Optimalizációs módszerek a beágyazott rendszerek területén

doktori értekezés tézisei

Havasi Ferenc

Témavezető:

Prof. Gyimóthy Tibor
tanszékvezető egyetemi tanár

Informatika Doktori Iskola
Szegedi Tudományegyetem
Szoftverfejlesztés Tanszék

Szeged
2013

Bevezetés

A beágyazott rendszerek életünk egyre több területén körülvesznek bennünket mobiltelefonok, DVD lejátszók, mp3 lejátszók, GPS vevők, autó elektronikai és épületgépészeti vezérlő eszközök, háztartási eszközök egész sora formájában. Technikai szempontból az egyik fő sajátosságuk, hogy erőforrásaik általában sokkal szűkebbek, mint egy személyi számítógépé, így az optimalizáció kérdése a rájuk fejlesztett szoftverek esetében kiemelt fontosságú. Elterjedtségük mértékét jól jelzi, hogy már 2002-ben is a mikroprocesszoroknak csak 2%-kát használták föl személyi számítógépekben [13], a többit (98%-ot) beágyazott rendszerben, és azóta ez az arány tovább nőtt. A terület gazdasági és társadalmi jelentősége ennek köszönhetően egyre nagyobb, így a velük kapcsolatos eredmények is nagy gyakorlati jelentőséggel bírnak.

A dolgozat három beágyazott rendszerekkel kapcsolatos eredményemet mutatja be, egy XML és kettő flash fájlrendszerrel kapcsolatosat.

1. **XML szemantikus kiterjesztés és tömörítés:** kidolgoztam egy, az XML attribútumoknak szemantikus definiálására alkalmas metanyelvet (SRML), amely egyik alkalmazásának köszönhetően az XML fájlok az eddigi XML specifikus tömörítőkhöz képest is akár 9-26%-kal jobban tömöríthetőek. [6] [8] [9]

2. **ARM kódtömörítés:** megalkottunk egy döntési fa alapú ARM kódtömörítésre alkalmas algoritmust, és annak az egyik legelterjedtebb Linuxos flash fájlrendszerbe (JFFS2) történő hatékony beillesztésének módját (egy tömörítési keretrendszert), amely az általános tömörítőkhöz képest a gyakorlatban 12-19%-os helymegtakarítást is jelenthet egy átlagos fájlrendszer image esetében. A keretrendszer része lett a JFFS2 fájlrendszernek és azon keresztül a Linux Kernelnek, a tömörítési módszer pedig egy USA szabadalommal lett levédve. [3] [12]

3. **Flash fájlrendszerekre optimalizált B+ fa algoritmus:** a B+ fa algoritmusát fejlesztettük tovább úgy, hogy az a gyakorlatban flash fájlrendszeri környezetben is hatékonyan használható legyen. Az új adatszerkezet és a kapcsolódó algoritmus magába foglalja a memóriában és a flashen tárolandó adatokat is, figyelve az írási/olvasási teljesítményre, memóriafogyasztásra, illetve a beágyazott rendszereknél gyakori váratlan áramkimaradás kezelésére is. Az algoritmus része lett az UBIFS fájlrendszernek, azon keresztül pedig a Linux Kernelnek, és felhasználásra került a Nokia N900 okostelefonban is. [5]

	Tézis	Publikációk
1.	XML szemantikus kiterjesztés és tömörítés	[6], [8], [9]
2.	ARM kód tömörítésen alapuló flash fájlrendszer helykihasználási optimalizáció	[3]
3.	Flash fájlrendszer teljesítményjavítása továbbfejlesztett B+ fával	[5]

1. táblázat. Tézisek és publikációk

1. XML szemantikus kiterjesztés és tömörítés

Az XML (eXtensible Markup Language) az egyik legelterjedtebb strukturált szöveges fájlformátum, amelyet számos helyen alkalmaznak mind asztali, mind szerver, mind pedig beágyazott rendszeri környezetben, illetve azok közötti kommunikációban. Az itt elért eredmények egyik gyakorlati alkalmazásával az XML fájlok kisebb helyen tárolhatóak el, ami különösen a beágyazott rendszerek és a hálózati alkalmazások területén hasznos.

Az eredmény alapgondolata az XML dokumentumok és az attribútumnyelvtanok analógiáján alapul, amelyet a 2. táblázatban mutatunk be.

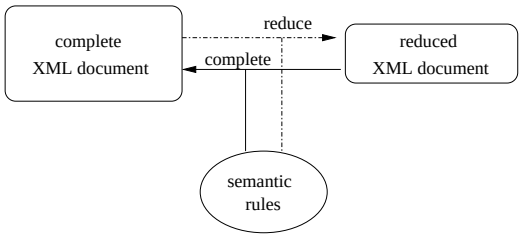
Attribútum nyelvtan	**XML**
nemterminális	XML elem
formális szabályok	XML elem specifikációja a DTD/XSD-ben
attribútum specifikáció	XML attribútum specifikációja a DTD/XSD-ben
szemantikus függvény	—

2. táblázat. Attribútum nyelvtan és XML közötti analógia

Látható, hogy XML környezetben eddig nem volt a szemantikus függvényekkel analóg fogalom. Ezt a hiányt töltöttük be azzal, hogy megalkottuk az SRML-nek elnevezett metanyelvet, amellyel definiálhatók az attribútumok szemantikus kiszámítási módjai. DTD környezetben az SRML külön XML alapú fájlban tárolható, XSD környezetben pedig az XSD appinfo részében.

Az analógiát tekintve különböző típusú attribútum nyelvtanok mintájára definiálhatók a velük analóg SRML típusok, amelyek egyben bejárási stratégiát is adnak az egyes attribútum-előfordulások kiszámítására. A dolgozatban az S-attribútum és L-attribútum nyelvtannal analóg S-SRML és L-SRML leírást definiáltuk.

