Az Intel fordító működése

Hogyan győzte le méréseink során az Intel fordítója a gcc-t?

A Linux egyre növekvő szerepének következtében a fejlesztők és a kutatók egyre több fejlesztőeszközt igényelnek. Ebből az okból született meg az Intel legújabb, kiválóan testreszabható C++- és Fortran-fordítója is, amit kifejezetten olyan alkalmazásfejlesztőknek szánnak, akik az Intel IA-32 és Itanium rendszerein dolgoznak. Ezek a fordítók támogatják a szigorú ANSI szabványt és néhány más népszerű kiterjesztést is. Ez a cikk az Intel IA-32-es fordítójának a tulajdonságaival és hatékonyabbá tételével foglalkozik. A cikk hátralévő részében az Intel IA-32-es C++- és Fortran-fordítóira csak „Intel-fordító”-ként fogunk hivatkozni.
Az Intel fordítója a program minden szintjén megpróbálja feljavítani annak képességeit: a magas szintű ciklusoktól és függvényhívásoktól kezdve egészen a fordító adatfolyamának hatékonyabbá tételéig és az alacsonyszintű változtatásokig – ilyen például az utasításütemezés, az alapszintű szerkezetkialakítás és a regiszterfoglalás. Ebben a cikkben elsősorban csak az Intel fordítójára jellemző hatékonysági jellemzőkkel foglalkozunk.

Hagyományos hatékonyságnövelő ötletek

A két leggyakrabban használt hatékonyságnövelő módszer a sokszor végrehajtódó utasítások számának csökkentése és az utasítások gyorsabb változataikra történő cserélése. Ebbe a két csoportba számos hagyományos egyszerűsítő eljárás tartozik: a másolás és változó híresztelés, a felesleges kifejezések eltüntetése, a felhasználatlan kódok eltávolítása, az átlátások megszüntetése, a függvénybeszúrás, a záró műveletek többszöri ismétlődésének eltakarítása és így tovább.
Az Intel fordítója mindkét esetre rengeteg egyszerűsítő módszert tud. Rengeteg helyi egyszerűsítés az úgynevezett SSA-n, vagyis a maradandó egyszeri hozzárendelésen alapul. A felesleges (vagy részlegesen felesleges) kifejezések eltávolításáért felel többek között a Chow-algoritmus is (lásd az ***Irodalomjegyzék*** 6. pontjában), ami akkor tekint egy kifejezést feleslegesnek, ha szükségtelenül többször kerül kiértékelésre egy végrehajtási szakasz során. Például a következő állításban:

```
x[i] += a[i+j*n] + b[i+j*n];
```

Az `i+j*n` kifejezés kétszeri kiértékelése felesleges. A részleges ismétlődés azt jelenti, hogy egy kifejezés valamelyik része bizonyos végrehajtási szakaszokon többször ismétlődik, de nem feltétlenül az összes végrehajtási szakaszon. Vegyük a következő példát:

```
if (c) {
    x = y+a*b;
} else {
    x = a;
}
z = a*b;
```

Az `a*b` kifejezés részlegesen felesleges. Ha az `else` hajtódik végre, az `a*b` kifejezés csak egyszer értékelődik ki; ha azonban az `if` ág hajtódik végre, akkor a kifejezés kétszer számolódik ki. A kódot így módosíthatjuk:

```
t = a*b;
if (c) {
    x = y+t;
} else {
    x = a;
}
z = t;
```

Így az `a*b` eredménye csak egyszer értékelődik ki, függetlenül attól, hogy melyik ág hajtódott végre.
Ezt az átalakítást megfontoltan kell használnunk, mivel az ideiglenes változók számának növekedése, amelyek elvileg regiszterekben tárolódnak, az élettartam növekedéséhez vezethetnek, ezáltal a regiszterek számának a csökkenése következik be. Egy másik algoritmussal (az ***Irodalomjegyzék*** 9. pontja) – ami nagyon hasonlít Chow algoritmusához – a nem használt tárolókat szüntethetjük meg, amiket még a lekérdezésük előtt egy másik tároló vált fel, illetve a részlegesen használt tárolókat, amik csak bizonyos futási szakaszokon nem kerülnek felhasználásra. Az SSA-n alapuló egyéb hatékonyságnövelő eljárások a változó- és feltételkiértékelésekre vonatkoznak. Vegyük a következő példát:

```
if (x>0) {
    if (y>0) {
        . . .
        if (x == 0) {
            . . .
        }
    }
}
```

