kifejtésére például nem kerül sor – igaz, hogy ezekkel a módszerekkel általában növelni lehet a kód sebességét, ám alkalmazásukkor maga a kód is hízik. A második szintet a következőképpen engedélyezhetjük:

```
gcc -O2 -o proba proba.c
```

Az 1. táblázat a `-O2` szint optimalizálásait is tartalmazza. A `-O2` szint a `-O1` szint összes elemét magába foglalja, illetve számos továbbit is tartalmaz.

A 2.5 szint (-Os)

Különleges optimalizálási szint (`-Os`, mint size, vagyis méret), esetében minden a kódot nem növelő, második szintbeli eljárás engedélyezésre kerül. Ilyenkor a hangsúly a méret korlátozására kerül, a sebesség ellenében. Tartalmaz minden második szintű optimalizálást, kivéve a határhoz igazítási (alignment) eljárásokat. A határhoz igazítás azt jelenti, hogy minden függvényt, hurkot, ugrást és címkét olyan címre tolunk el, mely a kettő valamely hatványának többszöröse. Az eljárás géptípustól függő. A határokhoz igazítással a kód és az adatterek mérete és a futtatás sebessége egyaránt nő; ez az oka annak, hogy ezek az eljárások ezen a szinten letiltásra kerülnek. A méretoptimalizálást a következőképpen engedélyezhetjük:

```
gcc -Os -o proba proba.c
```

A gcc 3.2.2-es változata alatt a `-Os` szinten a `reorder-blocks` engedélyezve van, a 3.3.2-es változatnál viszont le van tiltva.

A 3. szint (-O3)

A harmadik, és egyben legmagasabb szint újabb optimalizálásokat tartalmaz (lásd az 1. táblázatot), esetében az elsődleges szempont a sebesség növelése, akár méretnövekedés árán is. A `-O2` szint optimalizálásain túl magában foglalja a `rename-registert` is. Az `inline-functions` (függvények helyi kifejtése) optimalizálást szintén végrehajtja, amivel javul a kód teljesítménye, ám nagymértékben nőhet a mérete is, függően attól, hogy pontosan milyen függvényeket érint a művelet. A harmadik szintet az alábbi módon engedélyezhetjük:

```
gcc -O3 -o proba proba.c
```

Bár a `-O3` szinten gyorsabb kódot kapunk, a méretnövekedés kiolthatja a sebességnövekedés kedvező hatásait. Ha például a kód mérete meghaladja a rendelkezésre álló utasítás-gyorsítótár méretét, akkor számos teljesítménycsökkentő tényezővel kell számolnunk. Lehetséges tehát, hogy a `-O2` szint alkalmazásával jobban járunk, hiszen esetében nagyobb a valószínűsége annak, hogy a kód elfér az utasítás-gyorsítótárban.

A géptípus megadása

Az eddig említett optimalizálások komoly javulást eredményezhetnek az alkalmazás teljesítményében és méretében, ám a célgép típusát megadva további előnyökre is számíthatunk. A gép processzorának típusát a `gcc -march` kapcsolójának segítségével adhatjuk meg. (*2. táblázat*)
Az alapértelmezett géptípus az i386. A GCC minden más i386/x86 alapú géptípuson is működik, ám az újabb processzorok esetében előfordulhat, hogy gyengébb teljesítményt kapunk. Ha fontos a kód hordozhatósága, akkor a fordítást az alapértelmezett beállítással végezzük.
Ha inkább a teljesítmény növelését tartjuk szem előtt, akkor válasszuk ki a gépünknek megfelelő típust.
A géptípus kiválasztásának teljesítményre gyakorolt hatását egy egyszerű példával szemléltethetjük. Készítsünk egy egyszerű próbaprogramot, mely tízezer elemen végez buborékrendezést. A tömbbe az elemeket fordított sorrendben helyezzük el, vagyis a legrosszabb, legtöbb műveletet kívánó esetet vizsgáljuk. A fordítás menetét és a futtatási időket az *1. kódrészlet* ismerteti.
A géptípus megadásával – ez esetben egy 633 MHz-es ***Celeron*** processzorról volt szó – a fordító képessé válik arra, hogy az adott processzortípushoz leginkább illeszkedő utasításokat állítson elő, illetve egyéb, kifejezetten a géptípusra jellemző optimalizálásokat is el tud végezni. Amint az 1. kódrészlet is szemlélteti, a géptípus megadásával a futtatási

