Arquitectures d'altes prestacions

PID_00250609

Ivan Rodero Castro
Francesc Guim Bernat

Temps mínim de dedicació recomanat: 6 hores

Universitat Oberta de Catalunya

Ivan Rodero Castro

Enginyer d'informàtica i doctor per la Universitat Politècnica de Catalunya. Ha impartit docència a la Facultat d'Informàtica de Barcelona (UPC) d'assignatures dels àmbits d'arquitectura de computadors, sistemes operatius i sistemes paral·lels i distribuïts, tant a grau com a màster i doctorat. Des del 2009 fa docència i recerca a Rutgers University (Nova Jersey), on és també el director associat del Rutgers Discovery Informatics Institute. Consultor dels Estudis d'Informàtica, Multimèdia i Telecomunicació de la Universitat Oberta de Catalunya des del 2010. Centra la seva recerca en l'àrea dels sistemes paral·lels i distribuïts, incloses la computació d'altes prestacions, la *green computing*, la informàtica en núvol i les dades massives.

Francesc Guim Bernat

Enginyer d'Informàtica i doctor per la Universitat Politècnica de Catalunya. Ha impartit docència a la Facultat d'Informàtica de Barcelona (UPC) en assignatures dels àmbits d'arquitectura de computadors, sistemes operatius i sistemes paral·lels i distribuïts, tant a grau, com a màster i doctorat. Des de 2008 fa docència com a consultor dels Estudis d'Informàtica, Multimèdia i Telecomunicació de la Universitat Oberta de Catalunya. Des de l'any 2008 és arquitecte de processadors a la companyia Intel Corporation.

Índex

Introducció

Durant molts anys el rendiment dels processadors ha anat explotant el nivell de paral·lelisme inherent als fluxos d'instruccions d'una aplicació. El 1971 es va posar a la venda el primer microprocessador simple, l'anomenat *Intel 4004*. Aquest processador, de 4 bits, estava format per una sola unitat aritmeticològica: un banc de registres amb 16 entrades i un conjunt de 46 instruccions. El 1974, Intel va fabricar un microprocessador nou de propòsit general de 8 bits, el 8080, que contenia 4.500 transistors i era capaç d'executar un total de 200.000 instruccions per segon.

Des d'aquests primers processadors, que només permetien executar 200.000 instruccions per segon, s'ha arribat a arquitectures tipus Sandy Bridge (Intel, Sandy Bride Intel) o AMDfusion (AMD, pàgina AMD Fusion), que es poden arribar a executar unes 2.300 bilions d'instruccions per segon.

Per tal d'assolir aquest increment en el rendiment, a grans trets es distingeixen tres tipus de millores importants aplicades als primers models esmentats anteriorment:

- Tècniques associades a explotar el paral·lelisme de les instruccions.
- Tècniques associades a explotar el paral·lelisme a nivell de dades.
- Tècniques associades a explotar el paral·lelisme de fils d'execució.

El primer conjunt de tècniques va consistir a mirar d'explotar el nivell de paral·lelisme de les instruccions (en anglès, *instruction level parallelism*, ILP d'ara en endavant). que componien una aplicació. Alguns dels processadors que les van usar són VAX 78032, PowerPC, els Intel Pentium o bé els AMD K5 i AMD K6. Dins aquest primer grup de millores, s'hi troben, entre d'altres, arquitectures súper escalars, execució fora d'ordre d'instruccions, tècniques de predicció, *very long instruction word* (VLIW) o arquitectures vectorials. Aquestes darreres optimitzacions eren força interessants des del punt de vista de l'ILP, ja que es basaven en la possibilitat que tenen els compiladors de millorar la planificació de les instruccions. D'aquesta manera el processador no necessita dur-les a terme.

Lectures complementàries

Sobre la *very long instruction word*, podeu llegir:

J. Fisher (1893). "Very Long Instruction Word Architectures and the ELI-512". A: *Proceedings of the 10th Annual International Symposium on Computer Architecture* (pàg. 140-150).

I sobre arquitectures vectorials:

R. Baniwal (2010). "Recent Trends in Vector Architecture: Survey". *International Journal of Computer Science & Communication* (vol. 1, núm. 2, pàg. 395-339).

El segon conjunt de tècniques va consistir a mirar d'explotar el nivell de paral·lelisme a nivell de dades. En aquests casos s'aplicaven les mateixes instruccions sobre blocs de dades de manera paral·lela, per exemple, la suma o resta de dos vectors d'enters. Aquest tipus de tècniques va permetre augmentar de manera radical la quantitat d'operacions de coma flotant (FLOPS) (en an-

glès, *floating point operations per second*) que les arquitectures podien proporcionar. No obstant això, com es veurà més endavant, aquest model de programació no es pot aplicar de manera genèrica a tots els algorismes de computació.

El tercer conjunt de tècniques intenten explotar el paral·lelisme associat als fils que componen les aplicacions. Durant les dècades de 1990-2010, la quantitat de fils que componen les aplicacions ha augmentat notòriament. Es va passar de l'ús de pocs fils d'execució a l'ús de centenars de fils per aplicació. Un exemple clar d'aquest tipus d'aplicació són les emprades pels servidors, per exemple Apache o Tomcat. No obstant això, també trobem aplicacions multifil (*multithreaded*) en els computadors domèstics. Alguns exemples d'aquestes aplicacions són màquines virtuals com Parallels i VMWare o navegadors com Chrome o Firefox. Per tal de donar suport a aquest tipus d'aplicacions, el maquinari ha anat fent l'evolució natural causada per aquesta demanda creixent de paral·lelisme.

Aquesta tendència es va fer palesa amb els primers multiprocessadors, processadors amb *simultaneous multithreading*, i s'ha confirmat amb l'aparició de sistemes multinuclis. En tots els casos, les aplicacions tenen accés a dos conjunts o més de fils d'execució disponibles dins el mateix maquinari. En les primeres arquitectures aquests fils es trobaven repartits en diferents processadors. En aquests casos, l'aplicació podia executar treballs paral·lels: un en cadascun dels processadors. En les segones arquitectures, un mateix processador proporcionava accés a diferents fils d'execució. Per tant, en aquests escenaris, diferents fils d'execució es poden executar en un mateix processador.

Lligat a aquest segon tipus d'arquitectures, durant la primera dècada del 2000 van aparèixer arquitectures de computació emprades en entorns gràfics (GPU) amb un nivell de paral·lelisme molt alt. Tot i que aquestes arquitectures noves s'orientaven a la renderització gràfica, ràpidament se'n va estendre l'ús en el món de computació d'altes prestacions.

L'evolució d'arquitectures de processadors ha anat vinculada a l'aparició de noves tècniques i paradigmes de programació. En general, les generacions emergents han anat acompanyades de complexitats i restriccions noves que s'han hagut d'incorporar al disseny d'aplicacions pensades per a explotar aquestes presentacions noves. En tots els casos, aquestes funcionalitats noves s'han pogut usar emprant llenguatge de baix nivell o màquina. Per exemple, l'aparició d'unitats vectorials va anar acompanyada d'una extensió del joc d'instruccions amb instruccions per a especificar còmputs amb registres vectorials.

No obstant això, explotar les noves funcionalitats emprant llenguatge de baix nivell no és una cosa factible, tant per complexitat com per productivitat. Per això han anat apareixent models de programació nous que han permès explotar-les alhora que en mitiguen les complexitats. Open-MP, MPI, CUDA, etc. han estat exemples d'aquests models.