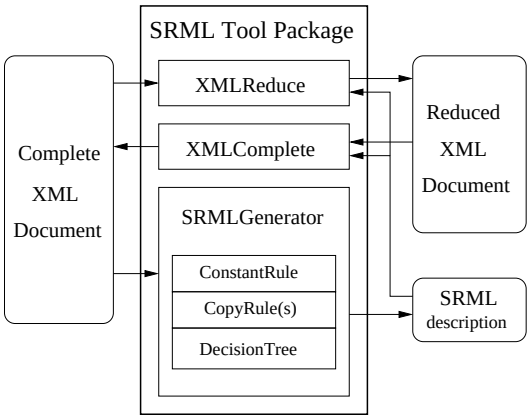
Az SRML egyik alkalmazása a kompaktálás, amelynek elve a 1. ábrán látható.



1. ábra. Kompaktálás és dekomaptálás

A kompaktálást a "reduce" művelet végzi, amely az XML fájlból eltávolítja azokat az attribútumokat, amelyek az SRML szabályai alapján helyesen kiszámíthatóak, míg a "complete" művelet ennek az ellenkezője.

Az SRML-ben lévő szemantikus függvényekben leírt összefüggés két forrásból származhat: meghatározhatják azt az adott XML fájlformátumot ismerő szakértők, illetve előállíthatják gépi tanulási algoritmusok. A két módszer kombinálható is úgy, hogy egy tanuló algoritmusnak kiindulásként odaadjuk a szakértők által alkotott szabályokat szabályokat, amelyeket az tovább bővíthet. Az implementáció (SRML Tool) szerkezetét a 2. ábra mutatja be.



2. ábra. Az implementáció felépítése

Módszerünket a CPPML fájlformátumon próbáltuk ki, amely C++ programok XML reprezentációjára lett kifejlesztve. Az előállításához szükséges kódminőség biztosítása területén világszínvonalú eszköz az SZTE Szoftverfejlesztés Tanszékén került kifejlesztésre Columbus programcsomag néven. A 3. táblázatban látható az eredeti CPPML fájlok mérete, majd a kézzel írt (szakértői) szabályok alapján, gépi tanulásos szabályok alapján, illetve a kettő kombinációjával készített szabályok alapján kompaktált CPPML fájlok mérete az SRML méretét is beleszámolva. A kompaktálási arány 57-79% között mozog.

Fájlméret	Szakértői (kézi) szabály	Gépi tanulásos	Kombinált
SymbolTable (399 321)	296 193 [74.17 %]	313 873 [78.60 %]	281 088 [70.40 %]
Jikes (2 233 822)	1 736 285 [77.72 %]	1 737 872 [77.79 %]	1 367 244 [61.20 %]
AppWiz (3 547 297)	2 238 308 [63.09 %]	2 589 526 [73.00 %]	2 038 569 [57.46 %]

3. táblázat. Kompaktálási eredmények

A kompaktálás használható az XML fájl tömörítésének javítására is, ugyanis a tömörítő programok annál hatékonyabbak, minél több összefüggést fel tudnak ismerni a tömörítendő fájlokban. Ezért van az is, hogy a kifejezetten XML-re specializált tömörítőprogramok (mint amilyen az XMill) jobb tömörítési arányt érnek el XML fájlok tömörítésekor, mint az általános célú tömörítők (mint például a gzip). Ugyanakkor az XML specifikus tömörítők sem képesek fölismerni azokat az összefüggéseket, amelyek az XML attribútumai között vannak. Így ha tömörítés előtt az általunk kifejlesztett módszerrel az XML fájlt kompaktáljuk, majd azt tömörítjük, akkor jobb eredményt fogunk elérni, mint ha csak egyszerűen tömörítenénk azt.

A 4. táblázatban az XMill nevű XML tömörítővel végzett mérések láthatóak. Az első oszlop a CPPML fájl eredeti méretét mutatja, a második az XMill által tömörített fájlméretet, a harmadik a kézzel írt szabály alapjáni kompaktálás utáni XMill tömörítés méretét, míg az utolsó oszlop estében a kombinált szabálygenerálás alapján kompaktált CPPML XMill tömörítés méretét. Látható, hogy a tömörítését a mi módszerünkkel kombinálva az eredeti tömörített mérethez képest akár 9-26%-kal is kisebb méretre tudtuk tömöríteni - természetesen az SRML méretét minden esetben beleszámolva.

Az irodalomban az XML fájlok szemantikus kiterjesztése először [11]-ben volt említve, amelyben a szerzők a DTD-t EBNF formájú formális nyelvtanná alakították, és ahhoz tudtak új attribútumokat definiálni. Ezek attól a pontól kezdve azonban már nem XML attribútumok. Ehhez képest a mi

Fájlméret	Csak XMill	Kézi szabály	Kombinált
SymbolTable	19 786	18 008	17 876
399 321	4.95 %	4.50 %	4.47 %
		8.98 %	9.70 %
Jikes	114 275	108 458	92 102
2 233 822	5.11 %	4.85 %	4.12 %
		5.09 %	19.40 %
AppWiz	145 738	134 217	106 773
3 547 297	4.10 %	3.78 %	3.01 %
		7.90 %	26.73 %

4. táblázat. Az XMill tömörítési eredménye kézzel írt (szakértői) és kombinált szabályok használatával

megoldásunkban valódi XML attribútumokat definiálunk, amely szervesen illeszkedik a jelenlegi XML forma leírókhoz - DTD/XSD - így adva hozzájuk egy természetes "szemantikus" kiegészítést. Ezzel válik lehetővé a kompakció is, mint a tömörítést segítő gyakorlati alkalmazás.