A legkülső feltételből egyértelműen kiderül, hogy `x` nem lehet egyenlő nullával, így a belső feltétel értelmetlen, ezért kiértékelése szükségtelen. Az előző példák talán mesterkéltnek tűnhetnek, de gyakran adódhatnak hasonló helyzetek címek kiszámításakor, makrókban vagy beszúrt függvényekben.
Az Intel fordítója erőteljes memória-összezavarást alkalmaz, hogy észrevegye, ha bizonyos memóriahivatkozások esetleg átfedik egymást. Ez az elemzés fokozza például a regiszterkihasználást, és lehetővé teszi a kódban lévő párhuzamosságok észlelését és kiaknázását – erre a következőkben még bővebben kitérünk. Az Intel fordítója az eljárásközi hívásokat is egyszerűbbé teszi, ebbe beletartozik az önműködő és kézi függvénybeszúrás vagy a részleges függvénybeszúrás, ahol a függvényből csak a szükséges részek kerülnek be. A fordító egyszerűbbé teszi az eljárásközi állandókat és a kivételkezelést is. Ha a „teljes program” elemzést bekapcsoljuk, bizonyos adatszerkezetek elhelyezkedése megváltozhat – például a Fortran COMMON BLOCK-jai –, hogy a memóriaelérés az adott processzoron hatékonyabbá váljon. Például az adatok elhelyezését

különféle határokhoz lehet igazítani. Ráadásul az Intel fordítója még okosabb döntéseket hozhat, meg tudja határozni, hogy mikor és hol van szükség egy-egy beszúrásra. Ehhez kétféle kialakítási eljárást használ fel: a nyugvó kialakítást és a változó kialakítást. A nyugvó kialakítás olyan adatokra hivatkozik, amik fordítási időben megbecsülhetők vagy kikövetkeztethetők, míg a változó kialakítás a program futása közben szerzett adatoknak megfelelően alakul. A kialakítás e két fajtájáról lesz szó a következő részben.

A kialakítás egyszerűsítése

Elsőként vegyük a nyugvó kialakítást, amire ebben a kódrészletben láthatunk példát:

```
g();
for (i=0; i<10; i++) {
    g();
}
```

Nyilvánvalóan a cikluson belüli hívás tízszer többször hajtódik végre, mint a cikluson kívüli hívás. Ezt sok esetben azonban nem lehet ilyen jól megbecsülni, mint ahogyan a következő példában sem:

```
for (i=0; i<10; i++) {
    if (feltØtel) {
        g();
    } else {
        h();
    }
}
```

Itt nehéz megmondani, hogy melyik függvény fog nagy valószínűséggel végrehajtódni. Ha a `h()` eljárás a program futásának megszakítását eredményezné vagy valamilyen egyéb olyan eljárás volna, ami nem tér vissza, akkor az első ág, vagyis a `g()` eljárás beszúrásával többre mennénk, mivel az valószínűleg többször hajtódna végre. Ilyen értesülések hiányában azonban rendkívül nehéz eldönteni, hogy melyik eljárást szúrjuk be, esetleg mindkettővel megtegyük-e. A másik lehetőség, hogy változó kialakítást alkalmazunk.

A változó kialakítás a program végrehajtása során nyeri az adatait. Ezzel a fordítót olyan előnyhöz juttatjuk, hogy módjában áll megismerni a program futásának menetét, és ennek megfelelően tudja egyszerűsíteni. Egy háromlépéses művelet részeként az alkalmazást először a kialakítást feltérképező műveletekkel együtt fordítjuk. Ezt követően a programot mintaadatokkal futtatjuk le, amiből a fordító számára összeáll egy olyan adatbázis, amit egy újabb fordítás alkalmával felhasználhat. Végül az adatbázisban található adatok segítségével a fordító már könnyen meghatározhatja, hogy melyik gyakran végrehajtódó programrészt helyezze egy csoportba, vagy a kód melyik részét kell átrendeznie, milyen függvényeket kell teljesen vagy részlegesen beszúrnia, illetve hogyan alakítsa ki a regiszterek lefoglalását. A regiszterfoglalás az Intel fordítójában gráfalapú egyesítésekre épül (lásd még az ***Irodalomjegyzék*** 5. pontjában), ami a kódot régiókba tagolja. Ezek a régiók általában ciklusok törzsei vagy egyéb összetartozó egységek. A kialakítási adatok birtokában a régiók hatékonyabban kiválaszthatók, és a programrészek tényleges végrehajtódásának a számán alapulnak, nem találgatásokon. Ilyen módon a programtörmelékek a program ritkábban futtatott területeire helyezhetők.