D'una banda, en aquest mòdul es presenten els conceptes fonamentals de les diferents arquitectures de computadors més representatives, com també exemples d'aquestes. D'altra banda, s'introdueixen els conceptes més importants en l'estudi de computació d'altes prestacions, com també els paradigmes de programació associats.

Objectius

Els objectius principals que ha d'assolir l'estudiant en acabar aquest mòdul són els següents:

1. Estudiar què és la taxonomia de Flyint i saber com es relaciona amb les diferents arquitectures actuals.

2. Entendre quines arquitectures de computadors hi ha i com es poden emprar en la computació d'altes prestacions.

3. Estudiar què és el paral·lelisme a nivell de dades i saber com les aplicacions poden incrementar el seu rendiment usant aquest paradigma. I dins d'aquest àmbit, entendre què són les arquitectures vectorials i com funcionen.

4. Entendre quins són els beneficis d'usar arquitectures que exploten el paral·lelisme a nivell de fil.

5. Estudiar quin tipus d'arquitectures multifil hi ha i saber-ne les propietats de cadascuna.

6. Entendre com es pot millorar el rendiment dels processadors explotant el paral·lelisme a nivell de fil i a nivell d'instrucció a la vegada.

7. Estudiar quins són els factors lligats al model de programació que cal tenir en compte a l'hora de desenvolupar aplicacions multifil.

8. Estudiar quins són els factors lligats a l'arquitectura d'un processador multifil que cal considerar en el desenvolupament d'aplicacions multifil.

9. Estudiar quines són les característiques del model de programació de memòria compartida i memòria distribuïda.

1. Descomposició funcional i de dades

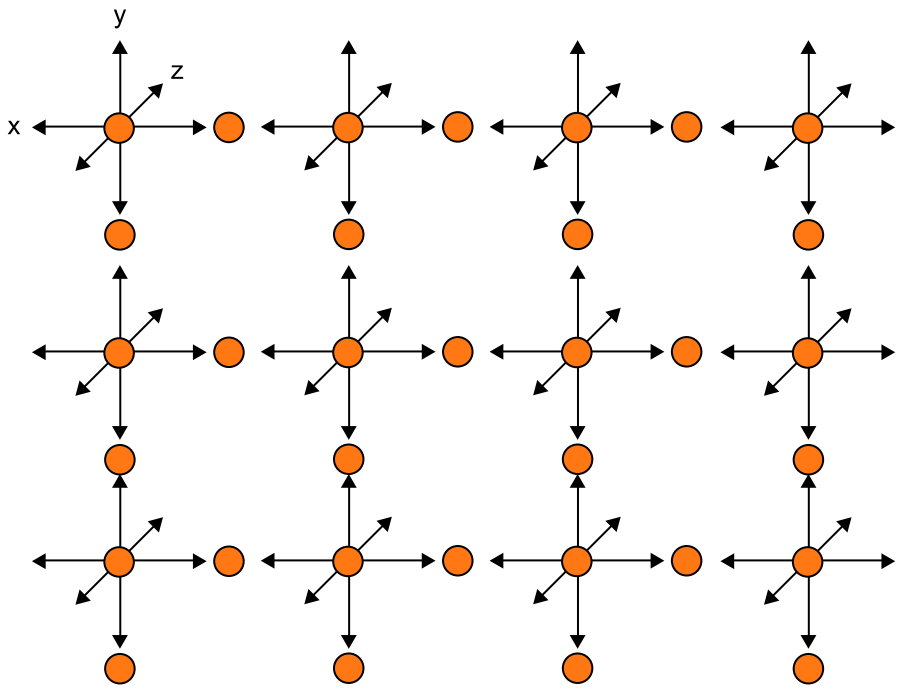
Com es mostra al llarg d'aquesta unitat didàctica la majoria d'arquitectures d'altes prestacions que avui en dia es poden trobar al mercat permeten executar més d'un fil d'execució de manera paral·lela. D'altra banda, algunes també proporcionen accés a unitats de procés especifiques que són capaces d'aplicar un mateix conjunt d'operacions a un conjunt de dades diferents de manera simultània.

Treure rendiment a totes les funcionalitats diferents que faciliten aquestes arquitectures implica que tant les aplicacions com els models de programació s'han d'adaptar a les característiques d'aquestes. D'una banda, els models de programació han de facilitar a les aplicacions maneres d'explicitar fonts de paral·lelisme. De l'altra, cal adaptar els algorismes que implementen les aplicacions a aquestes.

Per tal d'adaptar una aplicació a un model de programació orientat a arquitectures paral·leles cal estudiar-ne les funcionalitats i saber com processa les dades.

Figura 1. Descomposició de dades

$$(x,y) = F(x - 1, x + 1, z - 1, z + 1, y - 1, y + 1)$$



L'exemple anterior mostra un tipus d'aplicació que processa els diferents punts d'una malla segons els seus veïns. En aquest exemple es pot considerar que cada un dels punts pot ser tractat independentment. Per tant, a l'hora de dissenyar una implementació paral·lela per aquesta, es podria descompondre cada processament d'aquests punts de manera independent.

El procés de decidir com es processen les dades d'un problema determinat també es pot anomenar *descomposició de dades*. Aquesta descomposició acostuma a ser la més complexa a l'hora de paral·lelitzar una aplicació, ja que no solament s'ha de decidir com processen les dades els diferents fils d'una aplicació, sinó com aquestes es desen als diferents nivells de memòria cau. Depenent d'on es trobin ubicades en les diferents memòries cau i de la manera com els fils hi accedeixin, el rendiment de l'aplicació es pot veure substancialment minvat.

Descomposició de dades

Alguns exemples molt estesos d'aquests tipus de descomposició són el que s'anomena *partició en blocs per la computació en matrius*. L'objectiu principal acostuma a ser mirar de fer particions del processament de la matriu de manera independent pels diferents fils del processador i mirar de mantenir localitat a les memòries caus on es troben.

El segon dels problemes principals a l'hora de paral·lelitzar una aplicació consisteix a dividir el problema que es vol resoldre en les funcionalitats en què el problema pot ser descompost. Cal fer notar que poden ser executades de manera paral·lela o no.

