Budapesti Műszaki és Gazdaságtudományi Egyetem

Virtualizációs technológiák és alkalmazásaik
BMEVIMIAV89

Szoftvertechnikák a környezetváltás elkerülésére

Molnár Gábor

2011. december 16.

Tartalomjegyzék

1. Bevezetés

A virtualizáció napjaink egyik legtöbb figyelmet kapó technológiája, segítségével meglévő gépparkunk jobban kihasználható, felügyelhető, valamint a rendelkezésre állás is növelhető. A szerverközpontokon kívül, kliens oldalon a különböző operációs rendszerek akár párhuzamos használatára, illetve alkalmazói programok inkompatibilitásának feloldására ad effektív megoldást. A virtualizált számítógépek futását a Virtual Machine Monitor (VMM) felügyeli, ennek a feladata a különböző virtuális gépek ütemezése, indítása, leállítása, valamint bizonyos szintű naplózás.

Az x86-os architektúrában az utasításkészletet 4 különböző jogosultságú privélégium szintre, ringre osztjuk, ebből most a legszűkebb 3-as, *user*, illetve a legbővebb 0-s, *supervisor* módra fogok hivatkozni.

Ebben a környezetben hatékony megoldás a bináris fordítással volt, ekkor a vendég gép a végrehajtása előtt a VMM leellenőrzi, hogy a kód *supervisor* szinten fut-e, majd egy just-in-time fordító a *supervisor* szintű kódot kicseréli egy *user* módú alternatívára.

Másik alternatíva a paravirtualizáció volt, ekkor a vendég gépet módosítjuk úgy, hogy „tudjon" arról, hogy virtualizálva van, azaz a *supervisor* módú utasítások helyett egyből a VMM-et hívja. Jelenleg ezt a megoldást főleg a különböző integrációs komponensek használják (pl. VMware Tools, MS Virtual PC Integration Components, VirtualBox Guest Additions).

Később, a hardveres virtualizációt támogató processzorokban bevezették az ún. *root mode*-ot, amiben a VMM fut, így a vendég gépek a fenti 4 ringet látják, amennyiben olyan utasítást hajtanának végre, ami a fizikai hardverhez férne hozzá, úgy a vezérlés a VMM-hez kerül, ami feldolgozza ezt az utasítást, majd folytatja a vendég futtatását.

Napjainkban ez a technológia terjedt el, mostanra már teljesítménye jellemzően (többszörösen) túllépi a szoftveres megoldásokét, azonban a virtualizált gépek környezetváltása teljesítményvesztést jelent. A processzorgyártók próbálják csökkenteni ezt a költséget, azonban léteznek szoftveres megoldások a környezetváltások gyorsítására, illetve elkerülésére. Jelen dolgozat a VMware újabb verzióiban implementált megoldást [1] mutatja be.

2. A környezetváltás gyorsítása

A hardveres virtualizációt támogató a processzor lehetővé teszi a VMM-nek, hogy közvetlenül a processzoron, bármilyen beavatkozás nélkül hajtsa végre a vendég gép utasításait. Amennyiben a vendég gép a jogosultsági szintjén kívül eső utasítást használ, a futtatás megszakad, a vezérlés a VMM-hez kerül, melynek feladata a kivételt okozó utasítás végrehajtása, majd a vendég futtatása folytatódik *user* módban. A VMM-mel szemben fontos kritérium az ilyen váltások minél gyorsabb levezénylése.

Egy ilyen kivétel végrehajtása során általában értelmezni szükséges a kiváltó utasítást, amihez természetesen ismerni kell magát az utasítást is. Sajnos ezt legtöbbször a vendég gép memóriájából tudjuk lekérdezni (a vendég utasításszámlálójának megfelelően), amihez be kell járni a laptáblákat, illetve a

dekódolás is időigényes. A korszerű processzorok már adnak némi információt a hibát okozó utasításról, azonban ez általában nem elegendő.

Ennek a folyamatnak a gyorsítására a VMM cache-elést alkalmaz, melyet a vendég gép utasításszámlálójával indexel. Amikor egy kivétel keletkezik, a vendég gép utasításszámlálóját hashelve, majd a cache megfelelő bejegyzését kiolvasva megkapjuk a kiváltó utasítást és a megkezdhetjük a végrehajtást. Természetesen ellenőriznünk kell, hogy érvényes-e a cache bejegyzés, ezt a vendég gép és a cache-elt kód bitenkénti összehasonlításával tehetjük meg.