3. táblázat *A matematikai egységekkel kapcsolatos optimalizálások*

Beállítás	Leírás
387	Szabványos, 387-es lebegőpontos társprocesszor
sse	Streaming SIMD Extensions (Pentium III, Athlon 4/XP/MP)
sse2	Streaming SIMD Extensions II (Pentium 4)

időben 237 ms-os, vagyis 23 százalékos javulást értünk el. Fontos megjegyezni, hogy az 1. kódrészletben látható sebességnövekedéshez némi méretnövekedés árán jutottunk. A `size` parancs (*2. kódrészlet*) segítségével megvizsgálhatjuk a kód egyes részeinek méretét.
A 2. kódrészletből kitűnik az utasításrész (text rész) 28 bájtos növekedése. Ebben az esetben viszonylag kis árat fizettünk a sebességnövekedésért.

A matematikai egységgel kapcsolatos optimalizálások

Léteznek különleges, az ***i386*** és az ***x86*** géptípusra egyedileg jellemző optimalizálások, melyeket a programozónak kifejezetten ki kell választania, egyébként nem jutnak érvényre. Lehetőség van például matematikai egység választására – igaz, sok esetben ez önműködően megtörténik, a megadott processzor típusa alapján. A `-mfpmath=` kapcsolóval a *3. táblázatban* szereplő egységek közül választhatunk. Az alapérték a `-mfpmath=387`. Létezik egy egyelőre kísérleti jellegű beállítás is, melynél a program az `sse` és a `387` egységet egyaránt megpróbálja kihasználni (`-mfpmath=sse,387`).

Határhoz igazítási optimalizálások

A második szintnél jó néhány határhoz igazítási optimalizálásról volt szó. Ott említettem azt is, hogy ezekkel javítani lehet a teljesítményen, ám méretnövekedést eredményeznek. Ehhez a géptípushoz további három határhoz igazítási eljárás is létezik. A `-malign-int` segítségével a típusokat 32 bites határokhoz tudjuk igazítani. Ha 16 bites igazítású gépre fordítjuk a kódot, a `-mno-align-int` eljárást használhatjuk. A `-malign-double` optimalizálás segítségével a `double`, a `long double` és a `long long` típusokat tudjuk kétszavas határokra rendezni (letiltása a `-mno-align-double` paranccsal lehetséges).
A `double` típusok határhoz igazításával ***Pentium*** gépeken érhetünk el jobb teljesítményt, természetesen a méret rovására. A `-mpreferred-stack-boundary` beállítással a verem igazítására is van lehetőség. Esetében a fejlesztőnek a kettő valamely hatványát kell megadnia a határhoz igazításhoz. Ha például a fejlesztő a `-mpreferred-stack-boundary=4` értéket adja meg, akkor a verem 16 bájtos határhoz igazodik – ez az alapbeállítás is. ***Pentium*** és ***Pentium Pro*** processzorokon a verem `double` változóit 8 bájtos határhoz érdemes igazítani, a ***Pentium III*** processzorok viszont 16 bájtos igazítással teljesítenek jobban.