Lectura recomanada

Es recomana llegir l'article següent, en què s'expliquen diferents algorismes de partició tenint en compte la mida de les memòries cau:

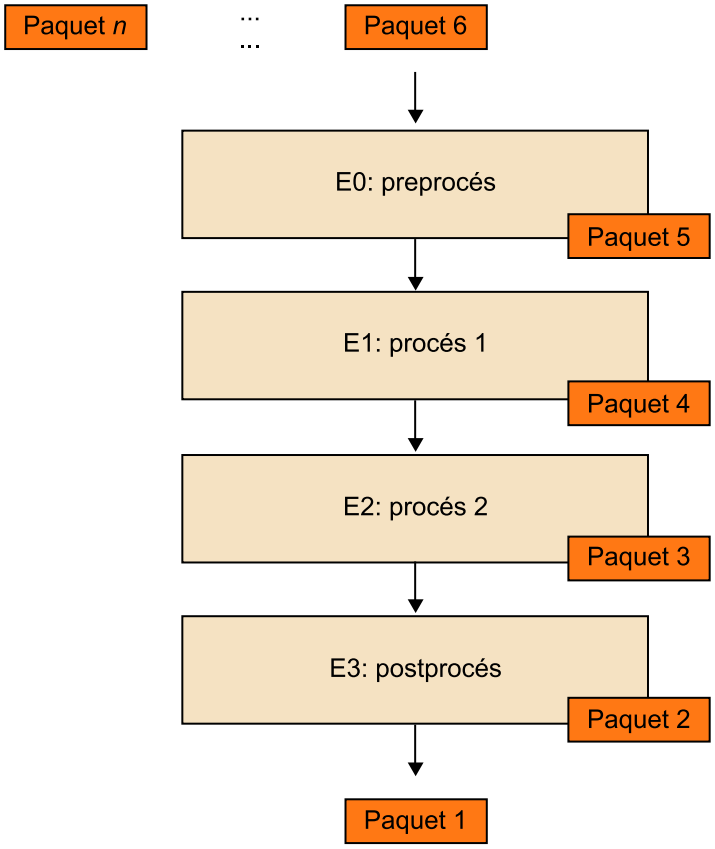
M. S. Lam, E. E. Rothberg i M. E. Wolf (1991).

La figura següent il·lustra un exemple d'aquest tipus de descomposició. Com es pot observar, el problema s'ha dividit en quatre etapes diferents: una de preprocés, dues de procés i una de postprocés. Cada element que és processat per l'algorisme en qüestió passa per cadascuna. No obstant això, en un moment donat podem tenir en cadascuna de les etapes un paquet diferent. Per tant, cadascun s'estarà processant de manera concurrent.

Descomposició funcional del problema

La definició de quines funcions defineixen el problema que es tracta, com es troben relacionades i com hi circulen les dades s'anomena *descomposició funcional del problema*.

Figura 2. Descomposició funcional



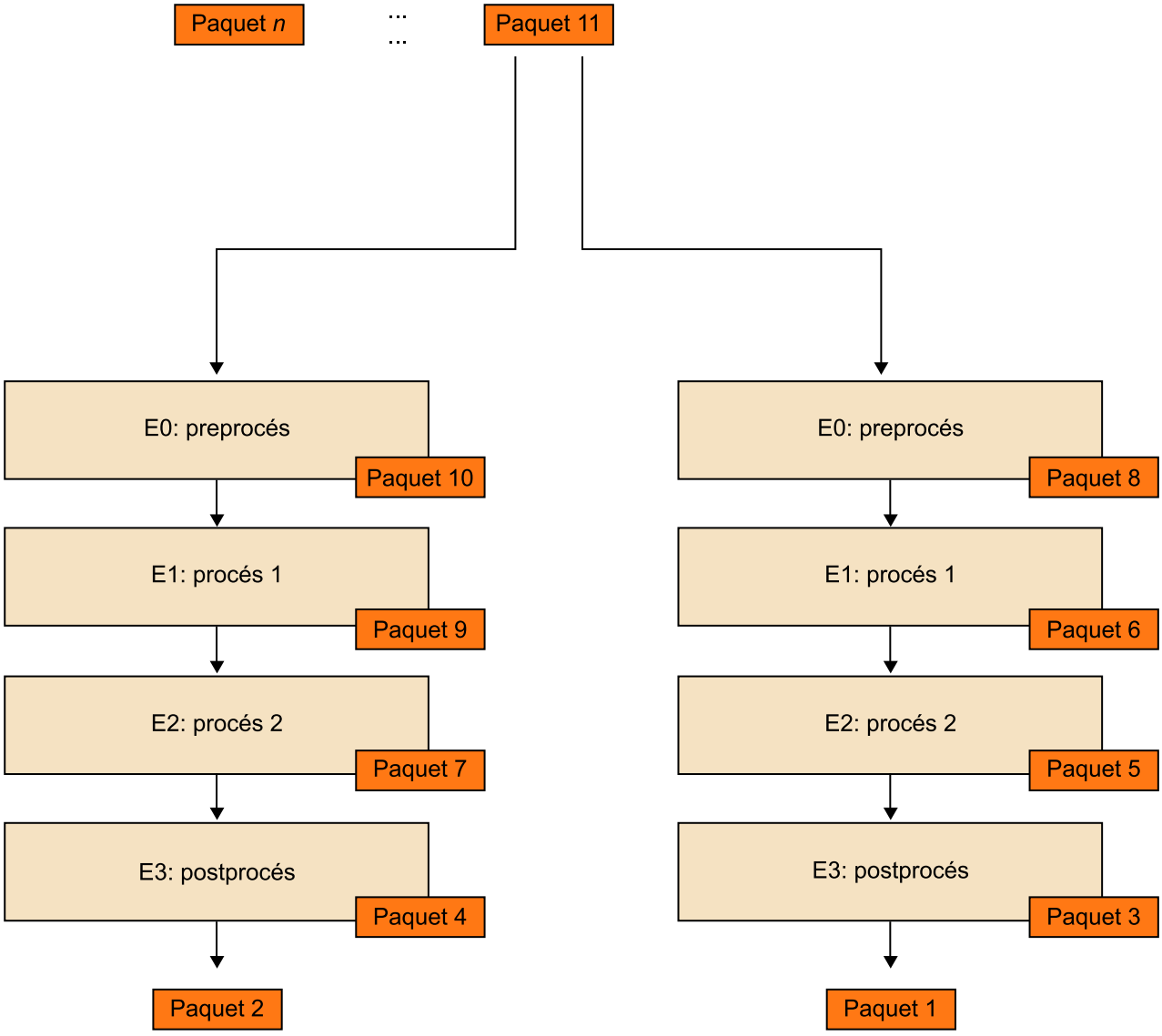
Tot i que els dos exemples anteriors tracten la descomposició de dades i funcional separadament, a la realitat aquest dos problemes estan directament relacionats. A l'hora d'analitzar la paral·lelització d'una aplicació, se n'estudien tant la descomposició funcional com de dades. En general es busca una combinació òptima de totes dues.

La figura següent mostra un exemple en què s'han aplicat tots dos tipus de descomposició. D'una banda, la descomposició funcional ha definit les diferents etapes del nostre algorisme i com cada una pot ser executada independentment de l'anterior. D'una altra banda, s'ha definit una descomposició de dades en què cadascun dels paquets pot ser processat de manera independent de la resta.

Vegeu també

En els apartats següents s'expliquen arquitectures d'altes prestacions en les quals es poden executar aplicacions que tenen paral·lelisme a nivell de fil o de dades. En el cas que es vulgui treure rendiment d'una aplicació determinada en una arquitectura determinada, caldrà estudiar com adaptar el tractament funcional i de dades al sistema on s'executarà.

Figura 3. Paral·lelisme a nivell de dades i funcionalitats



2. Taxonomia de Flynn

Des dels primers processadors fins a les arquitectures actuals s'han proposat molts tipus d'arquitectures de computadors diferents. Des de processadors amb un sol fil d'execució amb un *pipeline* en ordre, fins a processadors amb centenars de nuclis i unitats de computació que exploten el paral·lelisme a nivell de dades.

