UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

Planificació multinivell per a optimitzar l'ús de l'ample de banda de la jerarquia de memòria en multinuclis

Projecte final de carrera d'Enginyeria en Informàtica
Curs 2010/2011

Departament d'Informàtica de Sistemes i Computadors

Josué Feliu Pérez

Directors:
Julio Sahuquillo Borrás
Salvador V. Petit Martí

Juliol de 2011

Índex

Capítol 1

Introducció

1.1 Marc tecnològic

El mercat dels microprocessadors està dominat en l'actualitat pels processadors amb diversos nuclis o *multicore*. A aquesta situació s'ha arribat com a conseqüència dels problemes de consum, refrigeració i empaquetament que patien els processadors monolítics d'altes prestacions. És ben sabut que en els processadors multicore l'ample de banda de la xarxa que connecta els processadors amb la memòria principal és el principal coll de botella [D. 96], ja que un mateix controlador de memòria sol estar compartit per diversos nuclis. Per descomptat açò complica la seva escalabilitat però no cal descuidar que també el rendiment, inclús amb pocs nuclis, es pot veure reduït.

Les previsions tecnològiques indiquen que en 2017 hi haurà diversos centenars de cores en el mateix xip, i per tant, disposar d'un bon ample de banda per accedir a memòria i fer un bon ús d'ell, s'ha convertit en un punt crític dels sistemes actuals. A més a més, no cal oblidar que actualment les jerarquies de memòria inclouen normalment 3 nivells, i els nivells superiors solen estar compartits per diverses estructures de cau del nivell inferior, apareixent altres punts de contenció en el sistema. És per això que la planificació cal fer-la contemplant tota la jerarquia de memòria incloent les caus i no centrar-se únicament en la contenció per l'accés a la memòria principal.

1.2 Estudis previs

Amb l'objectiu de solucionar o mitigar aquest problema s'han realitzat diversos estudis. La majoria es troben centrats en com reduir l'accés a memòria que requereixen els processos, per exemple tractant de reduir les *prebúsquedes* innecessaries [E. 09]. Aquestes tècniques ofereixen bons resultats reduint l'ample de banda consumit pels processos. Tot i això, és inevitable que alguns processos realitzen molts accessos a la jerarquia de memòria pel tipus de dades que utilitzen o l'ús que realitzen d'elles. Per descomptat, aquest problema s'agreuja amb l'augment del nombre de nuclis dels processadors, ja que el nombre de processos que pretenen realitzar accessos a memòria creix, en principi, proporcionalment amb el nombre de nuclis.

És per això que també cal centrar l'estudi en com el sistema operatiu planifica els processos, per tal de que els requeriments dels accessos a memòria no excedisquen el límit pràctic del bus [C. 03b] [C. 04] [C. 03a], la qual cosa provocaria contenció en

l'accés a la memòria, amb moltes esperes i la lògica pèrdua de prestacions. Si a més a més, el nombre de processos és gran i el nombre de nuclis també, el sistema operatiu pot tenir moltes possibilitats de planificació diferents i és interessant guiar la planificació per a que el rendiment siga el màxim.

El comportament dels programes no és homogeni i aquestos solen combinar períodes de molt d'accés a memòria per llegir o emmagatzemar dades amb períodes on realitzen molt de càlcul sense accedir a memòria. Per tant, és necessària una planificació dinàmica dels processos que ens permeta aproximar l'ample de banda que un procés requerirà en el següent quantum per tal de cercar i llançar a execució amb ell els processos que oferisquen una bona combinació en l'accés a memòria i no penalitzen les prestacions per la contenció.

Els treballs publicats fins ara han arribat fins aquest punt, centrant-se únicament en la contenció en l'accés a memòria. El treball de D. Xu i C. Wu [XW10] és un bon exemple i el planificador que proposen serà implementat i avaluat per analitzar les seves prestacions. No obstant, amb el que hem comentat és evident que cal fer un pas més i emfatitzar la planificació tenint en compte la jerarquia de memòria completa, incloent les memòries caus i la seva estructura.

1.3 Estructura del Treball

La resta del present treball s'organitza com segueix. El Capítol 2 explica que són els monitors de prestacions i la PMU (Process Managment Unit) del processadors moderns. Comentarem alguns aspectes del programa *pfmon2* i la llibreria *libpfm* que ens permeten accedir a aquestos comptadors de prestacions. Per últim explicarem els passos per a instal·lar aquest software sobre Fedora Core 10. El Capítol 3 descriu la màquina que hem utilitzat per a realitzar els experiments i l'avaluació. Comentarem el tipus de processador de que disposa, les característiques destacables per al nostre treball i com aquestes afecten a la planificació. El Capítol 4 caracteritza les SPEC 2006, de forma semblant a com es fa en [T. 07], calculant l'IPC de cada benchmark sobre la nostra màquina i mesurant les fallades de la jerarquia de memòria, que ens serviran per a poder analitzar el funcionament del planificador. En el Capítol 5 discutim la proposta de planificació per evitar la contenció en l'accés a memòria [XW10] i expliquem els aspectes més interessants del planificador, tant a nivell de planificació com a nivell d'implementació. El Capítol 6 discuteix la nostra proposta per al planificador, que a partir de la base que veiem al Capítol 5, estén el planificador per a que tracte d'optimitzar l'ús de l'ample de banda de tota la jerarquia de memòria i no únicament de l'accés a memòria. El Capítol 7 descriu la metodologia utilitzada per a l'avaluació dels planificadors. Explicarem com utilitzarem els *benchmarks*, les càrregues de treball que crearem i els paràmeters del planificador que avaluarem. També detallarem com obtenim els temps de planificació per a les nosters càrregues sobre el planificador de Linux. Per últim, en el Capítol 8 mostrem les conclusions a les que hem arribat amb aquest treball i descrivim el treball que ens quedarà per realitzar en pròxims estudis.

Capítol 2

Monitors de prestacions

La complexitat dels sistemes informàtics s'ha incrementat tremendament al llarg de la seva evolució. Els subsistemes de cau jeràrquics, el llançament fora d'ordre o els processadors *multithread* han tingut un important impacte en el rendiment dels sistemes i en la seva capacitat de còmput, però a la vegada també s'ha incrementat la seva complexitat i les seves capacitats de configuració i utilització.

Per aquest motiu és interessant comptar amb software que ens permeta monitoritzar les prestacions que està obtenint el sistema, és a dir, oferint informació sobre diferents esdeveniments que tenen lloc durant una execució. Aquesta informació pot ser utilitzada simplement per avaluar el comportament, per exemple, sobre una càrrega donada. Un ús més complex però també més interessant que es pot fer d'aquesta informació és el de tractar d'adaptar el funcionament del sistema a partir d'aquesta informació que ens ofereixen els monitors de prestacions i per exemple, adaptar la freqüència del processador a la càrrega de treball que és té.

La informació de les prestacions pot ser obtinguda de dues formes diferents: instrumentant el codi o obtenint la informació directament del processador. La instrumentació del codi és pot realitzar mitjançant opcions dels compiladors o reescrivint parts del codi però té el problema de que el codi instrumentat és com a mínim lleugerament diferent del codi original que es volia controlar amb la qual cosa es modifica el programa a monitoritzar. A més, no sempre es pot tenir accés al codi de l'aplicació que es pretén monitoritzar per a instrumentar-lo.

Per això és interessant utilitzar monitors de prestacions. Aquestos programes obtenen la informació de les prestacions accedint directament a la microarquitectura i llegint els comptadors hardware de rendiment dels processadors moderns. D'aquesta forma es pot monitoritzar qualsevol programa o càrrega, sense disposar del codi font i sense necessitat de modificar els programes.

La unitat de monitorització del rendiment o PMU (Performance Monitoring Unit) és la que permet recollir esdeveniments del hardware del processador com poden ser esdeveniments sobre el *pipeline*, el bus del sistema o la jerarquia de memòria i comptabilitzar-los en els comptadors hardware que disposen els processadors moderns amb aquest propòsit. Les PMU són molt especifiques de la implementació de cada arquitectura amb la qual cosa ens podem trobar amb esdeveniments, nombre de comptadors i característiques addicionals diferents entre els diversos processadors que trobem en el mercat.

Amb la utilització de la informació que ens ofereixen aquestos monitors de prestacions podem ajustar la freqüència del sistema en funció de la càrrega del sistema com ja hem mencionat prèviament, o realitzar accions més complexes però interessants per a obtenir les millors prestacions com pot ser planificar els processos per a que optimitzen l'accés a memòria.

2.1 Perfmon2

Perfmon2 és un d'aquestos monitors de prestacions que permeten monitoritzar el rendiment dels programes dinàmicament sense necessitat de disposar del codi font dels programes, ja que accedeix a la PMU dels processadors per a obtenir dels comptadors, la informació dels esdeveniments que ens interessa en un moment donat.

És evident que el sistema requereix d'una interfície de nucli ja que la PMU està formada per registres del processador que únicament es poden escriure en el major nivell de privilegi, tot i que en algunes arquitectures, com la X86 o la IA64, la seva lectura es pot realitzar a nivell d'usuari. A més, cal tenir en compte que la PMU pot generar interrupcions que han de ser manejades a nivell de nucli i la monitorització per threads requereix *kernel hooks* per a gestionar els canvis de context i la creació i finalització de *threads*.

Perfmon 2 destaca per la diversitat de models d'ús. Podem variar àmbits de mesura entre el sistema complet, monitorització per thread o d'entorns virtualitzats; àmbits de control entre la mesura a nivell d'usuari o a nivell de nucli; i àmbits de processament per obtenir mesures offline per a optimització manual o optimització guiada per perfils o PGO (Profile-Guided Optimization) i mesures online per a realitzar una optimització dinàmica o DPGO (Dynamic Profile-Guided Optimization).

Cal comentar també que *perfmon2* tracta de convertir-se en un estàndar per a la monitorització del rendiment. Es requereixen ferramentes de monitorització per entendre el comportament del software en els sistemes i tractar de fer-los més eficients i per això fan falta ferramentes portables, flexibles, que suporten el major hardware possible, que siguen interessants per als desenvolupadors de software i que siguen acceptades i integrades en distribucions comercials i tot açò és el que tracta d'aconseguir *perfmon2*.

Perfmon 2 utilitza Linux com a sistema operatiu per la necessitat de crear una comunitat que espente el nou estàndard. A més, el software lliure té altres avantatges com són la seva disponibilitat, la compartició del codi, el suport per a múltiples arquitectures o l'escalabilitat dels sistemes que poden funcionar sobre Linux.

Per últim, cal comentar que a més del programa *perfmon2* s'ofereix també la llibreria *libpfm*, que conté les funcions que utilitza *perfmon2* per a que puguen ser utilitzades directament amb programes escrits en llenguatge C. En la implementació del nostre planificador utilitzarem aquesta llibreria i les funcions que ofereix per anar mesurant diversos comptadors de prestacions dels processos, que ens permetran decidir quins han de ser llançats en el següent quantum per a obtenir un rendiment òptim.

2.2 Instal·lació de perfmon2 sobre Fedora Core 10

La intenció dels desenvolupadors del *perfmon2* és que aquest siga fàcil d'instal·lar i que el sistema operatiu Linux incloga el suport per al nucli en les distribucions comercials. Això no obstant no ocorre actualment ja que es necessari recompilar el nucli per a afegir-li un parche que done suport a aquestes característiques. Després, caldrà instal·lar una llibreria amb les funcions que requereix el programa per a funcionar coneguda com a *libpfm*. Per últim, s'ha d'instal·lar el propi programa, el *perfmon2*. A continuació detallarem els pasos seguits per a instal·lar tot el software necessari per a que el *perfmon2* funcione en Fedora core 10.

El primer que hi ha que fer és instal·lar una sèrie de paquets que són necessaris per a poder modificar el nucli, recompilar-lo i instal·lar-lo. Aquestos paquets s'instal·len amb el següent comandament:

```
su -c 'yum groupinstall '"Development Tools"'
su -c 'yum install ncurses-devel qt-devel unifdef'
```

Després cal descarregar el codi del nucli. Actualment l'últim *kernel* al qual es dona suport *perfmon2* és el 2.6.29 i per tant aquest serà el que utilitzarem. Es pot descarregar de la pàgina www.kernel.org.

El "parche"que hem d'aplicar al nucli és el parche del perfmon corresponent a aquest nucli. El podem trobar dins la carpeta kernel patch de l'arbre de descarregues de la pàgina del perfmon 2. La seva adreça és: http://sourceforge.net/projects/perfmon2/files/kernel-patch/2.6.29/.

Una vegada tenim descarregats tant el nucli com el "parche", els descomprimim i seguim les instruccions del parche per aplicar-lo al nucli. Les instruccions són senzilles i únicament utilitzen la utilitat *patch* de Linux per aplicar les modificacions al nucli:

```
cd "al directori pare del nostre kernel"
cat ../perfmon-new-base-090222/*.diff | patch -p1
```

El següent pas és configurar el nucli per a que incloga el suport per al *perfmon2* activat. Hi ha diverses formes de fer-ho amb entorn gràfic (make xconfig o make gconfig) però nosaltres utilitzarem el menú basat en text:

```
make menuconfig
```

Una vegada oberta la utilitat busquem el suport per al *perfmon2* que es troba dins de Processor type and features -> Hardware performance monitoring support. Ací activem la interfície del *perfmon2*, el suport per a l'arquitectura que anem a utilitzar i guardem els canvis realitzats.