XML attribútumokhoz automatikus szemantikus szabálygenerálást nem találtunk az irodalomban. Attribútum nyelvtanos környezetben [4]-ben a szerzők publikáltak egy módszert, ahol ILP segítségével voltak képesek speciális formájú szemantikus függvényeket megtanulni. Ehhez képest a mi megoldásunk egy olyan keretrendszert definiál, amelybe bármilyen tanulóalgoritmus beleilleszthető, amelyhez mi konstans és másoló szabály-tanuló algoritmus mellett döntési fa alapú tanulási modult is implementáltunk.

A módszer alapgondolata, a metanyelv kidolgozása és az alapalkalmazási ötlet önálló eredményem, amelyet a [6]-ban publikáltam. A módszer tömörítési alkalmazása, szabályok gépi tanulási keretrendszerével kapcsolatos eredmények, melyeket a [8]-ban publikáltunk. A módszer XSD-re kiterjesztése elsősorban Kálmán Miklós szerzőtársam eredménye [9].

2. ARM kód tömörítésen alapuló flash fájlrendszer helykihasználási optimalizáció

A beágyazott rendszerek szűk erőforrásai közül az egyik a háttértár. Bár a multimédia célú eszközökben az adatok tárolását tekintve ez a korlát az utóbbi években egyre kevésbé kritikus, de a kód tárolás/futtatás tekintetében, illetve a (szám szerint túlnyomó többségben lévő) funkcionális jellegű beágyazott rendszereknél továbbra is fontos terület, amelynek az adattároló költsége mellett energiafogyasztási és sebességi vonzatai is vannak.

A háttértár méretének kérdésében az egyik egyszerű megoldási lehetőség a tömörítés. A tömörítési módok közül is az egyik legkényelmesebb a fájlrendszerbe illesztett transzparens tömörítés, amelynek köszönhetően se a felhasználónak, se az egyes alkalmazások fejlesztőinek nem kell törődnie a be- és kitömörítéssel. Ilyen tömörítési képességgel rendelkezik a JFFS2 is, amely az egyik legszélesebb körben használt Linuxos flash fájlrendszer. A JFFS2 tömörítő algoritmusként a zlib-et használja, ahogy a UNIX/Linux alatt rendkívül népszerű gzip tömörítő program is. A zlib egy általános célú tömörítő.

Az előző részhez hasonlóan itt is általánosan elmondható, hogy a tömörítés annál hatékonyabb, minél több előismeret, háttértudás áll rendelkezésre a tömörítő algoritmus számára a tömörítendő adatról. Emiatt egy-egy területre specializált tömörítő az adott területen hatékonyabb tömörítést tud végezni, mint egy általános célú tömörítő. Mivel a beágyazott rendszeres környezetekben az egyik legnagyobb arányú "adatfajta" a futtatható kód, és ezek közül pedig az egyik legnépszerűbb architektúra

az ARM, ezért kutatási területként azt céloztuk meg, hogy hogyan lehet hatékony ARM kódtömörítőt fejleszteni és olyan módon implementálni, hogy az jól kombinálható legyen a jelenlegi megoldásokkal.

Az algoritmusunk egy modell alapú tömörítő, amely modellként döntési fát használ arra, hogy a következő token valószínűségeloszlását megjósolja. Ezt a valószínűségeloszlást felhasználva aritmetikai kódoló segítésével hajtjuk végre a kódolást/dekódolást.

A módszerünk alapelgondolása C. W. Fraser [1] és M. Garofal [2] munkásságából indul ki, azokat kombinálja és fejleszti tovább hatékony ARM kódtömörítést elérve. Fraser munkájában gépi kód köztes reprezentációjához (intermediate representation) épít döntési fát a következő token valószínűségének megjóslásához. 0.19-es tömörítési arányt ér el, de ebbe nem számolja bele a modell méretét, amely igen nagy lehet. Garofal munkájában a megépített teljes fát MDL (Minimal Description Length) alapon vágja, így téve hatékonyabbá a tömörítést, és a modell méretét is figyelembe veszi.

Ezekből kiindulva algoritmusunk szintén döntési fát épít, és MDL elvű heurisztikát használ, azonban az építés és tömörítés módját a következő helyeken alakítottuk át, illetve fejlesztettük tovább:

- A tokeneket úgy alakítottuk ki, hogy az ARM gépi kód esetében hatékony legyen, azaz a 32 bites ARM utasításokat (döntési fa által jól kezelhető) 4 bites darabokra vágja, és utasításon belül át is rendezi a sorrendjét a könnyebb "jósolhatóság" miatt.

- Prediktorként 17 prediktort használunk:

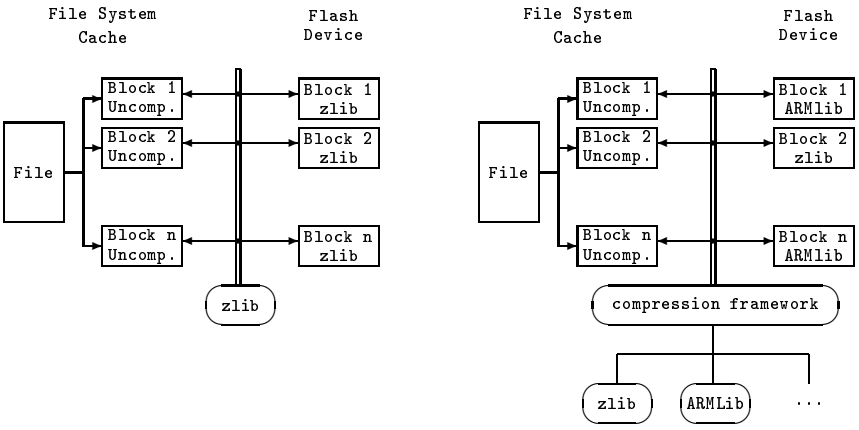
  - 16 redukált prediktort: az előző 16 tokent, azaz 2 ARM utasítást
  - 1 számított prediktort: ez mutatja meg, hogy a következő token az adott ARM utasításon belül hanyadik token (1-8)

- Modellként kétféle döntési fát tud használni az algoritmus különböző teljesítmény mutatókkal: bináris és többértékű fát. Mindkettő esetében lehetőség van a teljes kódbázisra egy fát építeni (ez esetben 1 nagy fa lesz a modell), vagy a kódbázist azonos méretű véletlen halmazokra szétbontva több fát építeni (ez esetben több kisebb fa lesz a modell). A megfelelő kombináció kiválasztásával a rendszer sebesség/méret aránya hangolható.