El 1972, Michael J. Flynn va proposar una classificació de les diferents arquitectures de computadors en quatre categories. Aquesta categorització classifica els computadors segons com gestionen els fluxos de dades i instruccions. És molt interessant considerant l'època en què es va fer i com es pot aplicar en les arquitectures d'avui en dia. Tot i que va ser definida fa trenta anys, la majoria de processadors actuals es poden incloure en alguna d'aquestes quatre categories. A més, molts dels treballs acadèmics han seguit utilitzant aquesta nomenclatura fins al dia d'avui. Les quatre tipus d'arquitectures que Flynn va descriure són les següents:

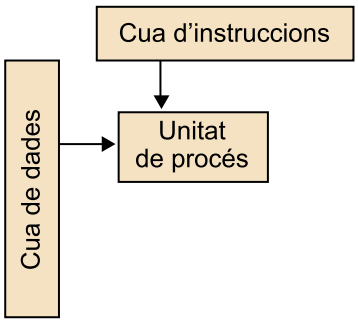
1) Una instrucció, una dada[1]: són les arquitectures tradicionals formades per una sola unitat de procés. Com es pot observar a continuació, no exploten ni el paral·lelisme a nivell de fil, ni el paral·lelisme a nivell de dades. En aquest cas, tenim un sol flux de dades i d'instruccions. La figura següent il·lustra el model abstracte d'un computador SISD.

[1] En anglès, *single instruction, single data stream* (SISD).

Computador SISD

Exemples d'aquest tipus de processador són IBM 370 (IBM, System/370 Model 145), Intel 8085 (Intel, 2011) o el Motorola 6809 (Hennessy i Patterson, 2011).

Figura 4. Una instrucció, una dada



2) Una instrucció, múltiples dades[2]: són les arquitectures en què una mateixa instrucció s'executa en múltiples unitats de procés de manera paral·lela usant diferents fluxos (*streams*) de dades. Cada processador té la seva pròpia memòria amb dades; per tant, tenim diferents dades en global, però la memòria d'instruccions i unitat de control són compartides. Les instàncies d'aquest tipus d'arquitectures més famoses són les unitats de procés vectorials. Aquest tipus d'arquitectures són emprades avui en dia en la majoria de processadors d'altes prestacions, ja que permeten fer càlculs sobre grans volums de dades de manera paral·lela. Així és possible incrementar notòriament els FLOPS d'un

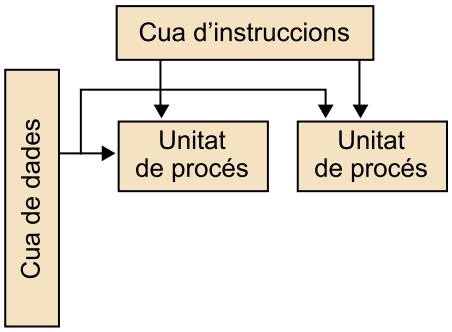
[2] En anglès, *single instruction, multiple data stream* (SIMD).

Unitats de procés vectorials

Els processadors de l'empresa Cray (insideHPC, *InsideHPC: A Visual History of Cray*) són exemples d'aquest tipus d'arquitectures.

processador. D'altra banda, com veurem més endavant, aquest paradigma de computació també s'empra dins de les arquitectures dedicades a processament d'imatge o de gràfics.

Figura 5. Una instrucció, múltiples dades



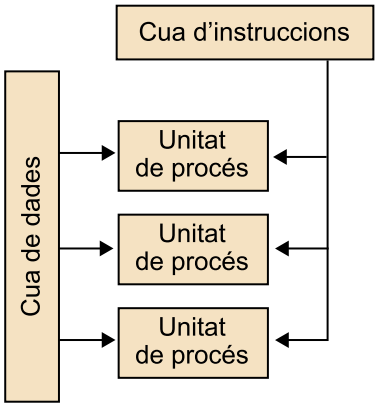
3) Múltiples instruccions, una dada[3]: permeten executar diferents fluxos d'instruccions sobre un mateix bloc de dades. En aquest cas, tenim diferents unitats de procés que operen sobre la mateixa dada. Aquest tipus d'arquitectures és molt específic i no se'n troba cap dins del món de l'arquitectura de processadors d'altes prestacions o de propòsit general. No obstant això, s'han fet implementacions de multiprocessadors de propòsit específic que segueixen aquesta definició.

(3)En anglès, *multiple instruction, single data stream* (MISD).

Multiprocessadors de propòsit específic

Un exemple d'això és un processador en què les unitats de procés es troben replicades per tal de tenir redundància. Aquest tipus d'arquitectures són especialment útils en entorns en què la fiabilitat és el factor més important, com, per exemple, l'aviació. Un altre exemple seria el que s'anomena *pipeline image processing*: cada punt de la imatge és processat per diferents unitats de procés, en què cadascuna aplica un tipus de transformació diferent sobre aquest.

Figura 6. Múltiples instruccions, una dada

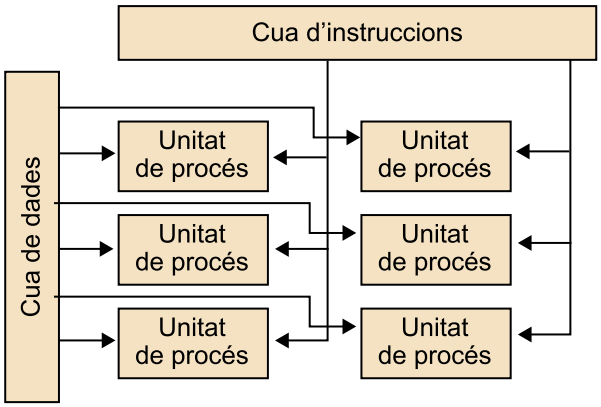


4) Múltiples instruccions, múltiples dades[4]: estan formades per múltiples unitats de procés que treballen sobre diferents fluxos de dades de manera paral·lela. Com mostra la figura següent, l'arquitectura està formada per diferents unitats de procés que tenen diferents fluxos d'instruccions que treballen

(4)En anglès, *multiple instruction, multiple data stream* (MIMD).

amb diferents fluxos de dades. Aquest és el clar paradigma de computació actual en què les arquitectures permeten processar diferents fluxos de dades amb diferents fluxos de computació.

Figura 7. Múltiples instruccions, múltiples dades



Les arquitectures MIMD faciliten l'accés a múltiples fils de computació i múltiples fluxos de dades. Els fluxos de computació es poden trobar completament aïllats entre si o bé compartir recursos i contextos. En el primer dels casos estarem parlant de diferents processos, en què cadascun dels processos conté un context d'execució completament aïllat de la resta. En el segon cas, estarem parlant de fils d'execució o *threads*, en què diferents fils d'execució comparteixen un mateix context. És a dir, comparteixen les dades i les instruccions.

És important fer notar que algunes de les arquitectures que es poden trobar en l'actualitat segueixen més d'un dels paradigmes anteriors. Per exemple, es poden trobar arquitectures multinucli que contenen unitats vectorials per a fer còmputs amb vector de manera paral·lela. Per tant, en un mateix processador trobarem parts SIMD i MIMD. Les arquitectures que es poden trobar dins el món de computació d'altes prestacions són bàsicament arquitectures SIMD i MIMD.

Vegeu també

Els propers apartats introdueixen els conceptes més importants de les arquitectures més freqüents en les arquitectures d'altes prestacions: les SIMD i les MIMD. Per a cadascuna, es discuteix un exemple de processador real per tal d'estudiar arquitectures reals.