El següent pas és ja compilar el codi font per a obtenir una imatge comprimida del kernel i després compilar els mòduls del kernel:

```
make
make modules
```

Una vegada ho tenim tot compilat procedim a instal·lar-ho amb els següents comandaments:

```
make modules_install
make install
```

Després sols ens queda ja editar el fitxer del GRUB per a arrancar amb el nostre *kernel* i reiniciar l'ordinador:

```
nano /boot/grub/grub.conf
shutdown -r now
```

Una vegada hem reiniciat l'ordinador i ens trobem utilitzant el kernel que hem compilat amb el suport per al *perfmon2*, procedirem a instal·lar la llibreria *libpfm*. L'última versió que hi ha disponible és la *libpfm-3.9* que podem descarregar de la pàgina del *perfmon2* (http://sourceforge.net/projects/perfmon2/files/libpfm/libpfm-3.10.tar.gz/download). Una vegada descarregat el fitxer la seva instal·lació és realitza de la següent forma:

```
tar -xvzf libpfm-3.10.tar.gz
cd libpfm-3.10
make
make install
```

Ja per últim sols ens queda instal·lar el propi programa *perfmon2*. Per a poder instal·lar-lo abans hem d'instal·lar la llibreria *libelf* amb els següents comandaments:

```
yum install elfutils-libelf-devel elfutils-libelf-devel-static
```

L'última versió del *perfmon2* és la 3.9 que podem descarregar de la seva pàgina (http://sourceforge.net/projects/perfmon2/files/pfmon/pfmon-3.9/). La instal·lació es realitzar com la de la llibreria amb els següents comandaments:

```
tar -xvzf pfmon-3.9.tar.gz
cd pfmon-3.9.tar.gz
make
make install
```

Amb tot el que hem realitzat el programa *pefmon2* deuria funcionar amb el seu comandament pfmon. No obstant això, en les instal·lacions que hem realitzat hem trobat problemes de rutes que feien que el programa no fora capaç de trobar la llibreria *libpfm.so.3*. La forma de solucionar-ho és editant el fitxer *libpfm.conf* que indica la ruta en la que es troben algunes de les llibreries que *perfmon2* utilitza i afegir la de la llibreria *libpfm.so.3*. El fitxer *libpfm.so.3* es troba dins de *libpfm-3.10/lib*, on *libpfm-3.10* és la carpeta descomprimida de la llibreria que hem descarregat abans. Els comandaments que hem de realitzar són els següents:

```
cd /etc/ld.so.conf.d/
```

```
nano libpfm.conf    (hem d'incloure la ruta de la llibreria en
   aquest fitxer)
ldconfig
```

Una vegada executat el *ldconfig* deuria estar tot configurat correctament per poder utilitzar el programa sense problemes.

Capítol 3

Arquitectura Intel utilitzada

El sistema que utilitzarem per a realitzar les proves sobre una arquitectura Intel, utilitzarà un processador Intel Xeon X3320 de la sèrie Yorkfield. Es tracta d'un processador llançat el primer quadrimestre de l'any 2008 i que presenta les següents característiques:

Processador	Intel Xeon X3320
Freqüència	2.5 GHz
Nombre de nuclis	4
Multithreading	No
Memòria cau L1	L1 d'instruccions: 4 x 32 KB L1 de dades: 4 x 32 KB
Memòria cau L2	2 x 3 MB compartida de 8 vies

Taula 3.1: Característiques del processador

El processador està format per dos *dual-cores* empaquetats conjuntament amb un procés de fabricació de 45 nm. Disposa, per tant, de 4 cores que funcionen a un freqüència de 2.50 GHz. Pel que respecta a la memòria cau, disposa d'una jerarquia de 2 nivells amb caus L1 i L2. La cau L1 es privada per a cada nucli i presenta una arquitectura Harvard que separa la memòria cau en dues caus, una per a dades i una per a instruccions. Cada nucli compta amb 32 KB de cau L1 per a dades i 32 KB de cau L1 per a instruccions, amb la qual cosa el total de memòria cau L1 del processador és de 256 KB.

La cau L2 presenta una arquitectura unificada on les dades i les instruccions s'emmagatzemen conjuntament. Està formada per dues caus, cadascuna privada a un *dual-core* i, per tant, compartida per els seus dos nuclis, amb una capacitat de 3MB. El processador compta per tant amb 6 MB de cau L2 associativa de 8 vies.

Altres característiques del nostre sistema són la velocitat del *front side bus* de 1333 MHZ i el fet de que el processador no suporte execució *multithread*. Per últim, comentar que el sisistema disposa de 4 GB de memòria principal formats per dos mòduls de 2 GB cadascun.

Aquest processador, i particularment el nivell 2 de cau, que està format per per dues caus L2, cadascuna compartida per dos nuclis, ens permetrà estudiar el comportament dels processos quan comparteixen i quan no comparteixen aquesta cau L2. D'aquesta forma, a més de la degradació de prestacions per la contenció en l'accés a memòria que realitzen els quatre nuclis a través del controlador de memòria, podrem analitzar tant la contenció per l'accés a la cau si dos processos realitzen molts accessos com l'augment de les fallades de la cau de L2 si els dos processos tracten d'emmagatzemar moltes dades en ella.

Una última cosa que pot resultar interessant saber és que el nostre processador identifica els nuclis de manera que P0 i P2 comparteixen una cau L2 i P1 i P3 comparteixen l'altra. Açò pot ser diferent en una instal·lació que utilitze un sistema operatiu o nucli diferent i ho indiquem perquè és important per entendre la planificació que realitzarem al tractar de deixar els processos que realitzen més accessos a L2 en nuclis amb caus diferents.

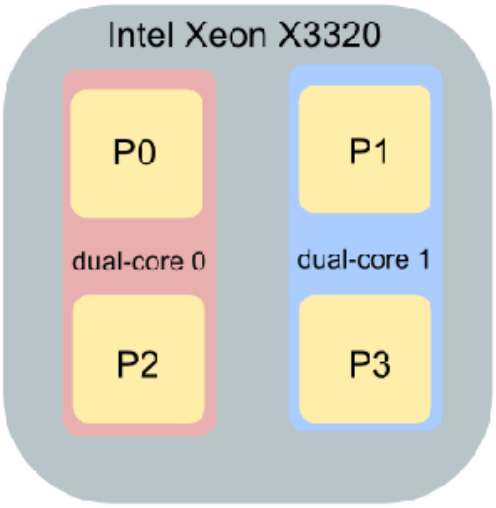


Figura 3.1: Esquema del processador Intel Xeon X3320

Un altre aspecte a comentar del processador és la seva capacitat per a treballar a diferents freqüències. En aquest projecte no treballarem sobre elles, però en futurs treballs si que serà interessant variar la freqüència de treball d'alguns nuclis en funció de les característiques dels programes que executen, amb l'objectiu de disminuir el consum sense afectar en excés al rendiment en l'execució d'una càrrega. Aquest processador permet canviar la freqüència de cada nucli de manera independent, la qual cosa és ideal, ja que molts processadors utilitzen dominis de freqüència que obliguen a que un grup de nuclis treballen a la mateixa freqüència, limitant les possibilitats de configuració i planificació. Per contra, les freqüències de treball disponibles estan molt limitades ja que únicament permet treballar a 2 GHz i 2,5 GHz. Indiquem aquesta informació a nivell informatiu, ja que com hem dit no anem a utilitzar-la en aquest projecte, i en futurs treballs caldrà estudiar amb més detall aquestos aspectes i com afecten a les possibilitats de configuració del processador i les possibilitats de planificació que s'obtenen.

Sobre aquest esquip, hem isntal·lat la distribució Fedora Core 10, amb el nucli 2.6.29 amb el suport del *perfmon2* per a poder accedir als comptadors de prestacions. Com hem comentat, aquest és l'últim nucli al que es dona suport dins del projecte del *perfmon2*. Sobre aquest sistema realitzarem totes les avaluacions dels planificadors.

Capítol 4

SPEC 2006

SPEC 2006 (Standard Performance Evaluation Corporation) és un conjunt de benchmarks que s'han convertit en un estàndar en la indústria. El conjunt de benchmarks que formen la suite SPEC 2006 permet estressar el processador, el subsistema de memòria i el compilador. Està dissenyat per oferir una mesura comparativa del rendiment de sistemes molt variats fent ús de carregues de treball desenvolupades a partir d'aplicacions reals d'usuari. Els benchmarks són proporcionats com a codi font, que permet a l'usuari compilar-los utilitzant les opcions de compilació desitjades.

Els benchmarks es poden dividir en dos grups: el primer està format pels benchmarks d'enters que serveixen per a mesurar i comprar el rendiment de la computació intensiva d'enters; el segon grup està format per benchmarks que utilitzen nombres en coma flotant i permeten mesurar el rendiment quan es treballa amb aquestos nombres. És interessant separar els dos grups de benchmarks ja que els enters tenen un IPC major en general, ja que entre d'altre realitzen menys fallades de cau, a més que presumiblement les instruccions enteres són més simples i la predicció de salts és millor, de forma que requereixen menys cicles que les de coma flotant.

En el nostre cas farem servir els benchmarks per a crear les nostres càrregues de proba combinant diferents benchmarks per avaluar la planificació. Donat que en primer lloc caracteritzarem els benchmarks, podrem crear les combinacions que ens interessen, estressant en major o menor mesura la jerarquia de memòria i utilitzant diferents programes que puguen tenir comportaments i degradacions de prestacions diferents al tenir que compartir l'ample de banda disponible entre ells.

4.1 Caracterització de les SPEC 2006

Per tal de poder planificar els processos de la millor forma possible, el primer pas és caracteritzar els programes que utilitzarem amb la màquina que anem a utilitzar per a realitzar els experiments. D'aquesta forma podrem diferènciar els programes en funció del nombre de fallades que realitzen en cada nivell de la jerarquia de memòria. Aquestes fallades dels programes seran les que utilitzarem en el planificador per guiar-lo i decidir els processos que han d'executar-se conjuntament en un quantum del planificador. En el planificador, les mesures es realitzaran dinàmicament, per tenir una aproximació millor del comportament que tindrà un programa en el següent quantum utilitzant les funcions de la llibreria *libpfm*, però no deixa de ser útil realitzar aquesta primera

caracterització per crear les carregues que ens interesse.

Els paràmetres que estudiarem seran les fallades de cada nivell de la jerarquia de la cau. En aquest cas el processador sols disposa de dos nivells de cau, amb la qual cosa estudiarem les fallades de la cau L1, que correspondran amb els accessos a la cau L2, i les fallades de la cau L2, que correspondran amb els accessos a la memòria principal. Aquestos últims seran la guia principal per a la planificació, ja que són els que presenten una penalització major, per la major latència en l'accés a memòria.

Per a realitzar la caracterització utilitzarem el programa *perfmon2*. La sintaxi per a llançar un procés i mesurar els esdeveniments és la següent:

```
pfmon -e event1[,eventN] programa
```

Els esdeveniments que utilitzarem per a realitzar les mesures que hem descrit són els següents dos. En primer lloc l'esdeveniment LAST_LEVEL_CACHE_MISSES serà el que utilitzarem per a mesurar les fallades de la cau de L2 i per tant els accessos que es realitzen a memòria principal, passant pel controlador de memòria que serà el coll de botella. El segon esdeveniment serà L2_RQSTS, que mesura les peticions o accessos que es realitzen a la cau de L2 i que correspon també a les fallades que realitza la cau del nivell anterior, en aquest cas L1. Aquest segon esdeveniment utilitza una màscara per a concretar més la mesura. Així es podria seleccionar entre les fallades que realitza un nucli del processador, les fallades degudes a prebúsqueda i altres tipus de mesures semblants. En el nostre cas utilitzarem la màscara MESI per a mesurar qualsevol accés que realitze el nostre programa.

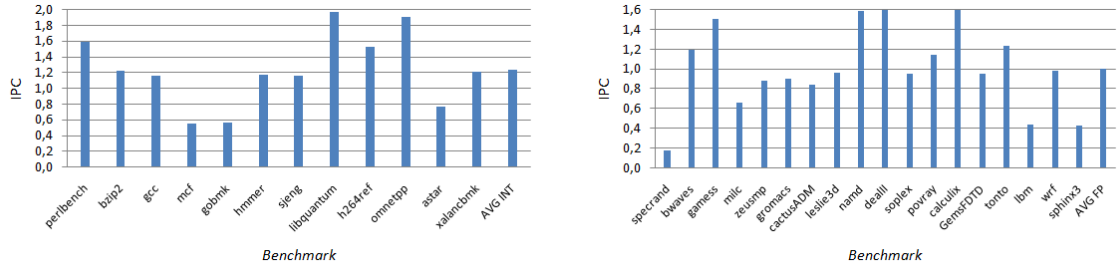
Juntament amb aquestos dos esdeveniments mesurarem també el temps que tarden a executar-se els benchmarks amb l'esdeveniment UNHALTED_CORE_CYCLES. Donat que el processador treballa a 2.5 GHz, sabem que el temps de cicles serà de 0.4 ns i d'aquesta forma podrem obtenir el temps en segons. Per últim mesurarem les instruccions executades amb l'esdeveniment INSTRUCTIONS_RETIRED.