- A modell építése közben MDL alapú megállási heurisztikát használunk (ellentétben Garofall módszerével, amely csak utólag vágja a fát): a részfa által tömörített kód méretének becslésére entrópiát használunk (mivel az aritmetikai kódoló hatékonysága megközelíti azt), a részfa letárolásánál pedig az adatszerkezet méretét beszorozzuk egy előre definiált tömörítési aránnyal - tekintve, hogy a modell majd egy általános célú tömörítővel még tömörítve lesz.

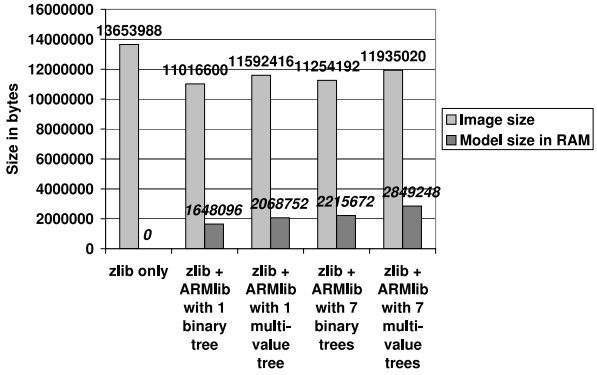
A kifejlesztett implementációt ARMLib-nek neveztük el, amely mellett az algoritmus hatékony gyakorlati alkalmazását is külön ki kellett dolgoznunk. A JFFS2 fájlrendszerbe implementáltuk egy új tömörítési keretrendszer bevezetésével, amely az addigi általános célú zlib algoritmussal kombinálja. Ennek szerkezetét a 3. ábra mutatja be.

Ez a megoldás az eredeti zlib tömörítőt (mellyel a JFFS2 minden adatot 4K-s blokkokra darabolva tömörített) kicseréli egy tömörítési keretrendszerrel, amelynek működési módja konfigurálható. Ha minimális méretre való optimalizálásra van beállítva, akkor betömörítéskor meghívja az összes elérhető tömörítőalgoritmust, és azzal tároltatja el, amely a legkisebb eredményt szolgáltatta. Minden tömörített



a: eredeti **b:** továbbfejlesztett

3. ábra. A JFFS2 tömörítési módszere



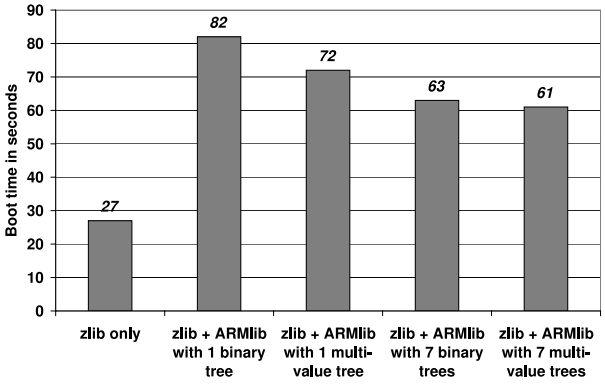
4. ábra. Tömörített image méretek

blokk mellé eltárolja azt is, hogy melyik algoritmussal lett betömörítve, így a kitömörítéskor tudni fogja melyikkel kell dolgoznia.

Ezzel a kombinált megoldással iPAQ H3600-as kézi számítógépen elért eredmények a 4. ábrán láthatóak, amelyek 12.6-19.3%-kal jobbak az eredeti csak zlib megoldáshoz képest. A módszer használatának legnagyobb hátulütője a sebességcsökkenés. A gyakorlatban ennek mértékeként leginkább a boot (bekapcsolási) idő növekedése tükrözi, amely a 5. ábrán látható.

Az így megvalósított keretrendszer azóta a JFFS2 fájlrendszer és a Linux kernel hivatalos része lett. A tömörítési algoritmus a 6,917,315-os számú USA szabadalomban [12] le lett védve.

A tömörítési algoritmus Gergely Tamás szerzőtársammal oszthatatlan eredményünk [3]. A keretrendszer és a Linux környezetbe való hatékony integrálásának elve és gyakorlata önálló eredményem. Az USA szabadalom [12] rajtam kívül Gergely Tamás és Beszédes Árpád nevén van, haszonélvezője pedig a Nokia Corporation.



5. ábra. Boot (bekapcsolási) idők

3. Flash fájlrendszer teljesítményjavítása továbbfejlesztett B+ fával

A beágyazott rendszerek túlnyomó többsége flash chipet használ háttértárként, alacsony fogyasztása és robosztussága miatt (nincs benne mozgó alkatrész). Azokon az alkalmazási területeken, ahol az adatok hatékony tárolása is szükséges módosítási lehetőséggel, ott az adatok tárolására fájlrendszert használnak.

A legtöbb fájlrendszert azonban hagyományos adathordozókra, azaz merevlemezekre tervezték, viszont a flash chipek néhány tulajdonságukban jelentősen különböznek a merevlemezektől. Az adatokat mindkét esetben blokkonként tudják felülírni (a merevlemezek esetében ezeket szektornak hívják és 512 bájt méretűek, a flashek esetében törlési blokknak nevezik, amelyeknek tipikus mérete 128KB), azonban a flash chipek esetében a régi tartalom törlésének van egy korlátja: egy törlési blokk kb. 100.000-szer törölhető garantáltan, utána a chip működése megbízhatatlanná válik.