További teljesítménynövekedés érhető el úgy, hogy nem magát a dekódolt utasítást cache-eljük, hanem az ez alapján generált kódot. Ekkor a hashfüggvény által mutatott címre ugrunk, ahol a kód leellenőrzi, hogy egyeznek-e még az aktuális, illetve a tárolt utasításadatok.

3. A környezetváltás elkerülése

Az optimalizálás érdekében a vendég gép környezetváltást okozó utasításait kezdetben párokba, majd blokkokba szervezzük és egy ilyen blokkot hajtunk végre egy környezetváltás alkalmával, vagyis a blokk első utasításánál megtörténik a környezetváltás, majd amennyiben a következő néhány utasítás között van *supervisor* módú, így azt is végrehajtjuk ennek a váltásnak a keretében.

A párba szervezést az motiválja, hogy a legtöbb fizikai címkiterjesztést használó 32-bites x86-os operációs rendszer 64-bites laptáblabejegyzéseket használ, melyeket két egymásutáni 32-bites darabban ír vagy olvas. Ezeket az utasításokat szervezzük párokba, így az első környezetváltás után a többi utasítást is végrehajtjuk.

3.1. Statikus blokkszervezés

Az optimalizálás egyik módja a blokkok statikus kialakítása. Ez annyit jelent, hogy a VMM egy kivételt okozó utasítás feldolgozása során megvizsgálja a virtuális gép kódját, hogy a következő néhány (az implementációban 15) utasítás között van-e olyan, ami újabb kivételt okozna. Amennyiben van ilyen utasítás, úgy a VMM a környezetváltás során eddig az utasításig (ha több is van, akkor a legutolsóig) legenerálja a kivételkezelő kódot, majd visszaadja a vezérlést a vendég gépnek (*user* módban).

3.2. Dinamikus blokkszervezés

A kivételt okozó utasítások két csoportba oszthatók, az egyikbe azok tartoznak (pl. CPUID, **OUT**, **HLT**), amelyek végrehajtása mindig kivételt okoz, illetve azok, amelyek néha okoznak kivételt, néha nem. Utóbbi csoport kezelése motiválja a blokkok dinamikus kialakítását.

Lehetséges, hogy eldönthető egy ilyen utasításról az operandusok, illetve a korábbi utasítások szerint, hogy okoz-e kivételt a végrehajtása, viszont az biztos, hogy nagyobb a számításigénye, mint valamilyen heurisztika alkalmazásának, melyről látni fogjuk, hogy az esetek nagy részében jó becslést ad az utasítás kimenetelére vonatkozólag.

Az alkalmazott heurisztika a következő: egy utasítást addig nem tekintünk kivételt okozónak, amíg nem vált ki három környezetváltást, ekkor a fejezet bevezetőjében bemutatott technika szerint járunk el. Ez a megoldás lehetővé teszi a nagy blokkok generálását, túlzott cache-elés nélkül, ami teljesítménycsökkenést okozna.

A felhasznált adatszerkezetre érvényesek a VMM adatszerkezeteivel szemben támasztott általános követelmények, nevezetesen minél tömörebbnek kell lennie és minél gyorsabb elérésűnek. Ezeknek a kritérimoknak megfelelne egy egyszerű, fix méretű hashtábla, de ez magával hozza azt a problémát, hogy információt veszíthetünk, ha kicsinek bizonyul a tábla. Ez esetben semmi nem garantálja, hogy lesznek olyan utasítások, amik elérik a harmadik környezetváltást, elképzelhető, hogy egyáltalán nem sikerül optimalizálni a végrehajtást, sőt bizonyos esetben teljesítménycsökkenést tapasztalunk (pl. olyan gyorsan töltődik túl a tábla, hogy egyetlen utasítás sem lesz, amit blokkba lehetne szervezni egy másikkal).

Ennek az esetnek a kiküszöbölése miatt a választott adatszerkezet egy 8 mélységű FIFO sorokból álló tömb. A kivételt okozó utasítást egy 35 bites hashértékre képezzük le, mely tartalmazza az utasításszámláló értékét, a hosszát illetve az utasítás bitjeit. Ezt a 35 bites értéket felbontjuk egy 9 bites indexre és egy 26 bites tagre. Az index a tömb sorait azonosítja, mivel 9 bites, így 512 sor lehet.