A partir de la informació obtinguda en aquestos esdeveniments calcularem l'IPC, el MPKI de L1 i el MPKI de L2. El IPC (Instructions Per Cycle) és una mesura de les prestacions que obté un benchmark i es calcula dividint les instruccions executades entre els cicles requerits. Quan major és el IPC millor serà el rendiment que aquest *benchmark* ofereix en un sistema concret. El valor de l'IPC pot disminuir per les fallades de la memòria cau (MPKI de L1 o MPKI de L2) o altres esdeveniments com poden ser els salts mal predits.

En la figura 4.1 podem veure el IPC per als *benchmarks* de les SPEC 2006. En la figura 4.1(a) tenim l'IPC per als *benchmrks* enters i en la figura 4.1(b) tenim l'IPC per als *benchmarks* que utilitzen coma flotant.

Podem veure com els *benchmarks* enters tenen un IPC major, sent la seva mitja de 1,23 mentre que la mitja de l'IPC per als *benchmarks* amb coma flotants és de 1. Açò podria ser degut a que realitzen més fallades en la jerarquia de memòria, menor precisió del predictor de salt o simplement a que les instruccions en coma flotant tenen una latència major.

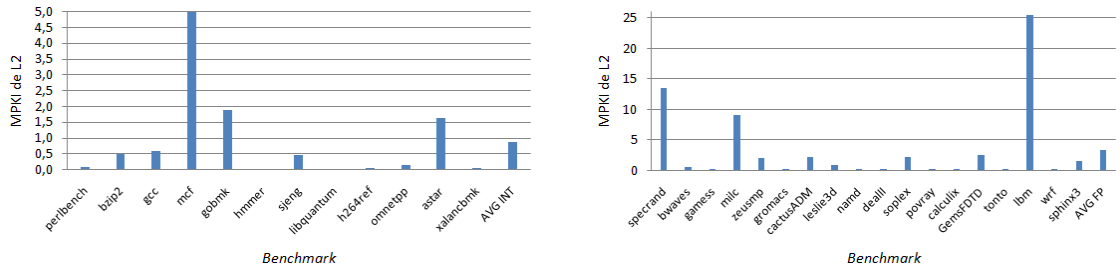
El següent paràmetre que hem comentat era el MPKI (Misses Per Kilo Instruction) de L2, és a dir, les fallades de L2 per cada mil instruccions. Es tracta de les fallades més importants, no per el seu nombre (ja que són menys nombroses que les fallades de L1) sinó per la seva penalització i és que, en cas de fallada en la cau L2, cal accedir a



(a) IPC *benchmarks* enters

(b) IPC *benchmarks* amb coma flotant

Figura 4.1: IPC dels *benchmarks* de les SPEC 2006



(a) MPKI de L2 dels *benchmarks* enters

(b) MPKI de L2 dels *benchmarks* amb coma flotant

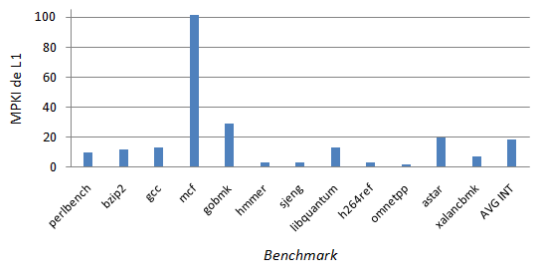
Figura 4.2: MPKI de L2 dels *benchmarks* de les SPEC 2006

la memòria principal que aproximadament és un ordre de magnitud més lenta. Podem aproximar la penalització per accedir a L1 entre 1-2 cicles, la d'accedir a L2 al voltant de 10-12 cicles i la d'accedir a memòria entorn als 100 cicles, d'ací la importància de tenir un nombre de fallades baix per a obtenir les millors prestacions possibles.

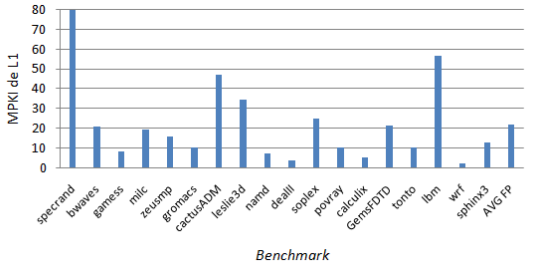
Com hem fet abans mostrarem en la figura 4.2 el MPKI de L2 per als *benchmark* enters 4.2(a) i amb coma flotant 4.2(b) de les SPEC 2006.

Podem apreciar que en general els *benchmarks* en coma flotant tenen un major nombre de fallades que els enters. Com hem comentat aquesta pot ser una de les raons per les que presenten un IPC més baix. De fet, podem veure com *mcf*, *gobmk* i *astar* que presenten el menor IPC entre els *benchmarks* enters són els que també presenten un MPKI de L2 major. Pel que respecta ls *benchmarks* amb coma flotant, que present un MPKI de L2 mitjà superior, destaca *lbm* amb un MPKI de L2 de 25,46 que és el major de tots els *benchmarks* amb diferència.

Per últim, avaluarem el MPKI de L1, és a dir, les fallades de L1 per cada mil instruccions i que correspondran als accessos a L2. Podem veure els resultats en la figura 4.3. Com hem explicat abans la seva penalització en cas de fallada és el temps d'accés a L2 que està al voltat de 10-12 cicles, a la que caldria afegir el temps d'accés a memòria principal en el cas en que es produïra una fallada de L2. És un punt menys crític que les fallades de L2, ja que la penalització és menor i a més també ho és la contenció ja que el nombre de transaccions que es poden realitzar entre L1 i L2 és major



(a) MPKI de L1 dels *benchmarks* enters



(b) MPKI de L1 dels *benchmarks* amb coma flotant

Figura 4.3: MPKI de L1 dels *benchmarks* de les SPEC 2006

que el que es pot realitzar entre L2 i memòria principal. No obstant això, no deixa de ser un punt de contenció que pot reduir les prestacions (si bé en menor mesura) i per tant és important avaluar-lo i tenir en compte al realitzar la planificació.

A més a més, en aquest processador concret solament tenim 2 nuclis compartint la cau L2. En el cas en que el nombre de nuclis fora major i la jerarquia tinguera més nivells, ens podríem trobar amb un major nombre de nuclis compartint una cau, implicant això, una major contenció tant per problemes d'accés com de capacitat. És per tant un tema interessant de tractar i de planificar.

El comportament que observem és semblant al que teníem per al MPKI de L2. Els *benchmarks* en coma flotant realitzen una mitjana de 21,68 fallades de L1 per cada 1000 instruccions com veiem en la figura 4.3(b), respecte a les 18,06 que realitzen els *benchmarks* enters com tenim en la figura 4.3(a). En un principi, no sembla una diferència massa significativa per a les prestacions, tot i que, com hem dit abans, si el nombre de nuclis fora major i compartiren la cau aquestes 4 fallades més podrien acumular-se i augmentar la contenció. Cal destacar també que el *benchmark* que realitza un major nombre de fallades és un *benchmark* enter: *mcf*, que realitza 101,26 fallades en la cau L1 per cada 1000 instruccions.

4.2 Degradació de les prestacions per contenció en l'accés a memòria

Hem comentat ja que el fet de que els programes, o en aquest cas els *benchmarks*, accedisquen molt a memòria principal, després de realitzar fallades en les diferents caus de la jerarquia té una penalització important degut a la latència d'accés a memòria principal. Però el problema s'agreuja quan són diversos processos els que tracten de realitzar molts accessos a memòria principal, ja que aleshores apareix la contenció en l'accés a memòria.

La contenció es produeix quan el controlador de memòria no és capaç de servir als processos totes les peticions que li realitzen, provocant cues de peticions pendents i retards extres als processos si aquestos no són capaços de continuar la seva execució sense les dades requerides. Si el nombre de peticions segueix creixent, el sistema pot

entrar en saturació, la qual cosa implicaria esperes per a totes les peticions de memòria i evidentment una major penalització en les prestacions.

Per a comprovar com es degraden les prestacions el que farem serà executar cadascun dels *benchmarks* conjuntament amb 1,2 i 3 programes *mem-bounded*. Aquestos programes *mem-bounded* l'únic que faran serà realitzar fallades de l'últim nivell de la jerarquia de cau, i per tant faran moltes peticions al controlador de memòria i molts accessos a aquesta, provocant previsiblement contenció en el *bus* principal i degradació de les prestacions al *benchmark* amb el que s'executen. Utilitzarem els programes *mem-bounded* amb una configuració per a que realitzen diferent nombre de fallades. En concret, utilitzarem configuracions per a un MPKI de L2 de 3, que aproximadament és la mitja de fallades que presenten els benchmarks de les SPEC 2006, un MPKI de L2 de 42 que és aproximadament el doble del MPKI de L2 mitjà del *lbm*, el *benchmark* amb major requeriment d'ample de banda entre L2 i memòria principal, i dues configuracions de 20 i 24 fallades per cada mil instruccions com a valors intermedis.

Llistat de codi 4.1: mem-bounded.c

```
1  #include <stdio.h>
2  #include <sys/time.h>
3
4  #define N 4000
5  #define M 256
6  #define ITER 100000
7
8  int main (int argc, char *argv[]) {
9    int i, j;
10   int A[N][M], B[N][M];
11   int nop;
12
13   if (argc != 2) {
14     printf("Error. Us prg1 nops\n", argc);
15     return 1;
16   }
17
18   nop = atoi(argv[1]);
19
20   while (1) {
21     for (j=0; j<N; j++) {
22       A[j][0] = B[j][0];
23     }
24     for (j=0; j<nop; j++) {
25       asm("nop");
26     }
27   }
28
29   return 0;
30 }
```

El codi del programa *mem-bounded* és el que mostrem en el llistat de codi 4.1.

Podem veure com utilitzem instruccions d'assemblador *nop* amb les que regulem el nombre de fallades que el programa *mem-bounded* realitza per cada 1000 instruccions.

El MPKI del *mem-bounded* ha sigut avaluat amb el *perfmon2* i tenim un paràmetre d'entrada que és el nombre de *nops* que realitzarà el programa per cada accés a memòria. En la figura 4.4 podem observar com varia el MPKI de L2 en funció del nombre de *nops* que realitzem.

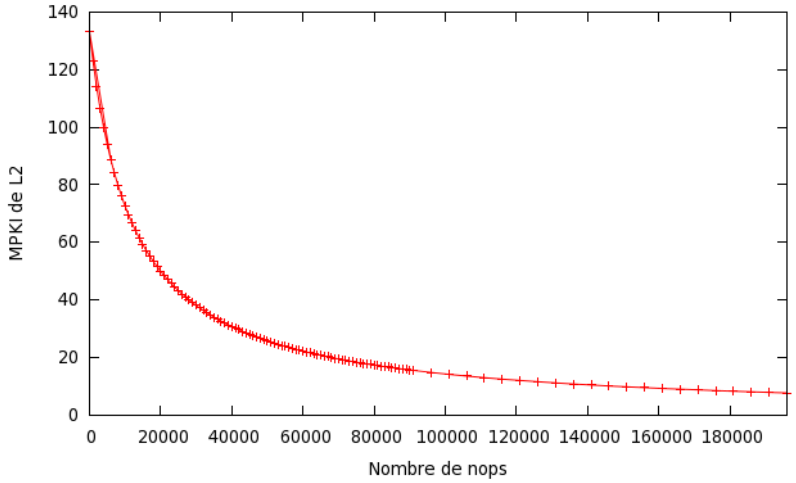
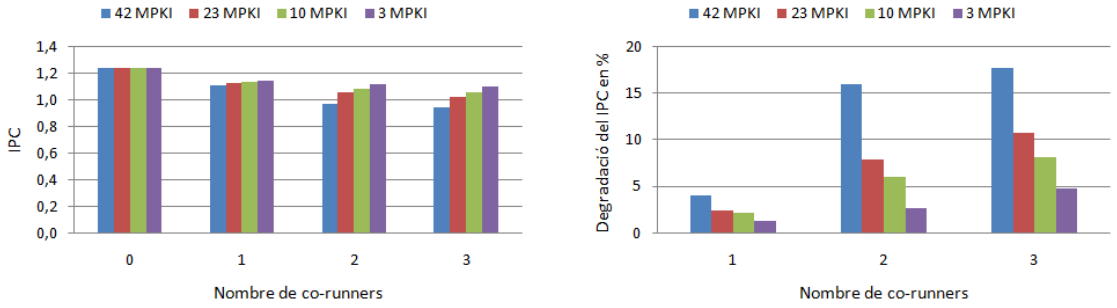


Figura 4.4: MPKI de L2 del programa mem-bounded.c en funció del nombre de nops

El que fem a continuació és executar aquest *mem-bounded* juntament amb els *benchmarks* per analitzar la pèrdua de prestacions. En la figura 4.5 mostrem la degradació de prestacions que sofreixen els *benchamrks* enters. Hem representat en la figura 4.5(a) l'IPC mitjà que obtenen els *benchmarks* enters quan s'executen en solitari i quan s'executen conjuntament amb 1, 2 i 3 instàncies del programa *mem-bounded* amb la configuració per a que presenten el MPKI de L2 que hem indicat abans. Podem observar com l'IPC mitjà és de 1,23 quan s'executen en solitari. Executant-se concurrentment amb un *mem-bounded* amb MPKI de L2 de 42 descendeix a 1,18, quan ho fa amb 2 *mem-boundeds* baixa als 1,04 i amb tres arriba a l'IPC mínim de 1,02 instruccions per cicle. Açò significa una degradació màxima del 18% com podem observar en la figura 4.5(b), on hem representat la degradació del IPC per a les mateixes situacions. En els *mem-boundeds* amb menys requeriments d'ample de banda la degradació de les prestacions és menor però no deixa de ser important, sobretot quan es combina amb 2 *mem-boundeds*, ja que en aquest cas el benchmark comparteix la cau de L2 amb un d'ells provocant el major descens de prestacions. Per últim, executant concurrentment els *benchmarks* amb 3 *mem-boundeds* que tenen un MPKI de L2 de 3, situació que pot ser bastant comú, la degradació que s'obte és del 4.72%.