Regiszteren belüli csoportosítás

A jelenlegi gépeken a párhuzamosan futtatható műveletek feltárása az egyszerűsítés érdekében rendkívül fontos. Az Intel fordítója kulcsszerepet játszhat a programon belüli párhuzamosan futtatható műveletek megtalálásáért folytatott erőfeszítésekben, úgy, hogy megkönnyíti az olyan hatékonyságnövelő eljárásokat, mint az önműködő összegyűjtés, az önműködő párhuzamosítás, vagy az OpenMP parancsainak a támogatása. Nézzünk meg egy példát, amiben azt mutatjuk be, miként alakul át a soros ciklus önmagától egy olyan formátumba, ami az Intel MMX és SSE/SSE2 (folyamatos SIMD-kiterjesztés) megoldásait képes kihasználni. Ezt a folyamatot „regiszteren belüli csoportosításnak" (Intra-Register Vectorization) hívjuk (lásd az ***Irodalomjegyzék*** 1. pontjában). Példánkban a következő függvény adott:

```
void vecadd(float a[], float b[],
➥float c[], int n)
{
  int i;
  for (i = 0; i < n; i++) {
       c[i] = a[i] + b[i];
  }
}
```

Az Intel-fordító az `addps` utasítás segítségével a ciklust négy egyszeres pontosságú lebegőpontos műveletté alakítja át, amelyek egy időben hajtódnak végre. Ilyenformán az eredmény a következőképpen nézne ki:

```
for (i = 0; i < n; i+=4) {
     c[i:i+3] = a[i:i+3] + b[i:i+3];
}
```

A skaláris eltakarító ciklus végrehajtaná a fennmaradó utasításokat, ha az `n` lépésszámláló nem lenne pontosan osztható néggyel. Ebben a lépésben több folyamat vesz részt. Elsőként – mivel elképzelhető, hogy a tömbök kezdőcímeiről nem áll rendelkezésre adat – egy futásidejű kódot kell beszúrni, hogy meg lehessen győződni arról, hogy a tömbök adatterületei nem fedik-e egymást (önműködő függéspróba), és hogy a ciklus fennmaradó része 16 bájtos határokra igazított címekkel fut-e (önműködő cikluscsupaszítás a megfelelő elrendezéshez).

A hatékony csoportosítás érdekében csakis ciklusokat vagy megfelelő hosszúságú kódrészeket csoportosítunk. Ha az ismétlésszám túl kicsi, akkor egy egyszerű soros ciklust használunk. Az egyszerű ciklusok mellett a csoportosító támogatja a csökkenő elemű ciklusokat is (mint amilyen például a számokból álló tömb összegzése, vagy egy tömb legkisebb és legnagyobb elemének megkeresése, feltételes szerkezetek, szórási számtan és egyéb kifejezések). Még a trigonometrikus matematikai függvények csoportosítása is lehetséges a csoportosítási matematikai könyvtár segítségével.

Hogy érzékelni lehessen a regiszteren belüli csoportosításnak köszönhető teljesítménynövekedést, a Linpack nevű mérőprogrammal (ez a ➲ http://www.netlib.org/benchmark címről tölthető le, C és Fortran nyelvű) kétszeres pontossággal lemértük a teljesítménykülönbséget. Ez a mérés egy lineárisegyenlet-megoldó teljesítményét méri, ami a tényezőkre bontáshoz és a megoldáshoz a `DGEFA` és a `DGESL` eljárások egyikét használja fel. A mérés főként abból áll, hogy – amíg a tényezők keresése folyik – az egyes együtthatók mátrixának oszlopain ismételten hívogatjuk az elsőszintű `BLAS DAXPY` eljárást. Alapbeállítások-

kal (`-O2`-es kapcsoló), egy 2,66 GHz Pentium 4-es processzorral a mérés eredménye 1,049 MFLOPS egy 100×100-as táblázat kiszámításakor. Ha engedélyezzük a regiszteren belüli csoportosítást (`-xW` kapcsoló) a Pentium 4-es processzorhoz, a teljesítmény egészen 1,292 MFLOPS-ig nő, ami összesen húszszázaléknyi teljesítménynövekedést jelent.