Sebességnövelés

A szabványos függvényeket – mint a `memset`, a `memcpy` vagy az `strlen` – használó alkalmazások esetében a `-minline-all-stringops` beállítással, a karakterlánc-műveletek helyi kifejtésével tudjuk növelni a teljesítményt. Természetesen mellékhatásként itt is számolnunk kell a kód méretének növekedésével.
A hurkok kibontása úgy történik, hogy a fordító egy-egy ciklusban a lehető legtöbb munkát végezteti el a programmal, így kevesebb ismétlésre van szükség. Ez esetben a teljesítmény javulása és a méret növekedése ismét együtt jár. A hurkok kibontását a `-funroll-loops` beállítással engedélyezhetjük. Azokban az esetekben, amikor az ismétlések számát nehéz meghatározni, márpedig ez a `-funroll-loops` használatának előfeltétele, a `-funroll-all-loops` optimalizálással lehet az összes hurkot kibontani.
Hasznos eljárás a `-momit-leaf-frame-pointer`, ám használata megnehezíti a kód hibáinak felderítését. Segítségével a keret mutatót (frame pointer) egy regiszteren kívül tarthatjuk, így kevesebbszer kell megadni és törölni az értékét. Mindemellett a regiszter elérhetővé válik a kód számára is. A `-fomit-frame-pointer` optimalizálás szintén jó szolgálatot tehet.
A `-O3` szinten, illetve a `-finline-functions` beállítás használatakor egy különleges átadott érték felületen keresztül megszabhatjuk, hogy legfeljebb mekkora függvényeket akarunk helyileg kifejteni. Az alábbi parancsban például a helyi kifejtésű függvények méretét 40 utasításban korlátozzuk:
`gcc -o rendez rendez.c –param max-inline-insns=40`

1. kódrészlet A géptípus megadásának egy egyszerű alkalmazás futására gyakorolt hatása

```
[mtj@camus]$ gcc -o rendez rendez.c -O2
[mtj@camus]$ time ./rendez
real    0m1.036s
user    0m1.030s
sys     0m0.000s
[mtj@camus]$ gcc -o rendez rendez.c -O2 -
➥march=pentium2
[mtj@camus]$ time ./rendez
real    0m0.799s
user    0m0.790s
sys     0m0.010s
[mtj@camus]$
```

2. kódrészlet Az 1. kódrészletben szereplő program méretváltozása

```
[mtj@camus]$ gcc -o rendez rendez.c -O2
[mtj@camus]$ size rendez
   text    data     bss     dec     hex filename
    842     252       4    1098     44a rendez
[mtj@camus]$ gcc -o rendez rendez.c -O2
➥-march=pentium2
[mtj@camus]$ size rendez
   text    data     bss     dec     hex filename
    870     252       4    1126     466 rendez
[mtj@camus]$
```

3. kódrészlet: Egyszerű példa a gprof használatára

```
[mtj@camus]$ gcc -o rendez rendez.c -pg -O2
➥-march=pentium2
[mtj@camus]$ ./rendez
[mtj@camus]$ gprof –no-graph -b ./rendez
➥gmon.out
Flat profile:
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
100.00      0.79     0.79        1   790.00   790.00  bubbleSort
  0.00      0.79     0.00        1     0.00     0.00  init_list
[mtj@camus]$
```

Ezzel a módszerrel kézben tarthatjuk a `-finline-functions` alkalmazása kapcsán jelentkező kódméret-növekedést.

A kód méretének optimalizálása

A verem alapértelmezett határhoz igazítási értéke 4 vagy 16 szó. Helyszűkével küszködő rendszereknél az alapértéket a `-mpreferred-stack-boundary=2` beállítással nyolc bájtra is állíthatjuk. Állandók, például karakterláncok vagy lebegőpontos értékek megadásakor ezek a független értékek általában saját helyet foglalnak el a memóriában. Jobb, ha nem engedjük szabadjára őket, ugyanis az azonos jellegű állandók összefogásával csökkenthetjük a tárolásukhoz szükséges hely méretét. Ezt a különleges optimalizálást a `-fmerge-constants` beállítással vehetjük igénybe.