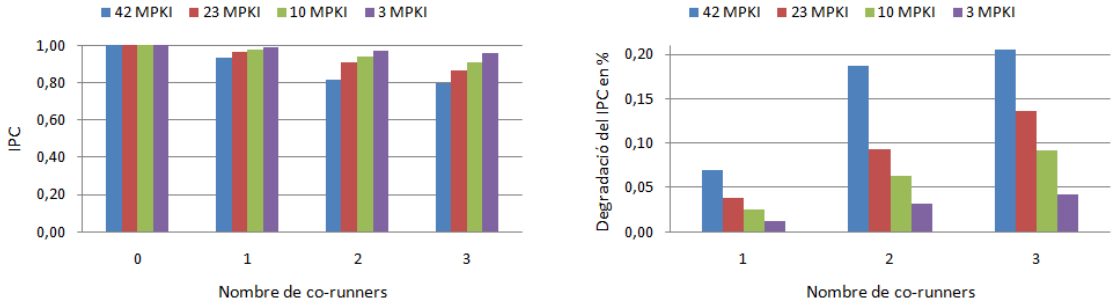
Per als *benchmarks* amb coma flotant s'han realitzat els mateixos experiment, obtenint la degradació que representem en la figura 4.6. S'observa el mateix comportament que amb els *benchmarks* enters però amb una degradació major. Així podem veure en la figura 4.6(a) que l'IPC mitjà per als *benchmarks* amb coma flotant és de 1 i aquest baixa fins a 0,8 quan s'executa concurrentment amb 3 *mem-boundeds* amb un MPKI de L2 de 42. Com veiem en la figura 4.6(b) açò significa una degradació de l'IPC del



(a) IPC mitjà dels *benchmarks* enters

(b) Degradació mitjana de l'IPC dels *benchmarks* enters

Figura 4.5: Valors mitjans per a les execucions dels *benchmakrs* enters concurrentment amb *mem-bounded*s



(a) IPC mitjà dels *benchmarks* amb coma flotant

(b) Degradació mitjana de l'IPC dels *benchmarks* amb coma flotant

Figura 4.6: Valors mitjans per a les execucions dels *benchmakrs* amb coma flotant concurrentment amb *mem-bounded*s

21%, front al 18% dels *benchmarks* enters. D'igual forma, executant concurrentment els benchmarks amb 3 *mem-bounded*s amb un MPKI de L2 de 3, tenim una degradació de l'IPC del 4%, que en aquest cas és inferior a la que obteniem als *benchmarks* enters.

4.3 Degradació de les prestacions per contenció en l'accés la cau L2

La degradació per l'accés a la cau de L2 suposa el mateix problema que hem descrit per a l'accés a memòria però en un nivell superior (més pròxim al processador) dins la jerarquia de cau. Aquest fet provoca que en un principi la contenció siga menys problemàtica, ja que el *bus* admet un nombre molt major de transaccions que el *bus* principal que connecta el controlador de memòria amb la memòria principal. A més, encara que es produïsca la contenció, la penalització serà menor que en la contenció per l'accés a memòria ja que aquesta cau es troba dins del processador i per tant els temps

d'accés són molt menors. No obstant, també es produeixen més transaccions en aquest punt que en l'accés a memòria amb la qual cosa segueix apareguent la contenció.

Per tant, com veurem en els experiments, l' accés a la cau L2 és un punt on el sistema pot provocar contenció, amb penalitzacions i per tant degradació de les prestacions, i tot que aquesta penalitza puga ser menor que en altres punts del sistema és convenient estudiar-la i sempre que siga possible evitar-la per a millorar el rendiment.

De manera semblant a com hem fet abans, hem creat un programa *l2-bounded* que realitza en cada iteració una fallada en la cau L1 i un encert en al cau L2. D'aquesta forma el que aconseguim és un programa que realitza moltes transaccions entre L1 i L2 i pràcticament ninguna entre L2 i memòria, permetent-nos per tant analitzar la contenció provocada per l'accés a L2 sense que la contenció per l'accés a memòria tinga ninguna influència. El codi per aquet programa *l2-bounded* és el que podem veure en el llistat de codi 4.2.

Llistat de codi 4.2: l2-bounded.c

```
1  #include <stdio.h>
2  #include <sys/time.h>
3
4  #define ITER 100000
5  #define N 2000
6  #define M 64  //256
7
8  int main (int argc, char *argv[]) {
9    int i, j;
10   int A[N][M], B[N][M];
11   int nop;
12
13   if (argc != 2) {
14     printf("Error (Numero de arguments = %d). Us prg1 nops\n",
         argc);
15     return 1;
16   }
17   nop = atoi(argv[1]);
18
19   while (1) {
20     for (j=0; j<N; j++) {
21       A[j][0] = B[j][0];
22     }
23     for (j=0; j<nop; j++) {
24       asm("nop");
25     }
26   }
27
28   printf("FI PROGRAMA %d\n", atoi(argv[1]));
29
30   return 0;
31 }
```

De la mateixa forma que fèiem abans, amb un paràmetre que indica el nombre de *nops* que realitzarem per cada fallada de L1 podem controlar el MPKI de L1 que realitza el nostre programa, que com recordem presenta un MPKI de L2 menyspreable. Ho podem veure en la figura 4.7.

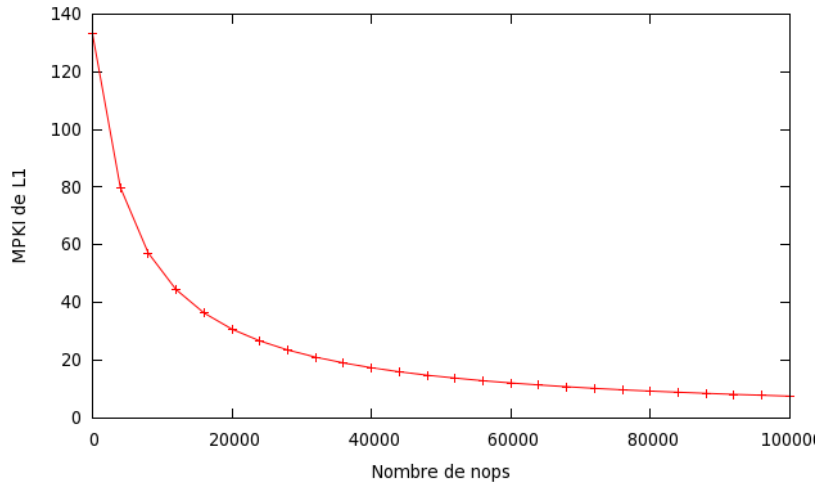


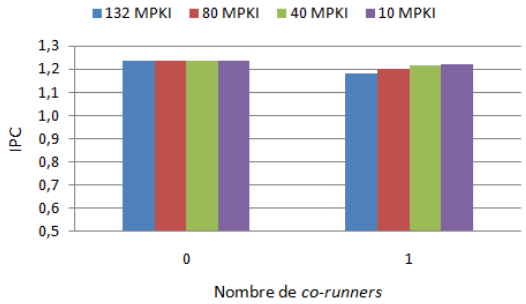
Figura 4.7: MPKI de L1 programa l2-bounded.c en funció del nombre de nops

A continuació s'ha estudiat la degradació que provoca l'execució concurrent dels *benchmarks* amb aquest programa *l2-bounded*. Recordem que en el nostre processador la cau L1 únicament és compartida per dos nuclis. Per tant, sols podrem estudar el cas en el que un *benchmark* s'executa concurrentment amb un programa *l2-bounded*. Com a valors del MPKI de L1 del *l2-bounded* hem escollit els valors 132, que és el MPKI de L1 màxim que hem aconseguit, els valors intermitjos 80 i 40, i finalment un MPKI de L1 de 10 que és el valor mínim que tenen els *benchmarks* de les SPEC 2006.

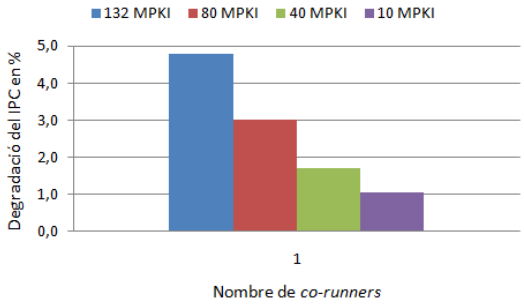
En la figura 4.8 mostrem el IPC per als *benchmarks* enters al executar-se en solitari i amb un *l2-bounded* 4.8(a) i la degradació que suposa aquesta reducció del IPC 4.8(b). Podem observar que la degradació per la contenció en l'accés a L1 és molt menor de la que obteniem per contenció en l'accés a memòria. Tot i això amb un MPKI de L2 de 132 s'obté una degradació del 4,77% i amb un MPKI de L1 de 80 és del 3%.

Pel que respecta als *benchmarks* amb coma flotant tenim la seva reducció del IPC i la degradació que suposa en la figura 4.9. Tal i com ocorria amb la degradació per la contenció en l'accés a memòria principal, la degradació en els *benchmarks* amb coma flotant és major que la que presenten els *benchmarks* enters.

Cal fixar-se en que la degradació per contenció en l'accés a L2 és semblant a la que s'obté per contenció en l'accés a memòria quan únicament tenim en compte 2 nuclis, la qual cosa ens ha sorprés. Si s'espera que en els pròxims processadors augmente molt el nombre de nuclis, els nivells de la jerarquia de cau previsiblement estaran compartits per més nuclis i la contenció i degradació podria molt major. En qualsevol cas en el nostre processador també existeix contenció i per això tractarem de reduïr-la amb la nostra proposta de planificació.

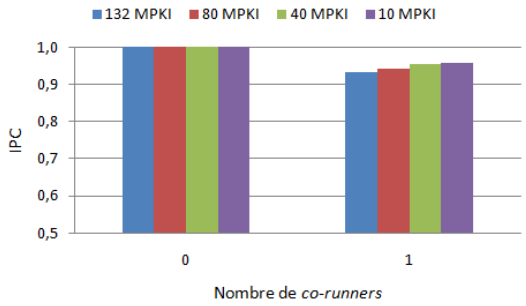


(a) IPC mitjà dels *benchmarks* enters

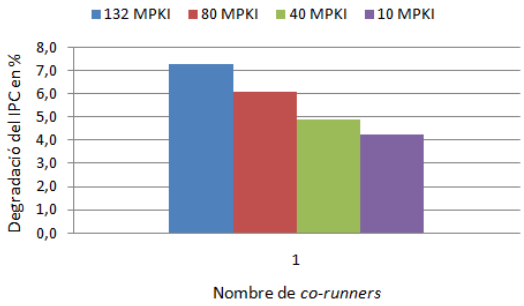


(b) Degradació mitjana de l'IPC dels *benchmarks* enters

Figura 4.8: Valors mitjans per a les execucions dels *benchmakrs* enters concurrentment amb un *l2-boundeds*



(a) IPC mitjà dels *benchmarks* amb coma flotant



(b) Degradació mitjana de l'IPC dels *benchmarks* amb coma flotant

Figura 4.9: Valors mitjans per a les execucions dels *benchmakrs* amb coma flotant concurrentment amb un *l2-boundeds*

Capítol 5

Planificació per aprofitar l'ample de banda de memòria

El primer planificador que estudiarem i avaluarem serà la proposta realitzada per D. Xu i C. Wu en el treball *On Mitigating Memory Bandwidth Contention through Bandwidth-Aware Scheduling* [XW10], que a la seva vegada està basat en alguns treballs previs de C. D. Antonopouls [C. 03b] [C. 04].