3. Arquitectures de processador SIMD

Les arquitectures SIMD, com ja hem dit, es caracteritzen perquè processen diferents fluxos de dades amb un sol flux d'instruccions. El tipus d'arquitectures més conegudes d'aquesta família són les arquitectures vectorials.

La majoria de processadors d'altes prestacions tenen unitats específiques de tipus vectorial. Com explica aquesta secció, aquests tipus d'arquitectures tenen certes propietats que les fan molt interessants per a dur a terme càlculs científics i processar grans volums de dades.

Exemple de suma vectorial

```
Paràmetres:
VectorEnters[1024] A;
VectorEnters[1024] B;
VectorEnters[1024] C;

Codi:
per(i = 0; i < 1024; i++)
   C[i] = A[i]+B[i];

Paràmetres:
VectorEnters[1024] A;
VectorEnters[1024] B;
VectorEnters[1024] C;

Codi:
RegistreVectorial512 a,b,c;

Per(i=0; i<512; i+=16)
{
   carrega(a,A[i]); carrega(b,B[i]);
   c = suma_vectorial(a,b);
   desa(C[i],c);
}
```

Com indica el nom mateix, a diferència de les SISD, aquest tipus d'arquitectures operen a nivell de vectors de dades. Per tant, a diferència de les primeres, amb una sola instrucció podem fer la suma de dos registres que són capaços de guardar *k* elements diferents.

Per exemple, una unitat vectorial podria treballar amb registres de 512 bytes. En aquest cas, un vector podria guardar 16 elements de tipus *float* (assumint que un *float* ocupa 32 bytes) i, per tant, amb una sola instrucció podria sumar dos blocs de 16 elements.

L'exemple de codi anterior mostra com es podria vectoritzar la suma de dos vectors A i B que es guarda en un tercer vector C (cada vector de 1.024 elements enters). Aquest exemple és senzill, però n'hi ha prou per tal de donar una idea de l'increment de rendiment que les aplicacions poden obtenir si poden usar correctament les unitats vectorials que el processador conté.

El primer dels codis mostra la clàssica suma de vectors tal com la faria un processador escalar normal. En aquest cas, l'aplicació farà 1.024 iteracions. En cada iteració, el processador llegirà el valor enter de la posició que s'està processant dels vectors a, b i c, en sumarà a i b, i finalment l'escriurà a C. El segon dels codis mostra com es pot fer la mateixa suma de vectors si disposem d'una unitat vectorial amb registres de 512 bytes. En aquest cas, com que treballem amb elements enters de 32 bytes, dins de cada registre vectorial podem guardar 16 elements. D'aquesta manera, a cada iteració es llegeixen 16 elements del vector A i B, es desen als dos registres vectorials a i b, se sumen al registre vectorial c, i finalment s'escriuen els 512 bytes del vector a la posició corresponent del vector C.

Com es pot observar, al codi vectorial l'increment de l'índex és de 16 en 16. Per tant, per cada iteració del codi vectorial se'n han de fer 16 del codi escalar. Tot i que el codi és senzill, dóna una idea del potencial d'aquest tipus d'arquitectures per a processar estructures de tipus vectors o matrius i de la millora que poden donar respecte d'unitats escalars tradicionals.

Cal fer notar que això no és tan sols un mecanisme programari. És a dir, el sistema i les biblioteques proporcionen un conjunt d'interfícies que permeten operar amb els blocs de 512 bytes, però aquestes crides s'acaben transformant al llenguatge natiu del processador, que és capaç d'operar amb aquests blocs. Cada processador vectorial facilita un conjunt d'instruccions específiques[5] a aquest per a operar amb les dades de tipus vectorial. Per tant, el tipus d'operacions vectorials que es podrà emprar en una arquitectura concreta dependrà de la ISA vectorial que aquesta proporcioni.

[(5)]En anglès, *instruction set architecture* (ISA).

En l'exemple anterior, si la ISA del processador vectorial no facilita una suma de dos registres vectorials, el compilador probablement traduirà la suma de dos registres a 16 sumes escalars consecutives. No obstant això, en cas afirmatiu, el compilador traduirà la suma dels dos registres en una sola instrucció assemblador que farà la suma dels dos elements.

L'exemple anterior mostra la vectorització d'un codi extremadament senzill. Però el procés d'adaptar el codi de l'aplicació per tal que aquesta treballi amb blocs de 512 bytes pot no ser factible en totes les aplicacions. Per exemple, això pot ser difícil o impossible en algorismes que treballen en estructures de dades representades en grafs. D'altra banda, en aplicacions científiques que treballen amb dades matricials, aquest tipus de processament és força habitual. No obstant això, en molts casos vectoritzar el codi és altament complex o impossible.

Lectura recomanada

Aquest subapartat no explica detalladament com vectoritzar aplicacions ni les arquitectures vectorials. Si voleu aprofundir en aquest àmbit, us recomanem que llegiu el llibre següent:

G. Sabot (1995). *High Performance Computing: Problem Solving with Parallel and Vector Architectures*. Reading, Massachusetts: Addison-Wesley.

3.1. Propietats dels processadors vectorials

Com acabem de veure, els processadors o unitats vectorials poden millorar substancialment el rendiment de les aplicacions quan poden ser vectoritzades. No obstant això, el benefici de fer operacions vectorials no només rau a poder fer *m* sumes o restes paral·leles. El fet de poder operar un conjunt de dades en bloc dóna als processadors vectorials certes característiques interessants:

a) Treballar amb blocs de dades fa que el compilador o el programador especifiqui al processador que no hi ha dependències entre els diferents elements que componen els vectors. Per tant, el processador no ha de fer validacions sobre riscos estructurals o dades entre els diferents elements dels vectors. Si és un processador escalar normal, el processador hauria de verificar que no hi ha riscos entre les diferents operacions dels elements dels vectors. No obstant això, el processador ha de computar i validar dependències només entre les operacions dels registres vectorials.

b) Per a dur a terme el càlcul paral·lel dels diferents elements dels registres vectorials, el processador pot emprar unitats funcionals replicades de manera paral·lela per a cadascun dels elements dels registres.

Suma de registres

Per a sumar dos registres de 512 bytes que contenen 16 elements enters, per exemple, el processador pot tenir 16 unitats de suma que poden computar en paral·lel la suma de cadascuna de les posicions.

c) En general, els processadors faciliten un conjunt d'instruccions vectorials força extens. En aquest subapartat només s'han esmentat les operacions de memòria i aritmètiques típiques (*add*, *sub*, *load*, *store*, etc.). Però el joc d'instruccions que acostumen a proporcionar és força més complet i complex en alguns casos.