Optimalizálás grafikus hardverelemekre

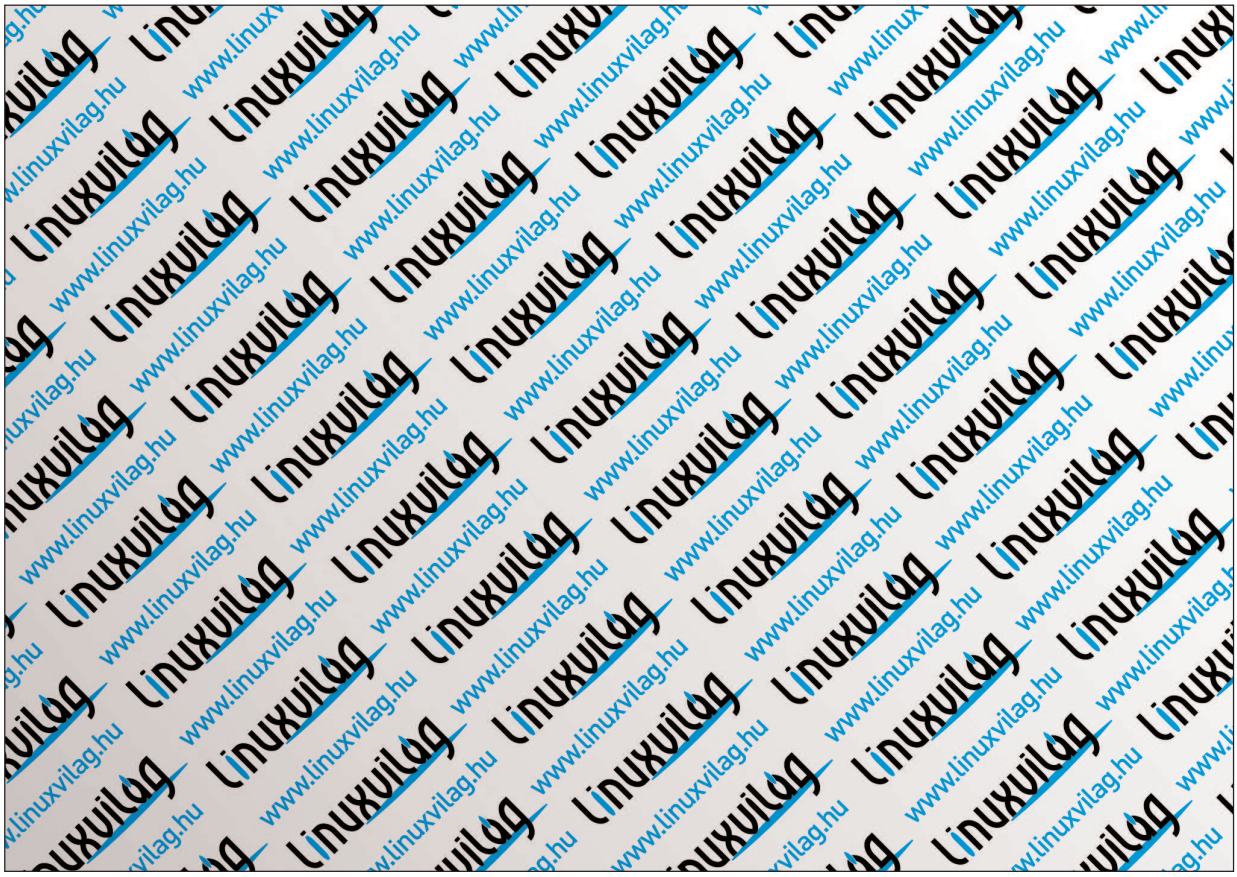
A megadott célgép típusától függően számos további kiterjesztés kerül engedélyezésre. Ezeket explicit módon is lehet engedélyezni vagy tiltani. A `-mmmx` és a `-m3dnow` például önműködően engedélyezésre kerül, amennyiben a megadott processzortípus támogatja az adott utasításkészletet.