Mindez azért jelent problémát, mert a legtöbb hagyományos fájlrendszer esetében vannak kitüntetett területek (pl. superblock, FAT, ...), amelyek rendkívül gyakran változnak. Ha ezeket egy az egyben flash-en tárolnánk, akkor (amellet, hogy lassú lenne a törlés művelet lassúsága miatt) a flash chip azon része nagyon hamar tönkre is menne.

Erre a problémára az egyik megoldás a Flash Translation Layer, vagy röviden FTL, amely egy köztes logikai réteget beillesztve "csereberéli" a fizikai törlési blokkokat használat közben, azért, hogy a terhelés egyenletes legyen. A legtöbb pendrive-on vagy memóriakártyán ezt a megoldást használják.

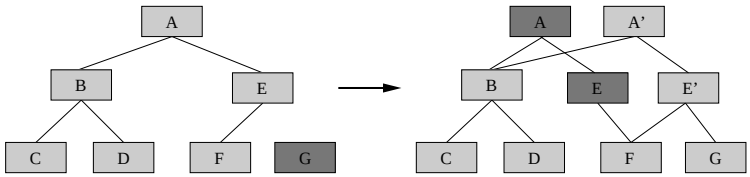
Az FTL alkalomszerű használatra egy megfelelő kompromisszum. Állandó használatra azonban, ahol az ilyen szintű "kopás" sem megengedhető, illetve a teljesítményvesztés is fontos, más megoldást kell találni. Egyik ilyen megoldás a flash fájlrendszerek használata, amelyek kifejezetten a flash chipek működésének ismeretében lettek tervezve.

Az egyik legelterjedtebb Linux alapú flash fájlrendszer a JFFS második verziója, a JFFS2 volt. A fájlrendszer (leegyszerűsített) alapgondolata az, hogy ne legyenek a flashen különleges, gyakori használatú területek, hanem legyen az egész tárterület egyetlen hatalmas napló. Ebbe a naplóba a fájlrendszer minden változást (pl. fájl törlés, írás, ...) beleír egy naplóbejegyzés formájában, és semmi-

lyen indexinformációt (amit dinamikusan módosítani kellene) a flashen nem tárol. Az indexinformáció a memóriában kerül összeállításra a fájlrendszer felcsatolásakor a teljes flash tartalom felolvasásával, és az összes naplóbejegyzés visszajátszásával. Ha új szabad blokkra van szükség, akkor a rendszer kiválasztja azt, amelyikben a legtöbb, már elavult naplóbejegyzés található, majd a még élő bejegyzéseket átpakolja, és törli a blokkot. Ezzel a megoldással csak annyi blokkot fog törölni a fájlrendszer, amennyi feltételen szükséges. Ennek a megoldásnak az ára azonban a felcsatolási idő lassúsága, és az, hogy a teljes indexinformációt a memóriában kell tárolni. Ez különösen nagy flash chipek esetén jelent problémát, ami miatt a JFFS2 tulajdonképpen 512MB fölötti flash chippeken gyakorlatilag használhatatlanná válik.

Mivel a gond a fájlrendszer alapjaiban volt, ezt megoldani csak új fájlrendszer tervezésével volt lehetséges. Az új fájlrendszer alapgondolata természetes módon az lett, hogy az indexinformációt a flashen (is) tároljuk el. A tárolásnak viszont olyan módját kellett kidolgozni, amely egyszerre flash-barát, és hatékony. Sok fájlrendszer használ B+ fa algoritmust indexének tárolására, így mi is ennek javítását céloztuk meg.

A LogFS fájlrendszer adatszerkezete ezt a problémát részben megoldotta a wandering tree (táncoló fa) algoritmus bevezetésével. Ez az algoritmus a klasszikus B+ fához nagyon hasonlóan működik, egy lényeges különbséggel, amely az új csomópont beszúrásának módja. Ezt a 6. ábrán keresztül lehet a legkönnyebben bemutatni.

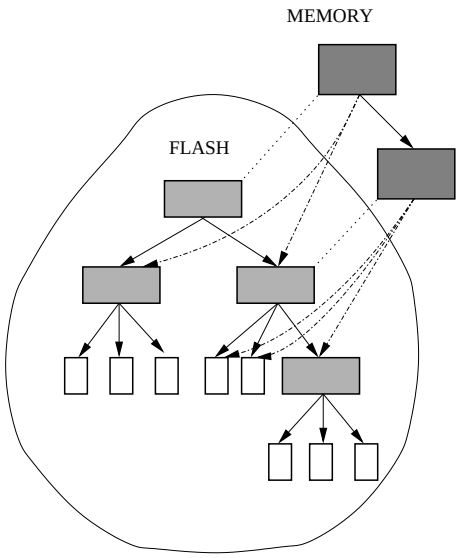


6. ábra. Wandering tree a beszúrás művelet előtt és után

Az ábrán látható fába a G csomópontot szeretnénk beszúrni. Hagyományos esetben az E köztes csomópontban lévő egyik pointert kellene csak módosítani hozzá, azonban ha ez flash-en van, akkor nem tudjuk olyan büntetlenül átírni, mint merevlemez esetében. Ezért nem írjuk át, hanem létrehozunk egy új E' csomópontot, amelyben már a G-re mutató pointer is benne van. Mivel azonban változott a fa köztes csomópontjának a helye, ezért a fa gyökerét, az A csomópontot is módosítani kellene. Ezt hasonló módon tesszük meg, azaz létrehozunk egy új A' csomópontot, amely így a fa új gyökere lesz. Törlésre és módosításra nem volt szükség, viszont keletkezett két "halott" csomópontunk. Ha ezt az algoritmust "után" alkalmazzuk, akkor a keletkező sok szemét olyan mértékben teszi hatékonytalanná az adattárolást, hogy az a gyakorlatban használhatatlan lesz. Így tehát önmagában a wandering tree sem kellően hatékony. Ennek hatékonyabbá tétele a tézis fő eredménye.