Jocs d'instruccions

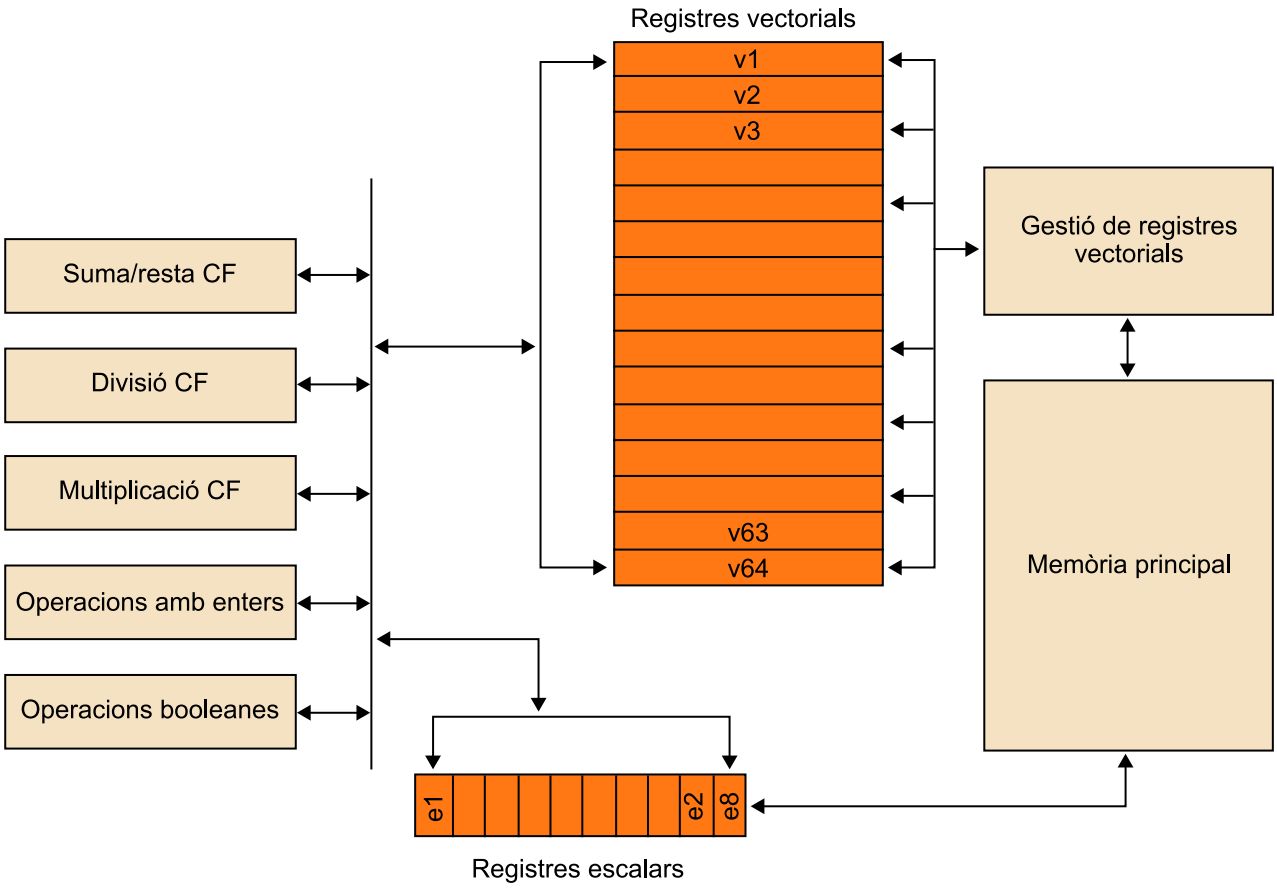
Un exemple en són els *broadcasts* o els *shuffles*. Alguns dels jocs d'instruccions més coneguts són les *advanced vector extensions* (AVX; I. Corp, 2011) o les *streaming SIMD extensions* (SSE; I. Corp, 2007).

Tots els punts discutits fan que les arquitectures vectorials siguin molt atractives per a aplicacions científiques o aplicacions d'enginyeria. Algunes de les aplicacions que fan un bon ús d'aquest tipus de prestacions són les simulacions de xocs de cotxes o les simulacions de temps. Tots dos tipus d'aplicacions empren grans volums de dades i poden executar-se en supercomputadors durant períodes de temps llargs. Un altre tipus d'aplicacions que es beneficien molt d'aquest tipus d'arquitectures són les aplicacions multimèdia, que es caracteritzen per un nivell abundant de paral·lelisme a nivell de dades.

3.2. Exemple de processador vectorial

La figura següent mostra un exemple de possible processador vectorial. Aquest subapartat presenta a alt nivell els diferents elements que podrien compondre un processador d'aquest tipus.

Figura 8. Exemple de processador vectorial



Com es pot observar, està compost per dos tipus de banc de registres diferents. El primer bloc són els registres. D'una banda, s'hi troben els vectorials, que poden guardar 512 bytes d'informació. Cadascun desa el que hem anomenat com a *vector*, és a dir, un conjunt d'elements de coma flotant o enters. En el cas que s'hi vulguin desar enters, un registre pot guardar un total de 16 elements (32 bytes per element). Si s'hi volen desar elements de coma flotant, s'hi poden desar un total de 8 elements (64 bytes per element). D'altra banda, el processador també disposa d'un conjunt de registres escalars. Algunes de les instruccions vectorials poden usar escalars com a part de l'operació o poden generar resultats escalars.

Lectura recomanada

Recordeu que si voleu aprofundir més en el disseny i la implementació de les diferents etapes que el componen, tipus d'operacions i funcionament d'accés a memòria, heu de llegir l'obra següent:

G. Sabot (1995). *High Performance Computing: Problem Solving with Parallel and Vector Architectures*. Reading, Massachusetts: Addison-Wesley.

El segon bloc són les unitats funcionals, que es divideixen en tres grans blocs. El primer de tot són les unitats funcionals dedicades a dur a terme el còmput sobre vectors que contenen elements de tipus coma flotant o que es volen operar com a coma flotant. El segon conjunt inclou la unitat funcional dedicada a fer el càlcul sobre registres vectorials que desen dades enteres. I, finalment, hi ha la unitat funcional que s'encarrega de dur a terme operacions booleanes sobre els registres vectorials.

El tercer i darrer bloc inclou els elements orientats a gestionar l'accés a memòria (ja sigui L1/L2 o memòria principal) i la transferència de dades d'aquesta als registres vectorials. En el cas dels registres vectorials, com que es treballen en blocs de 512 bytes, es disposa d'una unitat específica que s'encarrega de

gestionar-ne l'accés a memòria. Els registres escalars funcionen tal com funcionarien en un processador escalar normal. Per tant, no disposem d'una màquina específica per a dur a terme aquesta càrrega.

El processador vectorial introduït en aquest subapartat és un model bàsic d'arquitectura vectorial. De processadors vectorials se'n poden trobar molts durant les darreres dècades.

Processadors Cray i Sandy Bridge

Un exemple important de processadors vectorials són els Cray (insideHPC, *InsideHPC: A Visual History of Cray*), que van ser sobretot importants durant la dècada dels vuitanta i noranta i se'n van dissenyar fins a sis models diferents. Durant la primera dècada del 2000 Cray va anunciar diferents sistemes que, tot i que no eren exclusivament vectorials, tenien una part important del maquinari orientada a aquest tipus de processament.

El primer de tots va ser el Cray-1, que va ser presentat l'any 1976. S'executava a una freqüència de rellotge de 80 MHz i disposava de 8 registres de tipus vectorial. Cadascun d'aquests registres estava compost per 64 elements de 64 bits (8 bytes). Per a fer operacions vectorials disposava de sis unitats de computació vectorials diferents: coma flotant de suma, coma flotant de multiplicació, unitat per a desplaçar elements del vector, una unitat entera de suma, una unitat per a dur a terme operacions lògiques i finalment una unitat dedicada a fer unes operacions específiques anomenades *recíproques*.