Com ja hem comentat, els programes sofreixen degradació de les seves prestacions quan la utilització que fan del controlador de memòria és alta. Per tractar d'evitar açò, el planificador tracta de seleccionar els programes que s'executaran en cada *quantum* de manera que els requeriments combinats d'ample de banda entre L2 i memòria s'aproximen a l'ample de banda ideal calculat per a la càrrega. D'aquesta forma s'espera que programes que realitzen moltes fallades de L2 es combinen amb programes que en realitzen poques, obtenint al final una degradació inferior a la que s'obtindria si es planificara sense prendre compte aquest fet.

En els pròxims apartats anirem detallant el funcionament del planificador però els passos que segueix són els que veiem a l'algoritme 1.

5.1 Ample de banda mitjà ideal

La idea de definir un ample de banda mitjà ideal o IABW ve per la necessitat de determinar el valor de l'ample de banda al qual tractarem d'apropar-nos amb els programes que llancem en cada quantum donada una càrrega concreta. En un principi podríem pensar que per a obtenir les millors prestacions la millor opció seria tractar d'aproximar l'ample de banda requerit en cada *quantum* al màxim ample de banda disponible en la pràctica, que no el màxim teòric. Açò no és bona idea per dos motius. Per una banda, ens podríem trobar en la situació de que els programes requerirem més ample de banda del que hem previst, superant l'ample de banda màxim i provocant la saturació del *bus* i la conseqüent reducció de les prestacions. Per una altra banda, encara que ens trobem per baix del límit pràctic, la contenció pot existir igual.

Per això apareix la idea d'utilitzar aquest IABW, que ens permetrà planificar el programes de manera que l'ample de banda s'aproxime a aquest IABW. Es tracta d'un valor crític en la planificació, ja que si aquest valor és massa menut utilitzarem menys el programes amb un requeriment d'ample de banda alt, quedant aquestos per al final

Algorithm 1 Planificació atenent a la contenció en l'accés a memòria principal

Require: Que les càrregues incloguen el temps d'execució de cada benchmark, T_i, i el seu BTR mitjà, B_i.

1: Calcular el IABW. 5.1
2: Crear els processos, insertar-los en la cua i detenir-los. 5.2
3: **while** queden programes per finalitzar **do**
4: BW_{Remain} = IABW, CPU_{Remain} = P.
5: Extraiem el primer procés de la cua. Actualitzem el BW_{Remain} i decrementem CPU_{Remain}.
6: **while** CPU_{Remain} && la cua no està buida **do**
7: Dels programes de la cua extraiem aquell que maximitza la funció FITNESS(p). 5.3
8: Actualitzem BW_{Remain} i CPU_{Remain}.
9: **end while**
10: LLancem els processos extrets.
11: Esperem un quantum.
12: Detenim els processos.
13: Llegim la PMU de cada procés.
14: **if** No hi ha contenció 5.4 **then**
15: Actualitzem la previsió de l'ample de banda requerit per al pròxim quantum dels processos executats.
16: **end if**
17: Insertem els processos que no han finalitzat en la cua.
18: **end while**

de l'execució de la càrrega, on no tindrem més remei que planificar-los junts ja que no quedaran més programes i açò crearà molta contenció i degradació de les prestacions. Per contra, quan major siga l'ample de banda estarem creant més contenció en la planificació ja que com hem comprovat en la degradació de les prestacions, únicament combinar un programa amb 3 programes que tenen un MPKI de L2 de 2,7 provoca una degradació de les prestacions del 7%.

Per a poder calcular l'IABW, necessitem que juntament amb els programes que formen la càrrega s'incloga informació addicional sobre ells com és l'ample de banda mitjà requerit i el seu temps d'execució quan s'executa el programa en solitari. En aquest cas per a quantificar l'ample de banda requerit per cada benchmark s'utilitza el BTR (Bus Transaction Rate), mesurant les transaccions que cada programa realitza per microsegon entre la cau L2 i la memòria principal.

Assumim per tant, que tenim un càrrega formada per N programes que s'executarà en un sistema amb C nuclis, sent el nombre de programes major que el de nuclis. Per a cada programa necessitem també conèixer el seu temps d'execució T_i i i el seu BTR entre L2 i memòria principal B_i.

A partir d'aquesta informació es calculen els accessos totals a memòria que en un principi va a realitzar la càrrega com vegem en l'equació 5.1.

$$TotalMemoryRequest = \sum_{j=1}^{N} T_j * B_j \tag{5.1}$$

En realitat, al tenir la cau L2 compartida per 2 nuclis el nombre real d'accessos es complicat d'estimar ja que els programes presenten BTR diferents en diferents trams d'execució i per estimar el nombre de fallades caldria prendre en compte el company amb el que es va a executar en cada quantum. Donat que açò pot ser massa complicat per a estimar-ho ens quedem amb aquesta fórmula per a quantificar els accessos totals a memòria de la càrrega.

Necessitem també estimar el temps d'execució ideal de la càrrega. Aquest temps dependrà de com els programes són planificats per el sistema operatiu i la contenció que apareix entre ells però en el nostre cas assumirem una política *round-robin* sense contenció per a obtenir una fita en la que estimem el temps d'execució mínim per a la càrrega. La fórmula per a calcular-lo és l'equació 5.2. En ella, els programes que formen la càrrega estan ordenats en ordre creixent per el seu temps d'execució, de manera que T_1 és el programa més curt i T_N el més llarg.

$$IdealTournaroundTime = \frac{\sum_{i=1}^{N-C} T_{n_i}}{C} + T_{n_N} \tag{5.2}$$

A partir d'aquestos dos valors, l'ample de banda mitjà ideal per a la càrrega es defineix per l'equació 5.3.

$$IdealAverageBandwidth = \frac{TotalMemoryRequest}{IdealTournaroundTime} \tag{5.3}$$

Com ens comenten els autors en la seva publicació, l'IABW estimat no pot ser utilitzat directament com a l'objectiu per a l'estratègia de planificació. El motiu és que el requeriment d'ample de banda que hem utilitzat per a fer el càlcul és el que s'ha obtingut quan els programes s'executaven en solitari. Quan els programes s'executen

juntament l'ample de banda que obtindran serà menor, ja que els programes estan competint per l'ample de banda disponible, creant esperes i contenció. De fet, algunes vegades l'ample de banda que obtindran serà bastant menor del que requerien.

Per tant, per a mantenir un funcionament del planificador consistent amb el que s'ha comentat cal aplicar una reducció a l'ample de banda mitjà ideal. En l'article comentat, els autors han arribat a l'equació 5.4.

$$Sched.Target = -0,0118 * IABW^2 + 1,4571 * IABW - 6,2403 \quad (5.4)$$

Aleshores, l'objectiu de la planificació serà aproximar l'ampe de banda consumit en cada quantum a aquest valor de Sched.Target. No obstant això, en els nostres experiments hem comprovat que quan les càrregues tenen un BTR baix és millor deixar el IdealAverageBandwidth sense aplicar-li la reducció del Sched.Target. Si el BTR de la càrrega és baix i el reduïm encara més ens podem trobar en la situació en que els programes amb un BTR quasi nul maximitzen la funció *fitness* respecte als programes que presenten el BTR relativament alt i açò el que provoca és que els programes amb més BTR queden junts per al final de l'execució on no es poden planificar perquè queden pocs processos, creant un poc de contenció i degradant les prestacions més que quan s'aproxima l'ample de banda per a cada quantum a IABW.

5.2 Ptrace attach i Ptrace detach

Per tal de facilitar la implementació del planificador i poder modificar-lo i avaluar-lo de manera senzilla sense necessitat de realitzar canvi en el nucli del sistema operatiu, hem optat per implementar-lo en un programa en C. Aquest programa serà l'encarregat de crear els processos necessaris en funció de la càrrega que vulguem utilitzar i una vegada creats anirà detenint-los i reiniciant-los segons les seves regles de planificació. La interferència del sistema operatiu en la nostra planificació deuria ser mínima, ja que en cap moment deixem executar-se més processos que nuclis, de manera que en cada quantums tots els processos que el nostre planificador inicia deurien executar-se durant tot el quantum sempre que no hi haja més processos actius en el sistema.

Per detenir i reiniciar els processos com hem comentat farem ús de la funció *ptrace*, amb l'argument PTRACE_ATTACH per a detenir els processos i PTRACE_DETACH per a reiniciar-los. El que fa PTRACE_ATACH és bàsicament enviar una senyal SIGSTOP al procés, de manera que evita que el planificador del sistema operatiu el puga posar en execució. Per contra PTRACE_DETACH, reinicia el procés que es trobava parat, entre d'altres, amb l'enviament d'una senyal SIGCONT.

Amb els processo parats el planificador s'encarregarà de realitzar la seva tasca, actualitzant la informació dels programes quan calga, mantenint la cua d'execució actualitzada i seleccionant els processos que s'executaran en el següent quantum. Una vegada estan seleccionats, es desbloquegen amb la crida PTRACE_ATTACH i espera amb una funció poll el temps de quantum. La funció *poll* espera el temps de quantum sempre que els programes no tinguen alguna notificació com puga ser la seva finalització, que de produir-se faria retornar la funció immediatament i per tant entrar al planificador abans d'acabar el quantum. En qualsevol cas, quan poll retorna el planificador bloquejarà els processos amb PTRACE_ATTACH i de nou començarà a realitzar

la seva feia per al pròxim quantum.

Per assegurar-nos de que la planificació és compleix i que els programes reben correctament la senyal PTRACE_ATTACH el que fem és esperar a que el programa reba la senyal i canvie d'estat, comprovant després que el canvi d'estat no ha sigut per la seva finalització:

```
1  for (i=0; i<N; i++) {
2      ret = ptrace(PTRACE_ATTACH, proces[i].pid, NULL, 0);
3      if (ret == -1) {
4        printf("Error: cannot attach to %d: %s\n", proces[i].pid,
             strerror(errno));
5        return -1;
6      }
7
8      waitpid(proces[i].pid, &status, WUNTRACED);
9
10     if (WIFEXITED(status)) {
11       fatal_error("command process %d exited too early with
            status %d\n", proces[i].pid, WEXITSTATUS(status));
12     }
13   }
```

Açò ens evitarà que un procés finalitze sense donar-nos compte, ja que sinó provocaria errors al planificador ja que tractaria de treballar amb el pid d'un procés que ja no existeix. Per a les crides PTRACE_DETACH no és necessari realitzar aquesta comprovació ja que al haver fet la anterior tenim la certesa de que en un principi la funció no fallarà ja que el procés deu existir i deu estar parat si no ha hagut cal problema greu.

5.3 Selecció dels processos a executar en un quantum

Per a seleccionar els processos que seran executats en el següent quantums s'utilitza la fórmula *fitness*. Aquesta fórmula quantifica la diferència entre el requeriment de l'ample de banda del procés que espera per a ser planificat i l'ample de banda disponible per a cada nucli. El procés amb una diferència menor tindrà un *fitness* millor, i serà un dels programes que serà llançat en el següent quantum. Després de seleccionar un programa els seus requeriments d'ample de banda es resten al BW_remain i el nombre de processadors disponibles es redueix en una unitat.

La funció *fitness* és la següent:

$$FITNESS^P = \frac{1}{\frac{BW_{remain}}{CPU_{remain}} - BW^P_{required}} \tag{5.5}$$

No obstant, cal tenir en compte que per a cada quantum el primer procés de la cua d'execució es selecciona sempre. Açò evita que es puga produir la inanició d'algun procés, és a dir, que algun procés pels seus requeriments d'ample de banda és puga

trobar en una situació en la que no passe a executar-se mai. Seleccionant el primer procés de la cua d'execució per a que es llance sempre s'evita aquesta situació ja que després d'executar-se els programes s'afegeixen per la cua. Una vegada seleccionat aquest procés la resta de processos fins completar el nombre de nuclis del sistema són seleccionat amb la funció *fitness*.

5.4 Límit per a la contenció

En el nostre planificador considerem que l'ample de banda consumit en el quantum anterior és el mateix que requerirà en el següent quantum. Açò no obstant cal agafar-ho amb cura ja que si es produeix la saturació del bus, o fins i tot abans de que es produïsca, l'ample de banda que els processos obtenen pot ser menor que el que realment requerien degut a la contenció.

El que fem per a evitar aquest problema és definir un límit T_B, menor que l'ample de banda màxim mesurat. Quan la suma de l'ample de banda consumit per tots els processos en un quantum siga menor que T_B considerarem que no s'ha produït massa contenció i que la informació que agafem d'aquest quantum pot ser utilitzada per a preveure el comportament en el següent. Si per contra l'ample de banda total és superior a T_B, considerarem que s'ha produït molta contenció i aleshores no actualitzarem l'ample de banda per al pròxim quantum amb la informació del quantum anterior. El que farem en aquest últim cas és fixar els requeriment de l'ample de banda als valors mitjans que havíem mesurat quan el programa s'executava en solitari.