Az OpenMP és az önműködő párhuzamosítás

Az OpenMP a közelmúltban a C/C++ és Fortran nyelvű osztottmemória-kezelésen alapuló párhuzamos programozás szabványává vált. A felhasználó számára anélkül lehetővé teszi a párhuzamos futtatást, hogy a felhasználónak el kellene merülnie a művelet részleteiben: az ismétlések felosztásában, az adatmegosztásban, a szálak időzítésében vagy az összehangolásban. Ezeket a beállításokat alapul véve az Intel fordítója képes önműködően bármilyen kódból párhuzamosan futó kódot előállítani. Az Intel-fordító támogatja az OpenMP C++ 2.0 és OpenMP Fortran 2.0 szabványos beállításokat a párhuzamosság tulajdonságainak meghatározására. Ennek segítségével a többprocesszoros rendszereken az alkalmazások a lehető legnagyobb teljesítmény elérése érdekében igényeiknek megfelelően és pontosan állíthatják be a párhuzamossági tulajdonságokat.
A következőkben egy OpenMP-s beállításokat felhasználó példaprogramot láthatunk az Intel C++ Linux OpenMP fordítóhoz:

```
#define N 10000
void   ploop(void)
{
  int k, x[N], y[N], z[N];
  #gyakorlati pØlda az omp pÆrhuzamos tÆsra
  ➥zÆrt(k) osztott(x,y,z)
  for (k=0;  k<N; k++) {
    x[k] = x[k] * y[k] + workunit(z[k]);
  }
}
```

A `for` ciklust a szálak egy csoportja egymás között felosztva fogja végrehajtani. A `k` változó `private` (zárt), vagyis minden szál saját példányt kap belőle, míg az `x`, `y`, `z` változókat a szálak közösen használják.
A létrejövő többszálú programkód alább látható. Az Intelfordító futásidejű OpenMP hívásokat hoz létre, amik a szálak létrehozásáért, kezeléséért és összehangolásáért (lásd az ***Irodalomjegyzék*** 1–2. pontjában) felelősek.

```
#define N 10000
void  ploop(void)
{
    int k, x[N], y[N], z[N];
    __kmpc_fork_call(loc, 3,
    ➥T-entry(_ploop_par_loop), x, y, z)
    goto L1:
    T-entry _ploop_par_loop(loc, tid, x[],
    ➥y[], z[]) {
       lower_k = 0;
       upper_k = N;
       __kmpc_for_static_init(loc, tid,
       ➥STATIC, &lower_k, &upper_k, ...);
       for (local_k=lower_k;
       ➥local_k<=upper_k;
            local_k++)  {
          x[local_k] = x[local_k] *
          ➥y[local_k]+ workunit(z[local_k]);
       }
       __kmpc_for_static_fini(loc, tid);
       T-return;
    }
L1: return;
}
```

A többszálú kódkezelő beszúrja a szállétrehozó `__kpmc_fork_call` függvényt – `T-entry` belépési ponttal és adatkörnyezettel (például `tid`, thread id, vagyis szálazonosító) minden egyes ciklushoz. Ezzel belép az Intel OpenMP könyvtárába, és létrehoz néhány szálat a `for` ciklus párhuzamos futtatására.
Az OpenMP-beállításokkal kísért soros ciklus többszálú kóddá alakul úgy, hogy a ciklus alsó és felső határolói helyivé alakulnak, illetve a ciklus léptető változója is egyedivé válik. Végül létrejön egy többszálú futásidejű kezdeményező és összehangoló kód minden egyes `[T-entry, T-ret]` pár által meghatározott `T-region`-hoz. A `__kpmc_for_static_init` hívás kiszámolja a helyi ciklus alsó és felső határát, és az időzítő beállításoknak megfelelően sorra lépked az egyes szálakon. A példakódunkban állandó időzítést használunk.
A `__kmpc_for_static_fini` lib-hívás futásidőben értesíti a rendszert, hogy az éppen futó szál végzett egy ciklussal. Ahelyett, hogy a fordító teljes forrás–forrás-átalakítást végezne, mint ahogyan azt más fordítók teszik, például az OpenMP NanosCompiler és az OdinMP, az Intel fordítója ezeket a változásokat beépítetten kezeli. Ez egyprocesszoros rendszerek esetében lehetővé teszi az együttműködést az OpenMP-n kívül más fejlett, magas szintű fordítói módszerekkel, például a csoportosítás és ciklusátalakítások során.
Mindezek mellett a fordító lehetővé teszi az OpenMP-irányítókkal egyénileg beállított párhuzamosságot is, illetve a `-parallel` kapcsolóval a fordító önműködően hoz létre párhuzamos kódot. Ennek a kapcsolónak a megadásával a fordító önmaga ellenőrzi, hogy milyen ciklusok párhuzamosíthatók, vagyis milyen ciklusok nem tartalmaznak ciklusszintű függőségeket. Az önműködő párhuzamosítás a fejlett memóriatisztázó módszert használja fel elemzéséhez, illetve a saját szerkezetelemzőjét, hogy eldöntse, mikor szükséges a párhuzamosítás.