További lehetőségek

Számos a sebesség növelésére és a kódméret csökkentésére alkalmas optimalizálásról és kapcsolóról ejtettünk szót, most említsünk meg néhány mellékes, ám sokszor roppant hasznos lehetőséget.

A `-ffast-math` optimalizálás olyan átalakításokat végez, melyek nagy valószínűséggel helyes kódot eredményeznek ugyan, ám nem biztos, hogy szigorúan igazodik az *IEEE* szabvány előírásaihoz. Bátran használhatjuk, ám gondosan teszteljük le az eredményt.

Ha a globális közös alkifejezések kiküszöbölése engedélyezve van (`-fgcse`, `-O2` vagy magasabb szint), akkor



további két lehetőség nyílik a betöltési/elmentési műveletek számának csökkentésére. A `-fgcse-lm` és a `-fgcse-sm` optimalizálás a betöltő és elmentő műveletek hurkon kívülre helyezésével alkalmas a hurkon belül végrehajtott utasítások számának csökkentésére, amivel nő a hurok lefuttatásának sebessége. A `-fgcse-lm` (`load-motion`, betöltő műveletek) és a `-fgcse-sm` (elmentő műveletek) kapcsolókat együtt kell használni.
A `-fforce-addr` optimalizálás arra kényszeríti a fordítóprogramot, hogy a címeket regiszterekbe mozgassa át, mielőtt bármilyen aritmetikai műveletet végrehajtana rajtuk. Hasonló a `-fforce-mem` beállításhoz, mely a `-O2`, a `-Os` és a `-O3` szinten alapesetben engedélyezve van.
A mellékoptimalizálások közül utolsóként a `-fsched-spec-loadot` említeném meg, mely a `-fschedule-insns` optimalizálással együtt használható. A `-O2` szinttől kezdve alapesetben is engedélyezve van. Segítségével mód nyílik bizonyos betöltő utasítások elméletileg hatékonyabb végrehajtást eredményező áthelyezésére, amivel a lehető legkisebbre csökkenthető az adatfüggőségek miatt a futtatásban bekövetkező fennakadások száma.

A kapott javítások kipróbálása

A korábbiakban a `time` paranccsal vizsgáltuk meg az adott utasítások végrehajtására fordított időt. Az alkalmazások profilozásakor természetesen ennél jóval részletesebb betekintést kell nyernünk a kód működésébe. A ***GNU*** `gprof` segédprogram és a ***GCC*** fordító együttesen képesek megfelelni az ilyen jellegű igényeknek is. A `gprof` tárgyalása túlmutatna jelenlegi témánkon, ám a *3. kódrészlet* jól példázza a használatát.
A kódot a `-pg` kapcsolóval fordítjuk le, így belekerülnek a profilozáshoz szükséges utasítások. A kód lefuttatása után az eredmények a `gmon.out` fájlba kerülnek, ebből a `gprof` segédprogrammal állíthatjuk elő az emberi szem számára is olvasható profilozási adatokat. A `gprof` futtatásakor ebben az esetben a `-b` és a `–no-graph` kapcsolókat használtam. Ha tömör kimenetet szeretnénk (vagyis el kívánjuk hagyni a mezők bővebb magyarázatait), a `-b` kapcsolót kell használnunk. A `–no-graph` kapcsoló letiltja a függvényhívási grafikon megjelenítését; ez egyébként az egyes függvények közötti hívásokat és a függvények futtatásának idejét szemlélteti.
A harmadik kódrészletre tekintve megállapíthatjuk, hogy a `bubbleSort`, vagyis a buborékrendezést végző eljárás egyszer került meghívásra, és futtatásának ideje 790 ms volt. Az `init_list` függvényt szintén meghívtuk, ennek futtatása kevesebb mint 10 ms-ot igényelt (ez a profilozási mintavétel felbontása), ezért mellette nullás érték szerepel.
Ha a sebesség helyett inkább a méretváltozások érdekelnek bennünket, akkor a `size` parancsot kell használnunk. Még részletesebb adatokhoz az `objdump` segédprogrammal juthatunk. Például a kódban lévő függvények listáját a `.text` részekre keresve állíthatjuk elő:

```
objdump -x rendez | grep .text
```

A kapott listából ezután ki tudjuk választani a komolyabb érdeklődésünkre is számot tartó függvényt.