La dificultat del problema queda molt clara amb el següent exemple real. Tenim el benchmark *lbm* de les SPEC 2006 que té un BTR entre L2 i memòria principal de 27,54 transaccions/usec quan s'executa en solitari. Si executem dos instàncies d'aquest benchmark el BTR que aconseguim per a cadascuna està al voltant de les 10 transaccions/usec, amb la qual cosa el total pot estar al voltant de les 20-22 transaccions/usec. Si fixarem el T_B a 20 tindríem el problema de entrar en contenció quasi sempre que s'execute un programa de *lbm*, ja que ell mateix supera el llindar. Si fixem T_B a 25 tenim el problema de no detectar aquesta contenció amb 2 processos *lbm* concurrents, amb la qual cosa actualitzarem mal els requeriments per al pròxim quantum i probablement la planificació serà incorrecta per a pròxims quantums, creant més contenció, esperes i degradació de prestacions.

Una solució alternativa a fixar l'ample de banda requerit a la mitja podria ser simplement no actualitzar l'ample de banda requerit i deixar el que teníem anterior però en les proves que hem realitzat hem obtingut millors prestacions fixant-lo a la mitja.

El motiu d'aquest comportament podria ser per la complicació d'eixir de la contenció quan s'entra. Si entren en contenció el valor de l'ample de banda que mesurem serà menor que el que el programa realment requeria. Si no ho detectem prendrem aquest valor com a l'ample de banda que el procés requerirà en el següent quantum. Al ser una previsió inferior al requeriment real del programa i al planificar-lo conjuntament amb altres programes tractant d'apropar-nos al IABW, probablement acabarà provocant més contenció ja que l'ample de banda real requerit pels programes serà superior al IABW. Entrar en aquesta situació és fàcil si els programes tenen requeriments de memòria grans i l'IABW de la càrrega és alt, però és complicat d'eixir ja que la conten-

ció provoca que baixe l'ample de banda utilitzat i dificulta que aquest supere el nostre llindar. Per tant, i com hem vist en l'exemple, utilitzar aquestos punts per eixir d'una suposada contenció pot ajudar al funcionament del planificador.

El llindar T_B es tracta per tant d'una solució simple a un problema molt més complex. En aquest projecte realitzarem proves empíriques variant T_B i ens quedarem amb el que oferisca millor resultats però en treballs futurs seria un punt important per a millorar ja que el seu impacte en les prestacions pot ser molt gran.

5.5 Longitud dels quantums

La longitud del *quantum* per al planificador és el temps que el planificador deixarà executar-se als processos lliurement entre planificació i planificació. Com veiem, el planificador quan entra en execució detén els processos actius, i després d'actualitzar la informació dels processos a partir de la PMU, selecciona els que seran llançats en el pròxim quantum, i els rellança, abans d'esperar el temps de *quantum* per a tornar a entrar a planificar. Cal diferenciar-lo per tant, del *quantum* del sistema operatiu, ja que el nostre planificador funciona sobre aquest, com un programa normal. De totes formes, donat que el planificador sols deixa en execució en cada *quantum* un nombre de processos igual o inferior al de nuclis, la influència que pot tenir en l'execució és molt limitada.

La longitud del *quantum* s'ha d'ajustar tenint en compte la manera que tenim d'estimar els requeriments d'ample de banda per al pròxim *quantum*. Donat que utilitzarem la informació del *quantum* anterior com a previsió per al següent, necessitem una longitude de *quantum* que permeta que aquesta previsió siga correcta. Un *quantum* massa curt por provocar errors en la previsió per considerar xicotetes ràfeques d'accessos a memòria que provocarien una previsió errònia. D'altra banda *quantums* massa llargs podrien no detectar ràfegues majors entrant també en una previsió incorrect. Per trobar el *quantums* òptim provarem amb diferents valors en els experiments i determinar per tant, el que funciona millor en la nostra planificació.

Un altre aspecte que cal tenir en compte al definir la longitud dels *quantums* és que quan els processos arriben a un nucli diferent al que es trobaven en execució, tindran que reconstruir el seu estat en la cau [C. 03b]. Açò provocarà fallades no contemplades en la previsió de l'ample de banda. Aquest fet limita les prestacions dels *quantums* curts en relació als llargs, ja que els primers tenen que realitzar moltes més reconstruccions de l'estat, provocant fallades extres. A més, açò és una qüestió que els planificadors actuals tenen en compte a l'hora d'assignar un nucli a un procés. El planificador que estudiem en aquest capítol encara podria beneficiar-se d'aquest fet si algun procés s'executa més d'un *quantum* seguit. Per contra, donat que la nostra proposta de planificació que veurem en el següent capítol assigna els nuclis als processos en funció dels requeriments d'ample de banda per al següent *quantum*, pedrem l'avantatge que ofereix aquesta característica respecte a la planificació del sistema operatiu.

Capítol 6

Extensió de la planificació per a contemplar l'ample de banda en tots els nivells de la jerarquia de memòria

La motivació per a estendre el planificador per a contemplar la contenció en la jerarquia completa de la cau i no únicament la contenció en l'accés a la memòria principal el trobem en la degradació que hem observat i mesurat en l'apartat abans.

La idea per a planificar els programes és seleccionar-los com es feia en el planificador que hem estudiat abans. La diferència es troba en que en aquest planificador una vegada seleccionats els programes s'enviaran a un nucli concret atenent a la seva previsió de l'ample de banda requerit entre L1 i L2.

El motiu per el qual sols tenim en compte l'ample de banda requerit entre L1 i L2 una vegada hem seleccionat els programes que s'executaran en el pròxim quantum és que la contenció i la degradació de prestacions en aquest punt és menor. Aquestes caus es troben dins del processador i açò provoca que tant la capacitat del bus com la seva velocitat siguen majors i per tant la degradació de prestacions menor. De manera simplista, podríem aproximar que si una transacció entre L2 i memòria principal deu esperar que la anterior finalitze podem estar parlant de una penalització de 100 cicles, mentre que si el programa deu esperar a que finalitze una transacció entre L1 i L2 la penalització pot ser de 10 cicles.

Cal tenir en compte també que en l'ordinador que anem a utilitzar per a realitzar les proves la planificació que podem realitzar entre L1 i L2 està bastant limitada ja que únicament disposem de dos memòries cau L2 cadascuna compartida per 2 nuclis. Per això, la degradació serà menor, igual que la millora de les prestacions per realitzar aquesta planificació de manera correcta, que si el nombre de nuclis que compartiren una cau fora major. No obstant això, el nostre esquema de planificació pot ser aplicat a qualsevol processador multinucli i en futurs treballs ampliarem l'avaluació a sistemes més amplis utilitzant el simulador Multi2Sim [USPL07].

L'algorisme que seguim en aquesta planificació és molt semblant al que teniem abans per a optimitzar l'ús de l'ample de banda per accedir a memòria 1. L'única diferència es troba en que després d'extraure tots els processos i abans de reiniciar-los per a que

s'executen un quantum, es selecciona el nucli en el que cada procés s'executarà en el següent quantum (algorisme 2, línea 10). L'algorisme complet on podem veure el que hem comentat és el algorisme 2.

Algorithm 2 Planificació per a optimitzar l'ús de l'ample de banda en tota la jerarquia de memòria

Require: Que les càrregues incloguen el temps d'execució de cada benchmark, T_i, i el seu BTR mitjà, B_i.

1: Calcular el IABW. 5.1
2: Crear els processos, insertar-los en la cua i detenir-los. 5.2
3: **while** queden programes per finalitzar **do**
4: BW_{Remain} = IABW, CPU_{Remain} = P.
5: Extraiem el primer procés de la cua. Actualitzem el BW_{Remain} i decrementem CPU_{Remain}.
6: **while** CPU_{Remain} && la cua no està buida **do**
7: Dels programes de la cua extraiem aquell que maximitza la funció FITNESS(p). 5.3
8: Actualitzem BW_{Remain} i CPU_{Remain}.
9: **end while**
10: A cada procés extret se li assigna el millor nucli en el que es pot executar atenent a les previsions de l'ample de banda requerit entre L1 i L2 de tots els processos 6.1.
11: LLancem els processos extrets.
12: Esperem un quantum.
13: Detenim els processos.
14: Llegim la PMU de cada procés.
15: **if** No hi ha contenció 5.4 **then**
16: Actualitzem la previsió de l'ample de banda requerit per al pròxim quantum dels processos executats a partir de la informació d'aquest quantum.
17: **else**
18: Actualitzem la previsió de l'ample de banda requerit per al pròxim quantum dels processos executats amb la seva mitjana.
19: **end if**
20: Insertem els processos que no han finalitzat en la cua.
21: **end while**

6.1 Selecció del nucli per a cada procés

Com hem vist, per poder fer la planificació atenent també a l'ample de banda requerit per cada programa per accedir a L2, cal indicar al sistema operatiu el nucli en el que volem que cada procés s'execute. Afortunadament Linux disposa de les estructures i funcions necessàries que implementen aquesta funcionalitat.

L'element clau per a obtenir aquest comportament és la màscara d'afinitat de les CPU dels processos que determinen en quin conjunt de nuclis pot escollir el planificador de Linux per a possar en execució un procés. Cal recordar que a nivell de sistema

operatiu Linux identifica cada nucli del processador com si fora un processador monolític. En el nostre cas fixarem aquesta màscara amb un únic nucli per a cada procés de manera que podrem controlar completament en quin nucli concret s'està executant cada procés en cada quantum.

La funció que hem d'utilitzar és *sched_setaffinity* que compta amb la següent interfície:

```
1 int sched_setaffinity ( pid_t pid, unsigned int cpusetsize,
     cpu_set_t *mask)
```

El primer argument de la funció correspon amb el pid al que volem assignar la màscara, el segon el la grandària de la màscara i el tercer correspon a un punter a la màscara. Com podem veure la màscara és del tipus cpu_set_t. Per a manejar aquesta estructura amb comoditat existeixen quatre macros. En el nostre cas utilitzarem únicament CPU_ZERO() per a eliminar els nuclis associats a una màscara i CPU_SET() per afegir el nucli desitjat a la màscara.

En el nostre sistema la planificació en tota la jerarquia de la cau està bastant limitada. Solament disposem de dos nivells de cau i únicament el segon nivell de la cau està compartit. A més, únicament està compatit per dos nuclis. Per això el que farem serà tractar d'equilibrar l'ample de banda per a accedir a cadascuna de les dos caus L2. Per fer-ho juntarem en una mateixa cau de L2 el procés amb un requeriment d'ample de banda entre L1 i L2 major amb el que presente aquest requeriment mínim, deixant els altres dos processos executar-se compartint l'altra cau L2.

Capítol 7

Metodologia per a l'avaluació

En aquest capítol explicarem com es va a realitzar l'avaluació de les prestacions dels planificadors. Comentarem la problemàtica que suposa que els benchmarks tinguen temps d'execució diferents i explicarem la solució que hem utilitzat per evitar aquest problema. També detallarem les càrregues que hem creat i els dos paràmetres principals del planificador que són el llindar de contenció que hem explicat abans i la longitud del quantum. Per últim, comentarem com avaluarem el temps d'execució del planificador de Linux.

7.1 Definició dels benchmarks amb el mateix temps d'execució

Tant per a la caracterització com per a l'avaluació de les prestacions s'han utilitzar benchmarks de les SPEC 2006 amb les càrregues de treball *train*. Les execucions amb aquestes càrregues ofereixen temps de resposta molt diferents amb benchmarks com *wrf*, *sphinx3* o *specrand*, la execució dels quals no arriba al segon i altres com *tonto*, que superen els 400 segons.

Aquesta gran diferència en el temps d'execució complicaria l'avaluació del planificador i faria que una simple estratègia de *long job first* donara bons resultats en la majoria de càrregues ja que podria balancejar millor la càrrega entre tots els nuclis.

Per tal d'evitar aquest problema el que es fa és executar cada benchmark en solitari durant 2 minuts, anotant el nombre de vegades que el benchmarks s'executa complet i el nombre d'instruccions que s'executen de la última instància no finalitzada. D'aquesta forma aconseguim igualar el temps d'execució de tots els benchmarks.

Aquest esquema presenta dos avantatges. El primer és que la composició de les càrregues és fixa i per tant els experiments es poden realitzar amb diferents polítiques de planificació sent els resultats comparables. El segon avantatge és que permet centrar l'estudi en la planificació per evitar la contenció en l'accés a memòria deixant de costat altres qüestions que poden aparèixer si els benchmarks tenen temps d'execució diferents.