Processzorazonosítás

Az Intel-fordító processzorazonosító tulajdonsággal bír, aminek a segítségével egy-egy elem többféle IA-32 rendszeren is felhasználható – vagy a kézi processzorazonosító, vagy az önműködő processzorazonosító segítségével. A kézi processzorazonosító használatával egy függvénynek többféle változatát is megírhatjuk, amelyek mindegyike kifejezetten egyfajta IA-32-es rendszerhez kötődik, vagy pedig általános érvényű, tehát minden IA-32-es rendszeren használható. Az Intel-fordító létrehoz egy kódot, ami önműködően eldönti, hogy az alkalmazás milyenfajta processzoron fut, és ennek megfelelően választja ki egy-egy függvénynek azt a változatát, amit felhasznál. Ezzel a futásidejű meghatározó rendszerrel a programozó számára lehetővé válik, hogy az alkalmazásába bizonyos rendszerfüggő lehetőségeket építsen be, mint amilyen például az SSE vagy az SSE2, anélkül, hogy feláldozná a rugalmasságot. Továbbá lehetőséget ad egyazon program különböző rendszereken történő futtatására, mégha azok nem is támogatnak bizonyos utasításokat.
Az önműködő processzormeghatározó is hasonlóan működik, azzal a különbséggel, hogy ebben az esetben a fordító az egyes függvények különböző változatait maga hozza létre. A fordítás

alatt a fordító eldönti, hogy különböző rendszerfüggő beállításokkal milyen függvényekből lehet jobb teljesítményt kihozni. Ezt követően ezek a függvények önműködően több változatban jönnek létre. Egy változat készül, ami tartalmazza a rendszerfüggő utasításokat, egy pedig, ami csak az általános utasításokat tartalmazza. Ennek a szolgáltatásnak az az előnye, hogy a programozónak nem kell újraírnia a kódot. Egy egyszerű forrásfájl esetében egy parancssori kapcsolóval megadható az önműködő processzorazonosítás bekapcsolása. Vegyük például a következő függvényt:

```
void init(float b[], double c[], int n)
{
  int i;
  for (i = 0; i < n; i++) {
       b[i] = (float)i;
  }
  for (i = 0; i < n; i++) {
       c[i] = (double)i;
  }
}
```

Az Intel fordítója ennek a függvénynek akár három változatát is létrehozhatja. Készül egy általános változat, ami minden IA-32-es processzoron képes futni. Készül egy változat, ami a Pentium III-as processzor SSE-re épülő lehetőségeit használja ki, és készülhet egy harmadik változat, ami a Pentium 4-es processzort aknázza ki úgy, hogy mindkét ciklust csoportosítja, és felhasználja az SSE2-es utasításokat. A létrejövő függvény a következő azonosítókóddal kezdődik:

```
.L1  testl     $-512, __intel_cpu_indicator
     jne       init.J
     testl     $-128, __intel_cpu_indicator
     jne       init.H
     testl     $-1, __intel_cpu_indicator
     jne       init.A
     call      __intel_cpu_indicator_init
     jmp       .L1
```

A kódban az `init.A`, `init.H` és `init.J` függvény az általános, az SSE és SSE2-es változat, ebben a sorrendben.