3.3. Vezérlésátadó utasítások a blokkokban

Az eddig tárgyalt megoldások nem foglalkoztak azzal az esettel, amikor valamilyen vezérlésátadó utasítás van a blokkban. Ezek az utasítások különösen akkor problémásak, hogy ha valamilyen feltételes ugrást kell végrehajtani, mert ezek kimenetele nem ismert előre, csak futásidőben derül ki, ezért nem is tudjuk biztosan elvégezni a blokkszervezést. Az ilyen utasítások kezelésére több alternatíva létezik.

3.3.1. Nincs vezlérlésátadás

A legegyszerűbb, és legkevésbé hatékony módszer megengedi az ilyen utasítások blokkba szervezését, de amennyiben ugrás történik, úgy a blokk végrehajtása is befejeződik és visszatérünk *user* módba.

3.3.2. Blokkon belüli ugrások

Az előzőnél lényegesen kifinomultabb megoldás lehetővé teszi a blokkon belüli ugrásokat. Mivel egy blokkon belüli ugrás lényegében azt jelenti, hogy a blokkban lévő kód vegy részét nem kell végrehajtani, így elvégezhetjük a blokkszervezést, úgy, mintha nem történne ugrás, legfeljebb futásidőben az utasítások egy részét nem hajtjuk végre.

A hatékony kód generálásának érdekében a fordító nem lépteti a vendég utasításszámlálóját a blokkon belül, csak akkor, hogy ha véget ér a blokk végrehajtása, illetve olyan rendszerhívás történik, ami kivélt okozhat.

3.3.3. Ciklusok

A legáltalánosabb megoldás tetszőleges számú blokkon belüli ugrást tesz lehetővé, melyekkel akár ciklusokat is végrehajthatunk egy környezetváltáson belül. Ezzel sok környezetváltást meg lehet spórolni, viszont két probléma is jelentkezik: amíg az egyik vendég ciklusait hajtjuk végre, addig nincs interrupt-feldolgozás, illetve a teljesítménycsökkenés léphet fel a túl hosszú végrehajtás során (olyan utasítást, ami nem okoz kivételt, jobb *user* módban futtatni).

Ezeknek a problémáknak az orvoslására megköveteljük, hogy minden iteráció végén, a visszalépésnél ellenőrzni kell,hogy van-e várakozó interrupt és legalább egy környezetváltást kiváltó utasítást le kell futtatunk, különben a visszalépés nem kerülhet bele a blokkba.

Ezen kívül még mindig fennáll annak a veszélye, hogy olyan ciklust hajtunk végre, amiben nincs környezetváltást kiváltó utasítás ,vagy csak olyan van, ami nem mindig okoz kivételt. Ennek orvoslására két csoportra osztjuk az iteráló ugrásokat, az alapján, hogy milyen környezetváltó utasítás van a ciklusmagban (okoz-e mindig kivételt, vagy csak néha). Utóbbi esetben dinamikus szervezésű blokkonként kevés számú, az implementációban maximum 10, ilyen utasítás lehet, előbbinél pedig nem határozunk meg ilyen korlátot, mivel azok mindig környezetváltást okoznak, ott nem léphet fel ez a probléma.

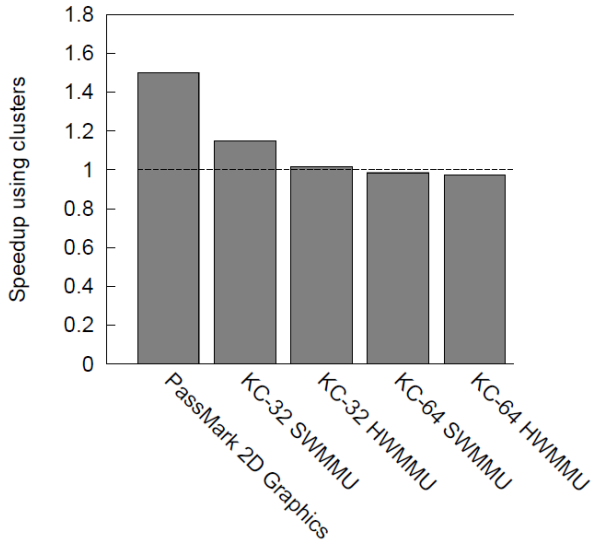
4. A kódkonzisztencia ellenőrzése

Mivel a blokkok kódjait többször felhasználjuk, ezért gondoskodnunk kell arról, hogy ne módosuljon a vendég gép kódja. Eltérés csak abban az esetben történhet, ha a hashelt címen egy új blokk kódját tároljuk, illetve önmódosító programok esetében.