A továbbfejlesztésként kidolgozott adatszerkezet (Tree Node Cache, röviden TNC) fő sémája a 7. ábrán látható. Adatszerkezete kezeli a háttértáron (flashen) és memóriában lévő elemeket is. Építése és módosítása dinamikusan történik, egyszerre ellát cache-elő, teljesítményjavító és hibakezelő funkciókat is. Főbb tulajdonságai a B+ fához képest:

- A TNC egy része a memóriában, egy része a flash-en tárolódik. Minden memóriában lévő csomópontra igaz az, hogy az összes őse is a memóriában van, minden flashen tárolt csomópontra pedig az, hogy minden leszármazottja a flashen van.



7. ábra. A TNC adatszerkezete

- Minden memóriában lévő csomópont tárolja azt, hogy a flashről honnan lett felolvasva, és hogy azóta ott módosítva lett-e vagy sem.
- Műveletei a következőek:

Keresés (olvasás):

1. A fa gyökerének beolvasása a memóriába, a p pointer erre a területre állítása
2. Ha p a keresett elem, akkor p értékével visszatérés
3. A p által mutatott csomópontban azon gyermek csomópont (részfa) megkeresése, amelyben a keresett elem van.
4. Ha ez a gyermek csomópont már a memóriában van, p átállítása ennek a címére, és a 2-es pontra ugrás
5. A gyermek csomópont beolvasása a flashről a memóriába, és a p pointer által mutatott csomópont módosítása úgy, hogy ezen gyermekcsomópont esetében a memóriába beolvasott csomópontra mutasson.
6. p pointer átállítása erre a csomópontra és a 2-es pontra ugrás.

Cache felszabadítás (pl. ha kevés a szabad memória):

1. Egy olyan index-csomópont keresése a memóriában, amely nem módosult a flashről beolvasás óta, és ez az összes gyermek-csomópontja a flash-en van. Ha ilyen nincs, akkor kilépés.
2. A csomópont szülőjében az erre a csomópontra mutató pointerének átállítása ennek a csomópontnak a flash területére, és ennek a csomópontnak a memóriából történő felszabadítása.

3. Ugrás az 1-es pontra, ha további memória felszabadítás szükséges.

Beszúrás (írás):

1. Az adat blokkok azonnali kírása a flash-re levél csomópontként. Az UBIFS ezeket egy BUD területnek[1] nevezett blokkban tárolja, ami kifejezetten levél csomópontok számára van fönntartva, azért, hogy megkönnyítse a hirtelen áramkimaradás miatti leállás utáni helyreállítást.
2. Az összes olyan csomópont keresése/beolvasása a memóriába, amely a B+ fa algoritmusa szerint módosításra szorul. (A legtöbb esetben ez egyetlen index csomópont.)
3. A B+ fa módosítási műveletek végrehajtása a memóriában.
4. Az összes módosított csomópont megjelölése módosítottként.

A fenti algoritmussal a beszúrás műveletek "összevárhatóak" a memóriában, így együttesen végrehajtva jelentősen kevesebb flash művelet szükséges, és kevesebb "szemét" keletkezik.

Commit (a módosított cache kiírása):

1. Egy olyan módosított index csomópont keresése, amelynek nincsen módosított gyermek csomópontja. Ha ilyen található, jelöljük n-nel.
2. n kiírása egy új csomópontként a flashre, ahová a gyerekcsomópontok címeinél azok flashen lévő címe kerül kiírásra.
3. Az n csomópont régi flash-en lévő helyének szemétként megjelölése, és n memóriabeli változatában a tárolt flash cím frissítése az új flash címre.
4. n ősét megjelölni módosítottként, n-et pedig nem módosultként.
5. Ugrás a 1-es pontra, amíg van módosított csomópont.

Törlés:

1. Az összes olyan csomópont keresése/beolvasása a memóriába, amely a B+ fa algoritmusa szerint módosításra szorul. (A legtöbb esetben ez egyetlen index csomópont.)
2. A B+ fa módosítási műveletek végrehajtása a memóriában.
3. Az összes módosított csomópont megjelölése módosítottként.

Hirtelen áramkimaradás esetén a memóriában tárolt információk elvesznek. Az UBIFS azért, hogy az ebből eredő információvesztést kiküszöbölje, a TNC-t egy naplóval kombinálja, ahol a következő információk tárolódnak:

[1] Kétféle adatot tároló törlési blokk létezik UBIFS-ben: BUD törlési blokk és nem-BUD törlés blokk. Az UBIFS csak a levél csomópontokat tárolja a BUD törlési blokkban, az összes többi típusú adat csomópontot a nem-BUD törlési blokkban.

- Minden új BUD törlési blokkra mutat egy naplóbejegyzés. A BUD törlési blokkok UBIFS-ben azok a törlési blokkok, amelyek a B+ fa leveleinek a tárolására vannak fönntartva. Ha az aktuális BUD terület megtelt, akkor egy új, szabad törlési blokk kerül lefoglalásra - erről kerül a bejegyzés a naplóba.
- Törlési bejegyzés minden csomópont törléséről.
- Commit bejegyzés minden commit után egy listával az aktuálisan aktív BUD törlési blokkokról.