Com ja s'ha esmentat, molts dels processadors actuals, tot i no ser processadors totalment orientats al processament vectorial, porten unitats vectorials. El processador de l'empresa Intel, Sandy Bridge (Intel, 2012), n'és un exemple. Dins el seu joc d'instruccions, el Sandy Bridge incorpora el joc d'instruccions ja esmentat AVX, a part de les SSE anteriors. No obstant això, els processadors Intel no són els únics que incorporen instruccions vectorials dins el joc d'instruccions. Els processadors de l'empresa AMD també ho fan.

4. Arquitectures de processador multifil o MIMD

Durant les primeres dècades de desenvolupament de microprocessadors (1970 i 1980), l'objectiu principal es va centrar a explotar al màxim el paral·lelisme de les aplicacions a nivell d'instrucció (SISD), anomenat també *instruction level paralellism* (ILP). Durant aquestes dècades es va mirar de treure rendiment a les aplicacions amb tècniques que permetien executar les instruccions fora d'ordre, tècniques de predicció més precises o jerarquies de memòria de més capacitat.

No obstant això, el rendiment de les aplicacions no escala proporcionalment amb la quantitat de recursos afegits.

Per exemple, passar d'una arquitectura que permet començar dues instruccions per cicle a quatre per cicle no implica necessàriament doblar el rendiment del processador. Fins i tot, en molts casos, encara que s'hi afegeixin molts més recursos, el rendiment de l'aplicació només s'incrementa lleugerament.

Aquest factor és el que va motivar l'aparició d'arquitectures MIMD, que consideren l'execució simultània de diferents fils d'execució en un mateix processador: paral·lelisme a nivell de fil[6] (Lo, 1997). En aquestes arquitectures:

[6]En anglès, *thread level paralelism* (TLP).

- Diferents fils d'execució comparteixen les unitats funcionals del processador (per exemple, unitats funcionals).
- El processador ha de tenir estructures independents per a cadascun dels fils que executa: registre de *renaming*, comptador de programa, etc.
- Si els fils pertanyen a diferents processos, el processador ha de facilitar mecanismes perquè puguin treballar amb diferents taules de pàgines.

Aplicacions paral·leles

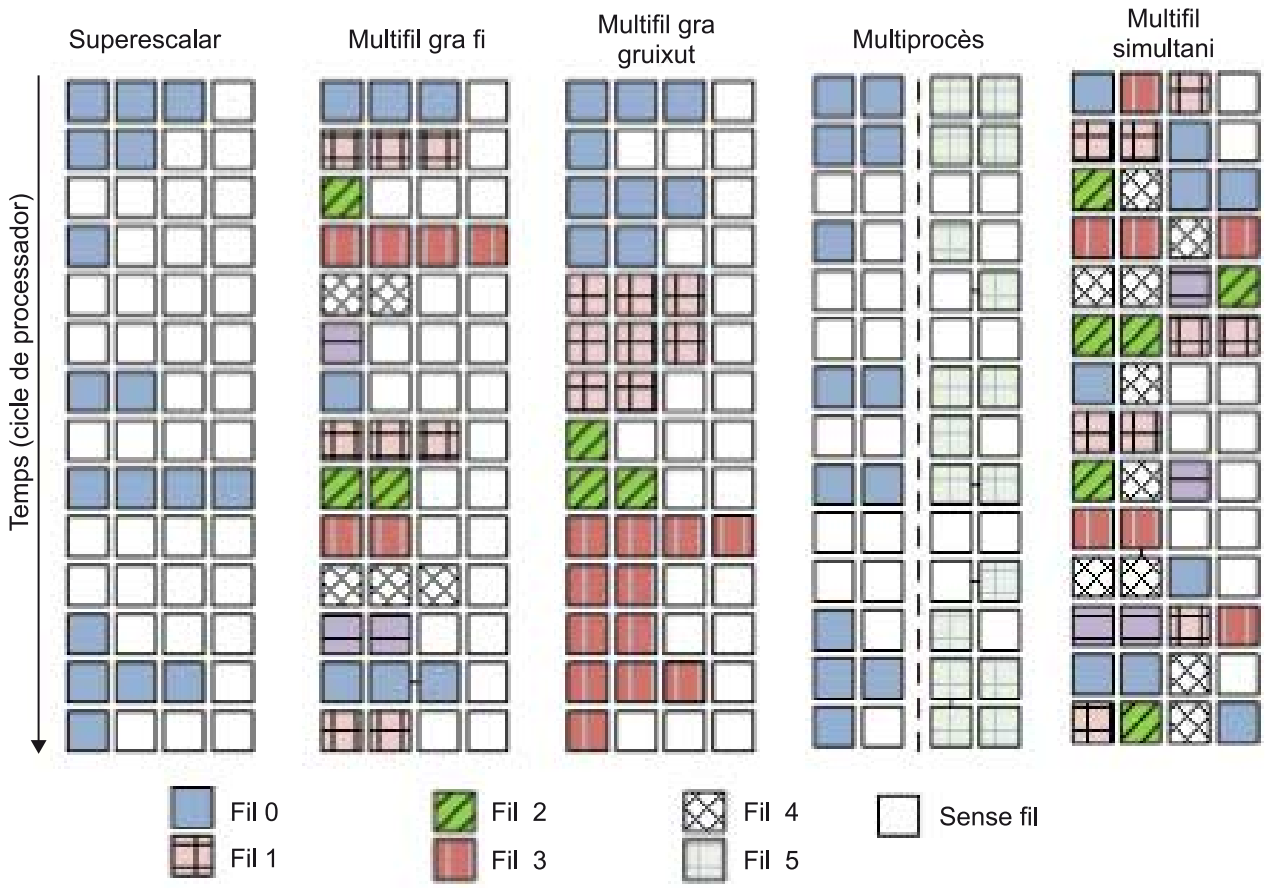
Els navegadors, les bases de dades, els servidors web, etc. són exemples d'aquest tipus d'aplicacions.

Com es mostra a continuació, les arquitectures actuals exploten aquest paradigma de moltes maneres diferents. No obstant això, la motivació és la mateixa: la majoria d'aplicacions actuals tenen un alt nivell de paral·lelisme i el rendiment puja afegint més paral·lelisme a nivell de processador. L'objectiu és millorar la productivitat dels sistemes que poden executar aplicacions que són inherentment paral·leles.

En aquests entorns actuals, treure rendiment explotant el TLP pot ser molt més efectiu en termes de cost/rendiment que explotar l'ILP. En la majoria de casos, com més paral·lelisme es faciliti, més rendiment se'n pot treure. En qualsevol cas, la majoria de tècniques que s'havien emprat per a sistemes ILP també es fan servir en sistemes TLP.

A continuació es presenten els diferents tipus d'arquitectures TLP que s'han proposat durant les darreres dècades. La figura 9 mostra com s'executarien diferents fils d'execució en cadascuna de les arquitectures introduïdes a continuació.

Figura 9. Model ILP enfront de models TLP



1) Arquitectures multiprocessador (MP). Aquesta és l'extensió més senzilla a un model ILP. En aquest cas repliquem una arquitectura ILP *n* vegades. El model més bàsic d'aquestes arquitectures és el multiprocessador simètric. El desavantatge principal és que, tot i que té molts fils d'execució, cadascun segueix tenint les limitacions d'un ILP. No obstant això, com es mostra més endavant, hi ha variants en què cada processador és alhora multifil.