7.2 Composició de les càrregues

Les càrregues que realitzem per avaluar el planificador estaran formades, per tant, per els 2 minuts d'execució de cadascun dels benchmarks que la formen. El nombre de benchmarks que hem decidit que formen una càrrega és de vuit per a deixar el grau de multiprogramació en 2. Aquestos benchmarks són seleccionats aleatòriament per tal de formar les diferents càrregues que utilitzarem per avaluar les dos propostes de planificació, tot i que es tracta en la mesura del possible que els càrregues presenten uns requeriments d'ample de banda mitjos o alts, ja que en cas contrari la planificació atenent al consum d'ample de banda perd el seu motiu.

La composició de les càrregues és emmagatzemada per tal de poder aplicar-la a tots els planificadors i comprar el seu rendiment, i la podem veure en la taula 7.1, ordenades per el seu IABW de forma decreixent.

Càrrega	Composició	IABW
WL#4	*lbm, lbm, mcf, GemsTDTD, astar, cactusADM, xalancmbk, tonto*	39,15
WL#3	*lbm, lbm, mcf, GemsFDTD, astar, xalancmbk, tonto, hmmer*	36,98
WL#7	*lbm, lbm, mcf, astar, bwaves, sjeng, dealII, xalancbmk*	35,58
WL#1	*lbm, lbm, mcf, GemsFDTD, h264ref, xalancbmk, tonto, hmmer*	34,23
WL#5	*lbm, mcf, GemsFDTD, astar, cactusADM, bwaves, xalancbmk, tonto*	26,20
WL#6	*lbm, mcf, GemsFDTD, astar, sjeng, dealII, namd, h264ref*	24,14
WL#2	*lbm, mcf, GemsFDTD, astar, h264ref, xalancbmk, tonto, hmmer*	23,29

Taula 7.1: Càrregues per a l'avaluació

7.3 Altres paràmetres de la planificació

El planificador presenta dos paràmetres que poden tenir una influència alta en el comportament del planificador: el llindar de contenció i la longitud del quantum. El llindar de contenció està explicat en l'apartat 5.4. És un paràmetre que pot arribar a ser crític per al funcionament del planificador. Segurament es podria treballar per trobar una forma millor de mesurar si s'ha produït o no contenció, però en el nostre cas ens limitarem a obtenir el valor òptim per a aquest paràmetre empíricament, avaluant el resultat de la planificació per a llindars de contenció entre 21 i 41 transaccions per microsegon en l'accés a memòria principal.

L'altre paràmetre important és la longitud dels quantums, entenent per longitud de quantum el temps que deixa el planificador per a que s'executen els processos abans d'interrompre'ls de nou, per a realitzar la planificació, com hem explicat en l'apartat 5.5. En aquests experiments avaluarem longituds de *quantum* entre 200 ms i 600 ms. Un *quantum* menor complicaria les prestacions, ja que el propi sistema operatiu treballa a 100 ms. D'altra banda longituds de *quantum* majors comencen a ser massa llargues,

ja que en aquest temps es poden observar fragments amb un nombre de transaccions major i fragments amb menys transaccions, sent interessant tractar-los per separat.

7.4 Planificació de Linux

A l'utilitzar per a cada benchmark els 2 minuts de la seva execució que hem mesurat en solitari, tenim el problema de no poder mesurar el temps de resposta del planificador de Linux sense realitzar cap interferència en el seu funcionament. Utilitzar els benchmarks d'aquesta forma ens obliga a estar detenint i reiniciant els processos cada poc de temps, per tal de comprovar si algun ha acabat, rellançar els que tinguen execucions pendents o finalitzar aquells que superen el nombre d'instruccions de l'última execució.

Per tant, el que fem per avaluar el planificador de Linux és seguir un esquema semblant al que utilitzàvem per als nostres planificadors amb la funció *ptrace* per a detenir i reiniciar els processos. Això si, en cada quantum es llançaran tots els processos, i per tant el planificador del sistema operatiu serà l'encarregat de decidir quins s'executen en cada moment. Una vegada finalitzat el quantum, el programa únicament rellançarà els que hagen acabat i tinguen execucions pendents i finalitzarà aquells que estant en la seva última execució superen el nombre d'instruccions que els toquen. D'aquesta forma no tindrà cap interferència en la planificació.

A pesar de que aquest fet pot semblar un desavantatge, ens permet realitzar una comparació més justa. Utilitzant la longitud dels quantums igual a la que utilitzem en els nostres planificadors aconseguim que els processos es detinguen i inicien de la mateixa forma, i per tant, les diferències en els temps d'execució deurien ser degudes exclusivament a la qualitat de la planificació i no ha cap altre tipus d'interferència.

Capítol 8

Avaluació de les propostes

Seguint la metodologia descrita en el capítol anterior, s'han realitzat tots els experiments per tal d'avaluar les dos propostes de planificació front a la planificació que realitza el propi planificador de Linux. Mostrarem dos tipus de gràfiques per exposar els resultats. En primer lloc, mostrarem l'acceleració mitjana que obtenim en les càrregues, fixant el *quantum* i variant el llindar de contenció. En segon lloc mostrarem per a cada *quantum* una gràfica en la que podrem veure l'acceleració màxima que s'aconsegueix per a cada càrrega independentment del llindar de contenció amb el qual s'aconseguisca.

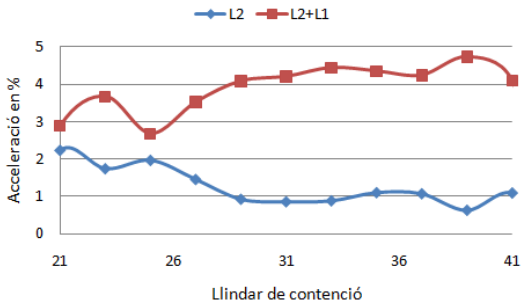
Per facilitar la comprensió ens referirem com a planificació L2, a la planificació per evitar la contenció en l'accés a memòria principal, tal com hem explicat en el capítol 5. D'altra banda, ens referirem a la nostra proposta de planificació explicada en el capítol 6, que tracta d'evitar la contenció en tota la jerarquia de memòria, incloent les caus, com a planificació L2+L1.

8.1 Acceleració mitjana per *quantum*

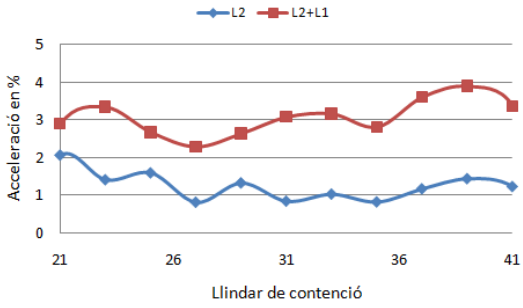
En la figura 8.1 mostrem l'acceleració mitjana que hem obtingut al avaluar les càrregues, variant el llindar de contenció. Podem veure els resultats en cada subfigura en funció del *quantum* amb el que treballen els planificadors. A continuació comentarem els resultats obtinguts per a cada *quantum*, però en general la nostra proposta de planificació, amb L2+L1, millora al voltant d'un 4% a la planificació que realitza Linux i al voltant d'un 2% la planificació únicament de L2.

Per entendre també els resultats, és important saber que quan major siga el llindar de contenció, sense entrar en excessiva contenció, en teoria es va a realitzar una planificació millor. Això ocorre perquè quan més baix siga el llindar de contenció és superarà un major nombre de vegades, i quan es supera es realitza una planificació estàtica, rebutjant les mesures de prestacions realitzades i utilitzant les mitjanes calculades en la caracterització. Per contra, si el llindar és massa alt, provoca que es cree contenció sense detectar-ho. Açò fa que les previsions d'ample de banda realitzades siguen menors de les que realment tenia el procés, i per tant la planificació no siga del tot correcta, com hem explicat en 5.4.

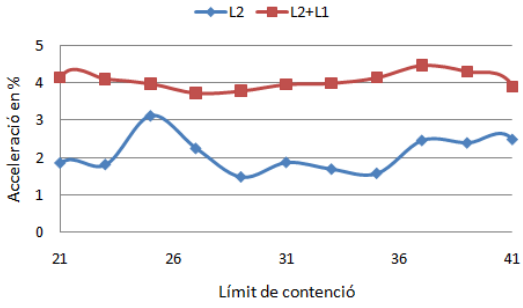
En la figura 8.1(a) tenim els resultats per a les planificacions amb longitud de *quantum* de 200 ms. Si ens fixem amb la planificació de L2, aconsegueix el seu màxim



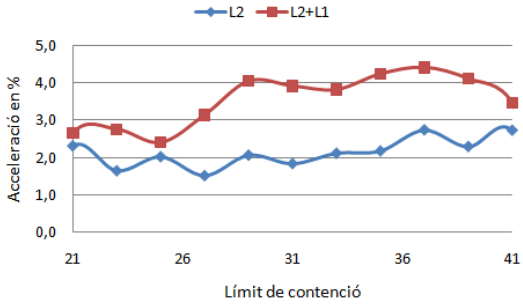
(a) *Quantum* = 200 ms



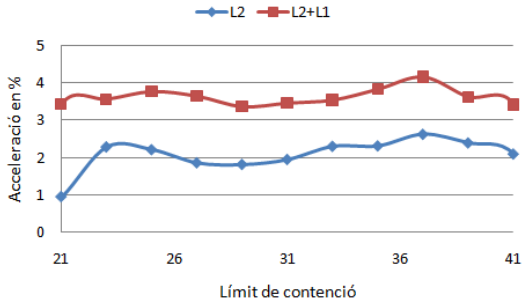
(b) *Quantum* = 300 ms



(c) *Quantum* = 400 ms



(d) *Quantum* = 500 ms



(e) *Quantum* = 600 ms

Figura 8.1: Acceleració mitjana comparant les tres planificacions variant el llindar de contenció.

amb un llindar de contenció de 21 i una acceleració mitjana del 2,24%. Segueix una tendència decreixent fins arribar a un llindar de contenció de 27, on l'acceleració mitjana s'estabilitza al voltant de l'1%. Per contra, l'acceleració mitjana de la planificació de L2+L1 segueix una tendència creixent, aconseguint el seu màxim amb un llindar de contenció de 39. El motiu pel qual podem trobar aquestes tendències oposades pot ser que al planificar també sobre L1, s'evita contenció entre L1 i L2, i això permet augmentar el llindar de contenció entre L2 i memòria, sense perill de sobrepassar-lo contínuament.

Els resultats per a *quantums* de 300 ms els tenim en la figura 8.1(b). Observem també una tendència decreixent en la planificació de L2, amb un màxim de 2,06% en

un llindar de contenció de 21 i després baixa per estabilitzar-se al voltat de l'1%. La planificació de L2+L1 arriba al seu màxim amb un llindar de contenció de 39 i una acceleració del 3,9%.

En la figura 8.1(c) tenim els resultats per a la planificació amb *quantum* de 400 ms. En aquest cas les tendències estan molt més estabilitzades i es troben sobre el 2% per a la planificació de L2 i un 4% per a L2+L1. Els màxims es troben en 3,11% per a la planificació de L2 amb un llindar de contenció de 25 i 4,47% per a la planificació de L2+L1 i un llindar de contenció de 37.

Per a *quantums* de 500 ms tenim els resultats en la figura 8.1(d). Tenim una tendència estabilitzada per a la planificació de L2 o inclús creixent ja que aconsegueix la millor acceleració amb un llindar de contenció de 37 i 2,75%. Amb la planificació de L2+L1 tenim uns millors resultats a partir d'un llindar de contenció de 29, amb el l'acceleració màxima de 4,41% amb un llindar de contenció de 37.

Per últim, en la figura 8.1(e) tenim l'acceleració mitjana per a *quantums* de 600 ms. En aquest cas tenim també tendències estabilitzades, al voltant del 2% per a la planificació de L2 i cap al 4% per a L2+L1. Els màxims es troben en un llindar de contenció de 37, amb un 4,17% per a L2+L1 i 2,64% per a la planificació únicament de L2.

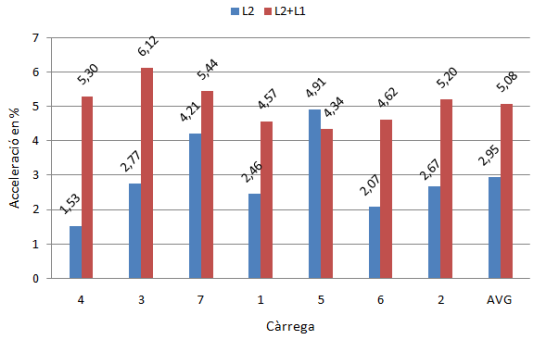
Tot i que per analitzar els resultats hem observat la tendència de la mitjana i sobre quins valors es troba, cal fixar-se en que el valor que realment importa és el màxim d'aquesta mitjana i el llindar de contenció òptim, amb el que s'aconsegueix aquest màxim. La tendència d'aquesta mitjana ens pot servir per trobar el llindar òptim i analitzar el comportament amb diversos llindars, però una vegada trobat el millor, aquest serà el que s'utilitzarà en les càrregues reals, esperant obtenir una acceleració mitjana com la que s'ha obtingut en els experiments de l'avaluació.