Áramkimaradás esetén a TNC fa helyreállítható a következő lépésekkel:

1. Induljunk ki a flashen eltárolt fa változatból.
2. Keressük meg a naplóban a legutolsó commit bejegyzést. Az összes ez utáni bejegyzést a naplóban újra "le kell játszani".
3. Az összes csomópontot, amelyeket a BUD törlési blokkokban találunk, újra be kell szúrni, és az összes törlést amely naplózva lett, újra le kell játszani a memóriában, az események megfelelő sorrendjében.

A TNC hatékonyságát a következő módon vizsgáltuk: egy üres 512MB-os fájlrendszeren a Linux kernel 2.6.31.4-es verzióját kitömörítettük, majd töröltük. Eközben mértük, hogy a flashen hány művelet kerül végrehajtásra (blokk méret egységben) TNC használata nélkül, illetve annak használatával a következő paramétereit variálva:

TNC buffer méret : A maximális mérete a TNC-nek, amelyet a memóriában használhat. Ha ez betelne, meghívja a shrink műveletet, amely a commit és cache felszabadítás műveletek együttese.

Shrink arány : Shrink művelet esetén ez mondja meg, hogy hány százalékát szabadítsa föl a rendszer a TNC buffer méretének.

Fanout : B+ fa csomópont gyermekeinek a maximális száma

Az eredmények a 5. táblázat és a 8. ábrán láthatóak. Látható, hogy a TNC a flash műveletek 98.2-99.4%-át megspórolja az eredeti wandering tree algoritmushoz képest. A TNC buffer méretének növekedésével a spórolási arány javul, míg a shrink aránynak nincs érdemleges hatása.

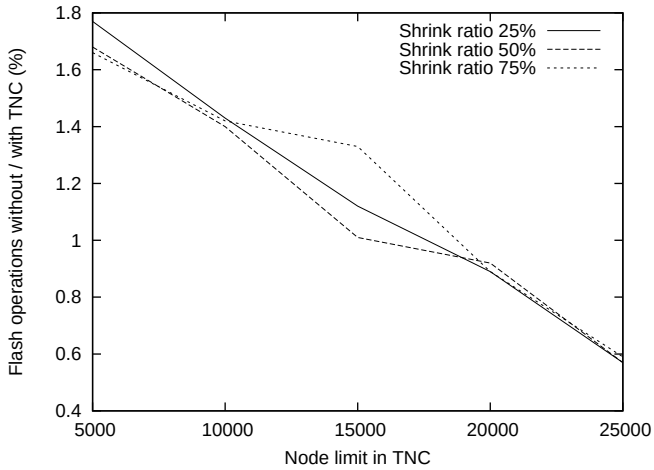
A 6. táblázat azt mutatja, hogy mi történik ha a fanout (maximális gyerekszám) paraméterét változtatjuk a TNC-nek. Ennek növelésével a TNC csomópontok száma csökken, ugyanakkor a csomópont mérete nő, hiszen több pointer és kulcs található benne. A méret e kettő szorzata, amelynek optimuma 32-es értéknél található a mi esetünkben.

Az 7. táblázat és a hozzá tartozó 9. ábra pedig a 2.6.31.4-es kernel forrásának különböző méretű részeivel történt mérés eredményét mutatja, amely a maximális TNC méretet (előre beállított határ nélkül futtatva) jeleníti meg a fanout és tárolt adat függvényében.

A kapcsolódó irodalomban [10] szerzői egy, a miénkhez hasonló célú módszert dolgoztak ki, azaz ők is a B+ fa optimalizálását célozták meg flash eszközön. A módszerük összevárja az összes változtatást a memóriában (a LUP-ban, azaz Lasy Update Pool-ban), és miután ez betelt, az adat csomópontok csoportosan kerülnek kiírásra. Ez az írási műveletenknél (a TNC-hez hasonlóan) időt spórol, de (a

Max. TNC méret	TNC nélkül	TNC-vel	Shrink Ratio	TNC-vel / TNC nélkül
5000	2161091	38298	25 %	1.77 %
10000	2211627	31623	25 %	1.43 %
15000	2191395	24632	25 %	1.12 %
20000	2244013	20010	25 %	0.89 %
25000	2192044	12492	25 %	0.57 %
5000	2163769	36273	50 %	1.68 %
10000	2250872	31570	50 %	1.40 %
15000	2225334	22583	50 %	1.01 %
20000	2225334	20002	50 %	0.92 %
25000	2183596	12457	50 %	0.57 %
5000	2215993	36759	75 %	1.66 %
10000	2290769	32578	75 %	1.42 %
15000	2244385	29956	75 %	1.33 %
20000	2238633	20002	75 %	0.89 %
25000	2205709	12958	75 %	0.59 %

5. táblázat. Flash műveletek száma (csomópont méretben)



8. ábra. TNC teljesítményének összehasonlítása a wandering tree algoritmuséval

TNC-vel ellentétben) nemhogy nem gyorsítja, hanem lassítja az olvasási műveleteket, mert a LUP-ot minden fában-keresési műveletnél át kell nézni. A TNC viszont általában gyorsítja az olvasási műveleteket is, mert a csomópontok egy része (ha más nem is, az időközben módosultak biztosan) a memóriában vannak gyorsan kereshető formában. A TNC jól viseli a hirtelen áramkimaradás okozta esteket, míg a [10] írói nem tárgyalják ezt a kérdést. Módszerüknek viszont az az előnye a TNC-vel szemben, hogy módosított csomópontokat szabadabban csoportosíthatják (nem csak szekvenciálisan), így könnyebben (és akár kevesebb memóriaigénnyel) vonhatják össze azokat a műveleteket, amelyek ugyanarra a fa területre vonatkoznak.