Nyelvi kiterjesztések

Amellett, hogy az Intel-fordító szigorúan megfelel az ANSI követelményeinek, léteznek kapcsolók, amelyek számos GCC-lehetőséget fednek le, például a `long long int`, a 0 hosszúságú tömbök és a változó mennyiségű kapcsolóval rendelkező makróhívások. A GCC stílusú beszúrt assembly kód úgyszintén támogatott. A fordító támogatja a DWARF2-es hibakövetési információkat, így a létrejövő programok a GDB-vel is nyomon követhetők. Bizonyos microsoftos kiterjesztések is engedélyezettek, mint például a `__declspec` tulajdonságok, és a Microsoft stílusú beszúrt assembly-kód.
A beszúrt assembly mellett az Intel-fordító beépítetten támogatja az MMX és az SSE/SSE2-es utasításkészleteket is. Így a fordító hozzáfér a rendszerfüggő lehetőségekhez, anélkül, hogy ez bármilyen teljesítménybeli visszaeséssel vagy hibás alkalmazással járna, ami gyakran abból adódik, hogy a beszúrt assembly-részek összeütközésbe kerülnek az Intel fordítója által alkalmazott elemző és átalakító utasításokkal. A beépített lehetőségek felhasználásával a programozó kiaknázhatja a processzorban rejlő lehetőségeket, ugyanakkor élvezheti a regiszterfoglalás, az ütemezés és az egyéb szolgáltatások előnyeit is.

Köszönetnyilvánítás

A szerzők külön köszönetet mondanak ***Zia Ansari***-nak és ***David Kreitzer***-nek azért a segítségért, amit a fordító részleteinek a megértéséhez nyújtottak. Ezenkívül köszönetünket fejezzük ki az Intel-fordítót létrehozó csapat minden tagjának.

Linux Journal 2003. február, 106. szám

Dale Schouten (Dale.A.Schouten@intel.com)
Az Intel-fordítólaborban dolgozik. Az Illinois-i Egyetemen szerzett PhD fokozatot. Dale egyúttal nem hivatásos zenész, és édesapja két különleges gyereknek.

Xinmin Tian (Xinmin.Tian@intel.com)
Szintén az Intel-fordítólaborban dolgozik. Ő irányítja az OpenMP párhuzamosítási csoportot. A Tsinghua Egyetemen számítógépes tudományok terén sorrendben BSc, MSc és PhD fokozatot szerzett.

Aart Bik (Aart.Bik@intel.com)
A számítógépes tudományok terén elért MSc fokozatát az Utrechti Egyetemen kapta, míg PhD fokozatát a Leideni Egyetemen érte el. Jelenleg az Intel-fordítólaborban párhuzamosításon és csoportosításon dolgozik.

Milind Girkar (Milind.Girkar@intel.com)
A számítógépes tudományok terén elért PhD fokozatát az Illinoisi Egyetemen kapta. Jelenleg az IA-32 fordító fejlesztési csoportot irányítja az Intel-fordítólaborban.

Irodalomjegyzék

1. *Aart Bik*, *Milind Girkar*, *Paul Grey* és *Xinmin Tian*: „Efficient Exploitation of Parallelism on Pentium III and Pentium 4 Processor-Based Systems”, Intel Technology Journal, Q1, 2001.
2. *Xinmin Tian*, *Aart Bik*, *Milind Girkar*, *Paul Grey*, *Hideki Saito* és *Ernesto Su*, „Intel OpenMP C++/Fortran Compiler for Hyper-Threading Technology: Implementation and Performance”, Intel Technology Journal, Vol. 6, Q1, 2002.
3. *E. Ayguade*: „NanosCompiler: A Research Platform for OpenMP Extensions”, In Proc. of the First European Workshop on OpenMP, 1999. október
4. *Brunschen* és *M. Brorsson*: „OdinMP/CCp – A Portable Implementation of OpenMP for C”, In Proc of the 1st European Workshop on OpenMP (EWOMP '99), 1999. szeptember
5. Guei-Yuan Lueh, Thomas Gross és Ali-Reza Adl-Tabatabai: „Global Register Allocation Based on Graph Fusion” LCPC, 1996
6. F. Chow, et al. New Algorithm for Partial Redundancy Elimination based on SSA Form. PLDI, 1997
7. *M. Wegman* és *K. Zadeck*: „Constant Propagation with Conditional Branches” ACM TOPLAS, 1991. április
8. *Rakesh Ghiya*, *Daniel Lavery* és *David Sehr*: „On the Importance of Points-To Analysis and Other Memory Disambiguation Methods for C Programs” PLDI, 2001
9. *R. Lo, F. Chow, R. Kennedy, S. Liu and P. Tu:* „Register Promotion by Sparse Partial Redundancy Elimination of Loads and Stores. Proceedings of the ACM SIGPLAN” Conference on Programming Language Design and Implementation, pp. 26–37., 1998. június
10. Intel C- és C++-fordítók Linuxhoz: ➲ http://developer.intel.com/software/products