Ezen problémák elhárítása érdekében minden blokk generált kódja egy ellenőrzéssel kezdődik, ha itt eltérést detektálunk, eldobjuk a meglévő tárolt kódot és végrehajtjuk a vendég aktuális környezetváltást okozó kódját. Természetesen az egy címhez tartozó környezetváltások elindítják a blokkszervezési eljárásokat.

A koherenciaellenőrzés a blokkok elején szükséges, azonban nem elégséges védelem a kód módosulásával szemben, lehetséges, hogy a környezetváltáskor még egyezés volt, de a kód változhat a végrehajtás során is, azaz a blokk lehet önmódosító. Ennek a problémának a megoldására a memóriaírások fordításánál ellenőrizni kell, hogy milyen lapokat ír a blokk. Amennyiben ez a lap a blokkhoz tartozik, akkor a blokk végrehajtása megáll. Mivel a blokkok jellemzően kevés utasítást tartalmaznak (a lapmérethez képest), ezért maximum 2 lap elérésével elég jó felső becslést tudunk adni a memóriaelérésre.

5. Összefoglalás



1. ábra. A fejlesztők által elvégzett mérések eredménye

A fent bemutatott technikáknak nagy előnye, hogy amennyiben nem tudnak optimalizálni a virtuális gép működésén, úgy nem lép fel mérhető teljesítményveszteség. A módszerek teljesítményének mérésére Windows XP-s vendég gépen futtattak a PassMark alapú 2D-s grafikai tesztelést, itt nagyjából 1,5-szeres pontszámot kapott a módszereket alkalmazó virtuális gép, valamint 32-bites fizikai címkiterjesztést (PAE) használó, illetve 64-bites Suse Linux 10.1-es vendégeken mértek kernelfordítási időt. Itt főként a 32-bites gépeken sikerült eredményt elérni, nagyjából 1,2-szer kevesebb idő kellett a fordításhoz, főleg a PAE miat, ami két 32-bites memóriaműveletet hajt végre egyszerre (3. fejezet). 64-bites vendég gépek esetén nem sikerült mérhető teljesítménynövekedést produkálni, viszont látszik az, hogy mérési hibán kívüli teljesítménycsökkenés sem történt. Az 1. ábrán látszik a fejlesztők méréseinek az eredménye, a kernelfordításoknál a 32-bites, a fenti technikákat nem alkalmazó vendég ideje a referenciaszint (KC-32 HWMMU).

A fent bemutatott technikák az évek során verzióról-verzióra fejlődtek, mire a jelenlegi állapotukat elérték, azonban még közel nem véglegesek ezek a megoldások, további fejlődés várható. Az első fejlesztési pont az alkalmazott heurisztikákra vonatkozik, ezek helyett egy olyan adaptív költségmodell alkalmazása lehet célszerű, ami meg tudja határozni, hogy egy utasítást megéri-e a meglévő blokkhoz hozzáadni, vagy azt inkább *user* módban érdemes futtatni. Egy ilyen modellnél a különböző CPU sajátosságok miatt mindenképp dinamikusan kell elvégezni a kalibrálást, pl. a VMM indulásakor.

Egy másik irány a blokkok kiterjesztése. Jelenleg a blokkszervezés már megengedi a blokkon belüli ugrásokat, azonban még komoly korlátot jelent, hogy kis számú soron következő utasítást vizsgálunk (ez jelenleg 15) a blokkszervezésnél, illetve ezeknek egymás utániak, valamint a blokkoknak környezetváltást okozó utasítással kell kezdődnie. Ezeket a feltételeket eltörölve, valamint elágazásbecslést, illetve in-order (sorrenden kívüli) végrehajtást is alkalmazva további teljesítménynövekedés érhető el, cserébe a fenti megoldásoknál sokkal komplexebb módszereket igényel.

További fejlődési irány lehet a laptáblabejárások cache-elése, amivel gyorsítható a blokkokon belüli memóriaelérések ideje. A jelenlegi megoldásokban be kell járni a vendég gép laptábláit, majd ezt kell lefordítani a vendég fizikai címére. Mivel pl. a ciklusokban ugyanazokat az utasításokat hajtjuk végre, így elképzelhető, hogy cache-elést alkalmazva csökkenthető a memóriaelérések ideje, ami lehetővé teszi, hogy egy blokkba több memóriaművelet kerülhessen, ezáltal nőhet a blokkok mérete.

Hivatkozások

[1] Ole Agesen, Jim Mattson, Radu Rugina, and Jeffrey Sheldon. Software techniques for avoiding hardware virtualization exits. *VMware Technical Report*, 2011.