Per això és important destacar que, la millor planificació amb la política de L2+L1 s'obté amb *quantums* de 200 ms i llindar de contenció de 39, amb una acceleració mitjana de 4,73%. Per la seva part, per a la política de planíficació de L2, s'ha obtingut la millor mitjana en un llindar de contenció de 25 i *quantums* de 400 ms, aconseguint una acceleració mitjana del 3,11%.

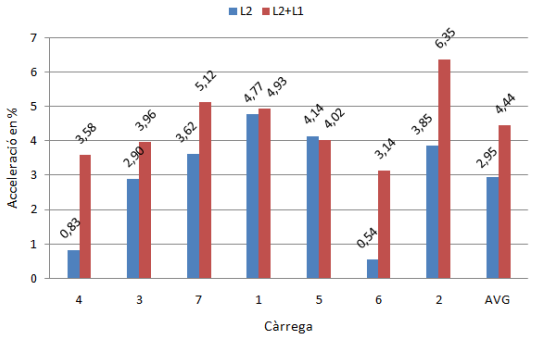
8.2 Acceleració màxima per càrrega

En la figura 8.2 mostrem l'acceleració màxima obtinguda per a cada càrrega en funció del *quantum* amb el que ha treballat el planificador i la mitjana que han obtingut totes les càrregues. Aquestes mitjanes poden ser una bona aproximació dels resultats dels planificadors sempre que obtinguérem una forma òptima per a determinar quan s'ha produït contenció en l'accés a algun element de la jerarquia de memòria.

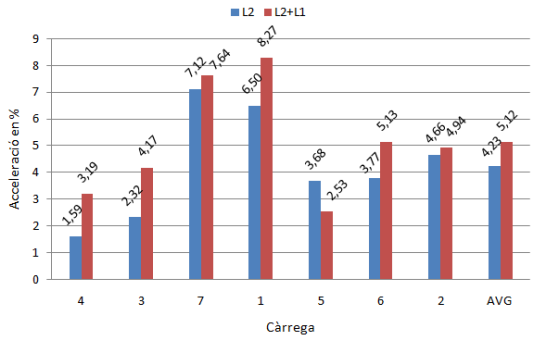
Comencem analitzant els resultats per al *quantum* de 200 ms, figura 8.2(a). Per a la planificació contemplant únicament la contenció en l'accés a memòria principal obtenim una acceleració màxima mitjana de 2,95%, mentre que amb la nostra proposta de planificació per evitar la contenció en tota la jerarquia de memòria, arribem al 5,08%. Són uns bons resultats, sobretot per a la nostra política de planificació ja que en ella l'acceleració màxima més baixa la tenim amb la càrrega 4 amb 4,34%. Cal destacar també que en aquesta càrrega funciona millor la planificació únicament de les fallades



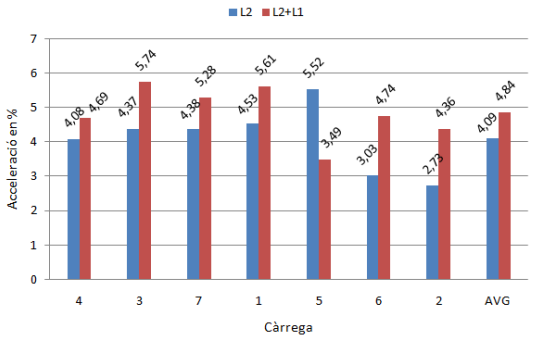
(a) *Quantum* = 200 ms



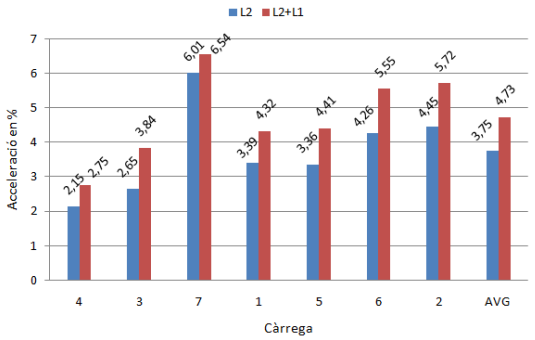
(b) *Quantum* = 300 ms



(c) *Quantum* = 400 ms



(d) *Quantum* = 500 ms



(e) *Quantum* = 600 ms

Figura 8.2: Acceleració màxima obtinguda per a les dos propostes de planificació en funció del *quantum*.

de L2 que la nostra (planificant les fallades de L1 i L2), ja que amb L2 únicament arriba a 4,91%.

En la figura 8.2(b) tenim els resultats per a *quantums* de 300 ms. Obtenim una mitjana per a la planificació de L2 de 2,95, amb un màxim de 4,77 i dos càrregues, la 4 i la 6, que no arriben a una acceleració del 1%. Per la seva part la planificació de L2+L1 arriba a una acceleració màxima mitjana del 4,44%, en aquest cas amb un màxim del 6,35% en la càrrega 2, que és la que té un IABW menor.

Amb *quantums* de 400 ms, en la figura 8.2(c), obtenim els millors resultats per a la

proposta de planificació de L2+L1, amb una mitjana de 5,12%, mentre que planificant únicament atenent a L2 dóna una mitjana de 4,23%. Destaquem que amb aquest *quantum* obtenim l'acceleració màxima major, amb una acceleració del 8,27% per a la càrrega 0.

Els resultats per a longituds de *quantum* de 500 ms els tenim en la figura 8.2(d). En aquest cas tenim la menor diferència entre les dos polítiques de planificació amb 4,09% per a la planificació de L2 i 4,84% per a la planificació de L2+L1. No obstant això, els dos resultats són bons. Destaquem també d'aquestos resultats que per a la càrrega 4, com ja hem observat amb altres longituds de *quantum*, la planificació de L2 obté una acceleració de 5,52% front al 3,49% de la planificació de L2+L1.

Per últim, en la figura 8.2(d) tenim els resultats per a *quantums* de 600 ms. Obtenim una acceleració màxima mitjana de 3,75% per a la planificació únicament de L2 i 4,73% si planifiquem sobre L2+L1.

Podem considerar positius els resultats obtinguts. Podem comparar els resultats de la planificació per evitar la contenció en l'accés a memòria principal amb els de [XW10]. En aquest treball, amb la mateixa planificació però un processador diferent, s'obté una acceleració mitjana del 3,4% per a les càrregues amb *benchmarks* de les SPEC 2006, mentre que en el nostre processador podríem al 4,23%. Per a la nostra proposta de planificació obtenim una acceleració màxima mitjana del 5,12%. No obstan això, els resultats no són realment comparables, ja que els processador són diferents, i simplement la comparació ens serveix per a observar que els resultats estan pròxims als obtinguts en altres treballs semblants.

Capítol 9

Conclusions i treball futur

9.1 Conclusió

En aquest treball hem estudiat l'impacte que té, en les prestacions d'un procés, el fet de tenir que executar-se concurrentment amb altres processos. En els processadors multinucli, els processos que s'executen en cada nucli han de compartir diferents estructures que en els processadors monolítics no es compartien, com el bus per accedir a memòria principal o el propi controlador de memòria. Açò implica que aquestos punts tinguen una utilització molt major, que pot arribar a sobrepassar la seva capacitat per moments, provocant contenció i afectant a les prestacions del sistema.

Açò és el que ocorre quan diversos processos tracten d'accedir molt a la memòria principal. Com veiem en l'apartat 4.2, la contenció per l'accés a memòria principal, quan un procés s'executa concurrentment amb altres que tenen un alts requeriments d'ample de banda, provoca una pèrdua de prestacions mitjana de fins a un 21% per a les càrregues SPEC 2006 que utilitzen coma flotant. Però dins la jerarquia de memòria, aquest no és l'únic punt on es produex contenció. El nostre processador disposa de dos memòries cau L2, cadascuna compartida per dos nuclis, i els accessos d'aquestos dos nuclis a la cau, també provoquen una contenció que pot reduir en un 4,77% les prestacions per a les mateixes càrregues, com mostrem en l'apartat 4.3.

I tot açò ocorre quan el controlador de memòria es troba compartit pels 4 nuclis del nostre processador i la memòria cau de L2 ho fa per 2 nuclis. Les previsions tecnològiques indiquen que en 2017 hi haurà diversos centenars de nuclis en el mateix xip, amb la qual cosa tant els controladors de memòria com les cau dels nivells superiors com L2 o L3, estaran compartides per més nuclis que en el nostre processador i la contenció i la pèrdua de prestacions serà major.

És evident, per tant, que necessitem polítiques de planificació que prenguen en compte els requeriments d'ample de banda que tenen els processos per accedir a tots els elements de la jerarquia de memòria i puguen planificar els processos per tal d'evitar aquesta contenció i pèrdua de prestacions. És el que fem al capítol 6, i els resultats, que podem veure en el capítol 8, ens donen una acceleració màxima mitjana del 5,12% respecte a la planificació del sistema operatiu. Si el nombre de nuclis creix com hem comentat, aquesta acceleració respecte a un planificador que no pren compte d'aquest requeriment d'ample de banda, creixerà en major mesura, la qual cosa farà inviable la utilització d'una política que no tracte d'optimitzar la utilització de l'ample de memòria

en la jerarquia de memòria.

9.2 Treball futur

Queden moltes idees en les que treballarem en els pròxims anys. En primer lloc, i respecte a la planificació per evitar la contenció en l'accés a la jerarquia de memòria, queda generalitzar l'algorisme per a que puga ser utilitzat en qualsevol processador, independentment de la jerarquia de cau de que dispose, distribuint de la millor forma possible els accessos. Per aconseguir-ho serà important treballar amb el simulador Multi2Sim [USPL07], que ens permetrà modelar els processadors i les jerarquies de memòria, per tal d'avaluar les prestacions que s'obtenen en cada cas.

Aprofundint més en temes de planificació, una altra idea sobre la que treballarem serà la gestió del voltatge i la freqüència amb els que treballa cada nucli. En aquest sentit es poden desenvolupar polítiques per tal de minimitzar el consum mantenint el rendiment o altres polítiques que tracten de minimitzar el consum mantenint un determinat rendiment. Aquest últim cas pot ser especialment útil per a dispositius mòbils com *smartphones* que funcionen amb bateria.

Relacionat amb aquestes dos idees, s'ha demanat en préstec a Intel un prototip que compta amb 48 nuclis. Aquest prototip disposa de 4 controladors de memòria, amb la qual cosa, cadascun es troba compartit per 12 nuclis. Açò pot provocar una contenció major en l'accés a memòria, i per tant és d'esperar que les polítiques de planificació que hem comentat tinguen una acceleració major respecte al planificador del sistema operatiu. D'altra banda, el prototip també compta amb diferents dominis de voltatge i de freqüència, permetent-nos treballar sobre aquests aspectes per contenir el consum mantenint les prestacions.

Per últim, però també relacionat amb la planificació i les freqüències tenim la planificació per a processadors amb nuclis heterogenis. Aquestos nuclis poden ser heterogenis pel seu disseny o per deficiències en la seva fabricació que poden afectar a algunes estructures d'un nucli, obligant-lo a no utilitzar-les o a treballar a una freqüència menor. Treballarem, per tant, en polítiques per a maximitzar el rendiment en aquests tipus de processadors.

Bibliografia

[C. 03a] C. D. Antonopoulos, D. S. Nikopoulos, and T. S. Papatheodoro. Report Scheduling algorithms with bus bandwidth considerations for SMPs. 2003.

[C. 03b] C. D. Antonopoulos, D. S. Nikopoulos, and T. S. Papatheodorou. Scheduling algorithms with bus bandwidth considerations for SMPs. *Proceedings of the 2003 International Conference on Parallel Processing (ICPP'03)*, page 547, October 2003.

[C. 04] C. D. Antonopoulos, D. S. Nikopoulos, and T. S. Papatheodorou. Realistic workload scheduling polcies for taming the memory bandwidth bottleneck of smps. *Proceedings of the 2004 IEEE/ACM International Conference on High Performance Computing (HiPC'04)*, pages 286–296, 2004.

[D. 96] D. Burger, J. R. Goodman and, A. Kägi. Memory bandwidth limitations of microprocessors. *Proceeding of the 23rd annual international symposium on Computer architecture (ISCA'96)*, pages 78–79, 1996.

[E. 09] E. Ebrahimi, O. Mutlu, and Y. N. Patt. Techniques for bandwitdh-efficient prefetching of linked data structures in hybrid prefetching systems. *HPCA-15*, 2009.

[T. 07] T. K. Prakash, and L. Peng. Performance Characterization of SPEC CPU2006 Benchmarks on Intel Core 2 Duo Processor. 2007.

[USPL07] R. Ubal, J. Sahuquillo, S. Petit, and P. López. Multi2Sim: A Simulation Framework to Evaluate Multicore-Multithreaded Processors. *Proceedings of the 19th International Symposium on Computer Architecture and High Performance Computing*, 2007.

[XW10] D. Xu and C. Wu. On Mitigating Memory Bandwidth Contention through Bandwidth-Aware Scheduling. *Parallel Architectures and Compilation Techniques (PATC'10)*, September 2010.