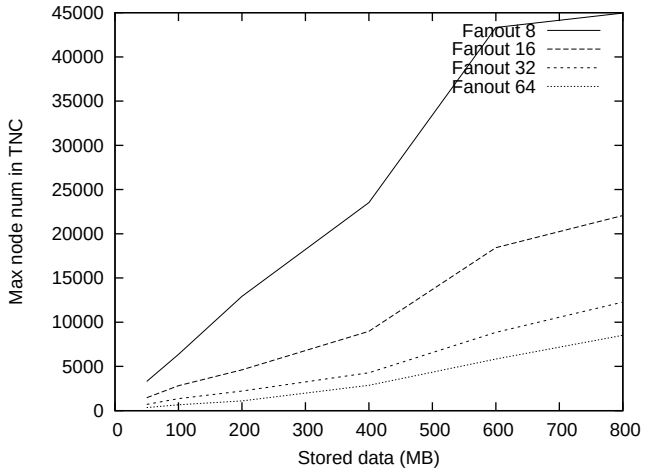
A [14] célja is B+ fa optimalizációja volt flash memórián. Ez a módszer is a memóriába gyűjti össze a változtatásokat az általa "Reservation Buffer"-nek nevezett területen. Amikor ez a terület betelik,

Fanout	Without TNC in nodes	With TNC in nodes	Max TNC in nodes	TNC node size	Max TNC in MB	Flash ops in MB
4	1134784	48392	64801	176	10.88	8.12
8	2168308	12405	23189	304	6.72	3.6
16	1304212	3577	9662	560	5.16	1.91
32	1024363	1317	4669	1072	4.77	1.35
64	1140118	3420	3671	2096	7.34	2.35
128	767005	1245	1586	4144	6.27	3.35
256	930236	1641	980	8240	7.7	4.35

6. táblázat. TNC fanout paraméter módosításának hatása

I/O Size (MB) \ Fanout	8	16	32	64
50	3302	1456	703	351
100	6364	2818	1355	671
200	12925	4620	2224	1106
400	23518	8978	4282	2861
600	43320	18426	8846	5840
800	44948	22070	12273	8527

7. táblázat. Maximális TNC méret a fanout függvényében



9. ábra. Maximális TNC méret a fanout függvényében

akkor kiírja azt egy "Index Unit"-ként. A szerzők bevezettek egy másik adatstruktúrát is, amit "Node translation table"-nek neveztek el, amely azt írja le, hogy melyik csomópont lett az adott Index Unit által módosítva. Ez azt jelenti, hogy amikor majd keresni akar egy csomópontot a fában, akkor a Node translation table-t és a kapcsolódó Index Unitot is át kell majd vizsgálnia.

A [7]-ban leírt módszer a [14] továbbfejlesztett változata. A "Reservation Buffer" helyett "Index Buffer"-t használ, amely nyomon követi a fa változtatásait, és ha ugyanazt a csomópontot érintik, amelyet korábban is, akkor a korábbit lezárja, vagy törli. Commit művelet esetében ezeket összevonva

egy lapra írja ki.

Az eredmények önálló eredményeim és a [5]-ban kerültek publikálásra. A TNC az UBIFS és a Linux kernel hivatalos részévé vált az általam vezetett tanszéki csapatnak köszönhetően, és felhasználásra került a Nokia N900 okostelefonban is.

Hivatkozások

[1] Christopher W. Fraser. Automatic inference of models for statistical code compression. *SIGPLAN Not.*, 34(5):242–246, May 1999.

[2] Minos Garofalakis, Dongjoon Hyun, Rajeev Rastogi, and Kyuseok Shim. Efficient algorithms for constructing decision trees with constraints. In *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '00, pages 335–339, New York, NY, USA, 2000. ACM.

[3] T. Gergely, F. Havasi, and T. Gyimóthy. Binary code compression based on decision trees. *Proceedings of the Estonian Academy of Sciences Engineering*, 11(4):269–285, 2005.

[4] T. Gyimóthy and T. Horváth. Learning semantic functions of attribute grammars. *Nordic Journal of Computing*, 4(3):287–302, Fall 1997.

[5] F. Havasi. An improved B+ tree for flash file systems. In *SOFSEM 2011: Theory and Practice of Computer Science: 37th Conference on Current Trends in Theory and Practice of Computer Scienece, LNCS 6543*, pages 297–307, 2011.

[6] Ferenc Havasi. XML semantics extension. *Acta Cybernetica*, 15(2):509–528, 2002.

[7] Ha-Joo Song Hyun-Seob Lee, Sangwon Park and Dong-Ho Lee. An efficient buffer management scheme for implementing a B-Tree on NAND flash memory. *Embedded Software and Systems*, pages 181–192, 2007.

[8] M. Kálmán, F. Havasi, and T. Gyimóthy. Compacting XML documents. *Information and Software Technology (Impact Factor 1.522)*, 48(2):90 – 106, 2006.

[9] Miklós Kálmán and Ferenc Havasi. Enhanced XML validation using SRML. *International Journal of Web & Semantic Technology (IJWesT)*, 4(4):1 – 18, 2013.

[10] Sai Tung On, Haibo Hu, Yu Li, and Jianliang Xu. Lazy-update B+-Tree for flash devices. *Mobile Data Management, IEEE International Conference on Mobile Data Management*, pages 323–328, 2009.

[11] G. Psaila and S. Crespi-Reghizzi. Adding Semantics to XML. In D. Parigot and M. Mernik, editors, *Second Workshop on Attribute Grammars and their Applications, WAGA'99*, pages 113–132, Amsterdam, The Netherlands, 1999. INRIA rocquencourt.

[12] Á. Beszedes T. Gergely, F. Havasi. Model based code compression, US patent number 6,917,315, 2003.

[13] J. Turley. The two percent solution. Embedded Systems Design, 2002.

[14] Chin-Hsien Wu, Tei-Wei Kuo, and Li Ping Chang. An efficient B-Tree layer implementation for flash-memory storage systems. *ACM Trans. Embed. Comput. Syst.*, 6(3):19, 2007.