Az optimalizálások vizsgálata

A *GCC* optimalizáló lényegében egy fekete doboz. Megadjuk neki a beállításokat és a megfelelő kapcsolókat, majd kapunk egy kódot, ami vagy jobb, vagy rosszabb. Ha javulást látunk, vajon mi történt pontosan? Erre a kérdésre a kód vizsgálatával kaphatunk választ.
A célutasítások kiírására a -S kapcsolóval vehetjük rá a fordítót:

```
gcc -c -S proba.c
```

Ekkor a `gcc` előfordítja a kódot (`-c`, mint compile), illetve megjeleníti a forrás assembly kódját (`-S`). A kapott assembly kimenet a `proba.s` fájlba kerül.
Az előző megközelítés hátránya, hogy csak assembly kódot látunk, a tényleges utasítások méretéről semmit nem tudunk meg. Ha ez a célunk, akkor az `objdump`-pal az assembly mellett natív utasításokat is elő tudunk állítani:

```
gcc -c -g proba.c
objdump -d proba.o
```

Az előfordítást a `-c` kapcsolóval kértük a `gcc`-től, a hibakeresési adatokat pedig a -g kapcsolóval illesztettük be. Az `objdump` a `-d` kapcsoló hatására szedi szét az objektumkódban szereplő utasításokat. Végül az assemblyvel megszórt forrást az alábbi paranccsal kapjuk meg:

```
gcc -c -g -Wa,-ahl,-L proba.c
```

A parancs futásakor a ***GNU*** assembler állítja elő a kódforrást. A `-Wa` kapcsolóval a `-ahl` és a `-L` kapcsolót adjuk tovább az assemblernek, amely így a szabványos kimenetre magas szintű kódból és assemblyből felépülő tartalmat ír ki. A `-L` kapcsoló a szimbólumtábla helyi szimbólumainak megtartását szolgálja.

Összefoglalás

Mivel minden alkalmazás más, nem adható olyan bűvös beállításegyüttes, amellyel minden esetben a legjobb eredményt lehetne elérni. Megfelelő teljesítményt a legegyszerűbben a `-O2` optimalizálási szint használatával kaphatunk. Ha a hordozhatóság nem szempont, akkor a `-march=` kapcsolóval adjuk meg a célprocesszor típusát. Ha tárhely tekintetében szűkösen állunk, akkor elsőként a `-Os` szinttel próbálkozzunk. Ha a lehető legnagyobb teljesítményt akarjuk kipréselni a kódból, akkor több szinttel is próbálkozzunk meg, a kapott kódot pedig vizsgáljuk meg az említett segédprogramokkal. Adott optimalizálások engedélyezésével vagy letiltásával szintén van esélyünk arra, hogy a lehető legjobb teljesítményt hozzuk ki a fordítóból.

Linux Journal 2005. március, 131. szám

A cikkhez tartozó források elérhetősége:
➲ ***www.linuxjournal.com/article/7971***

M. Tim Jones (mtj@mtjones.com)
A longmonti, Colorado állambéli Emulex Corp. vezető főmérnöke. Belsőprogram-tervező mérnök, emellett nemrég készült el BSD Sockets Programming from a Multilanguage Perspective című könyvével. Korábban kommunikációs és tudományos műholdakhoz írt rendszermagokat, jelenleg hálózati készülékekhez fejleszt belső programokat.