2) Arquitectures multifil, també conegudes com a *super-threading*. Aquesta va ser la següent de les variants TLP que va aparèixer. Al model de *pipeline* del processador s'estén considerant també el concepte de *fil d'execució*. En aquest cas, el planificador (que escull quina de les instruccions comença en aquest cicle) té la possibilitat de triar quin dels fils d'execució comença la instrucció següent en el cicle següent.

TERA Systems

Per exemple, TERA Systems (Alverson, Callahan, Cummings i Koblenz, 1990) podia treballar amb 128 fils a la vegada.

3) Arquitectures amb execució de fil simultània[7]. Són una variació de les arquitectures multifil que permeten a la lògica de planificació escollir instruccions de qualsevol fil a cada cicle de rellotge. Aquesta condició fa que la utilit-

(7)En anglès, *simulatenous multithreading* (SMT).

zació dels recursos sigui molt més elevada i eficient. El desavantatge més gran d'aquest tipus d'arquitectures és la complexitat de la lògica necessària per a dur a terme aquesta gestió. El fet de poder començar diverses instruccions de diferents fils és molt costós. Per això el nombre de fils que aquestes arquitectures acostumen a fer servir és relativament baix.

Hyperthreading

Per exemple, les arquitectures Intel amb *hyperthreading* (Marr, 2002) implementen dos o més fils d'execució, o bé l'Alpha 21464 (Seznec, Felix, Krishnan i Sazeide, 2002) implementa quatre fils d'execució.

4) Arquitectures *multicore*[8]. Conceptualment són similars a les primeres arquitectures esmentades, però a escala més petita. En el cas de sistemes MP hi ha *n* processadors independents que poden compartir la memòria, però no comparteixen recursos entre si, com ara una memòria cau de darrer nivell. Els SOC són una evolució conceptual dels MP però traslladada a nivell de processador. En un SOC, un mateix processador es compon per *m* nuclis en què es poden executar fils que poden estar relacionats o fins i tot poden compartir recursos.

(8)Anomenades *chip-multiprocessor* (CMP) o *system-on-chip* (SOC).

Els propers subapartats estudien cadascuna d'aquestes arquitectures.

4.1. Arquitectures *superthreading*

El TLP treu rendiment perquè comparteix els recursos entre diferents fils d'execució. No obstant això, hi ha dues maneres de dur a terme aquesta compartició. La primera consisteix a compartir els recursos en l'espai i el temps, és a dir, en un cicle concret i en una etapa concreta del processador, instruccions de fils diferents poden estar compartint les mateixes etapes del processador. La segona consisteix a compartir els recursos en el temps; és a dir, en un cicle concret i en una etapa concreta del processador, només s'hi poden trobar instruccions d'un mateix fil. Aquest segon tipus de compartició és coneguda com a *superthreading*.

Dins les arquitectures *superthreading* hi ha dues maneres de compartir els recursos en el temps entre els diferents fils. Les més habituals són compartició a nivell fi o compartició a nivell gruixut.

4.1.1. Compartició a nivell fi

En la compartició de recursos a nivell fi, el processador canvia de fil a cada instrucció que aquest executa. Així l'execució dels fils es fa de manera intercalada. Habitualment el canvi de fil es fa seguint una distribució *round robin*, és a dir, s'executa una instrucció del fil *n*, després del $n + 1$, del $n + 2$, etc. En els casos en què un fil està bloquejat, per exemple esperant dades de memòria, se salta i es passa al següent. Per tal de poder dur a terme aquest tipus de canvis, el processador ha de ser capaç de fer un canvi de fil per cicle.

L'avantatge d'aquesta opció és que pot esmorteir bloquejos curts i llargs dels fils que s'estan executant, ja que a cada cicle es canvia de fil. El desavantatge principal és que es redueix el rendiment dels fils de manera individual, ja que

fils preparats per a executar instruccions queden endarrerits per les instruccions dels altres fils. Això té implicacions de complexitat de disseny i de consum d'energia, ja que el processador ha de poder canviar de fil.

UltraSPARCT1

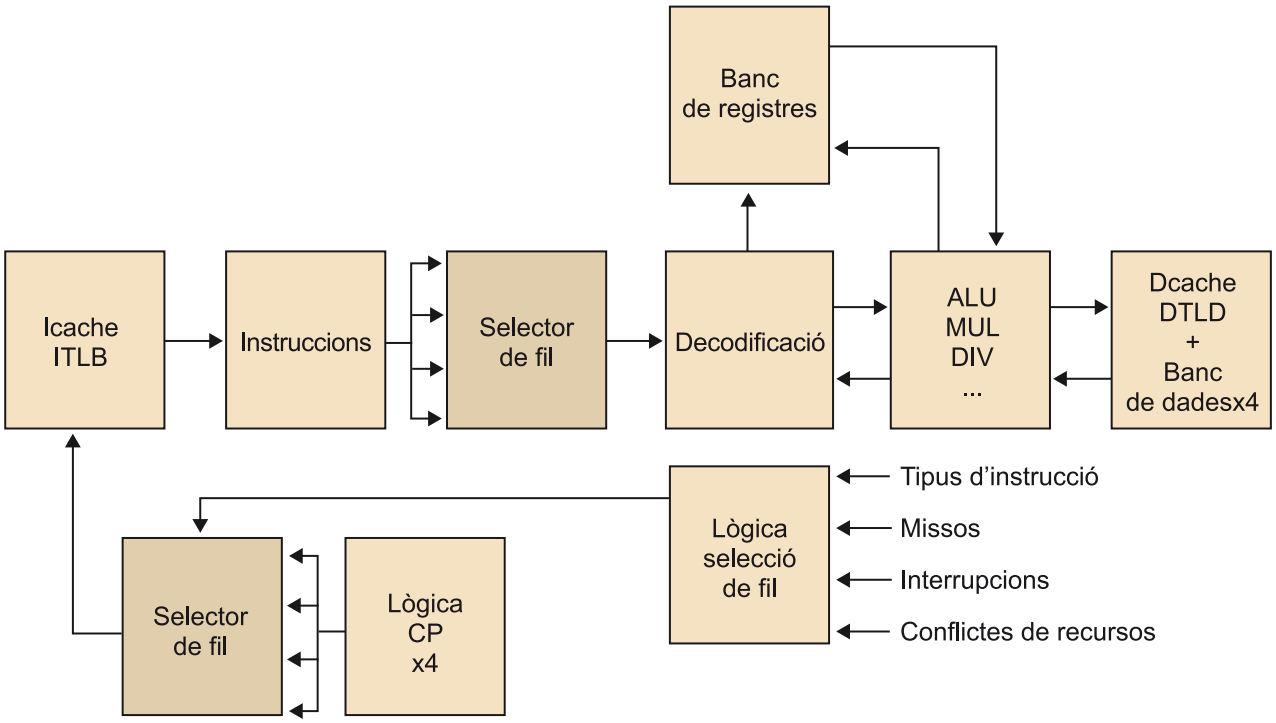
L'UltraSPARCT1 (Kongetira, Aingaran i Olukotun, 2005) és un exemple d'aquest tipus de processador.

El processador UltraSPARCT1, també conegut com a *Niagara*, va ser anunciat per l'empresa SunMicrosystems el novembre de 2005. Aquest nou processador UltraSPARC era multifil i multinucli, dissenyat per arquitectures de tipus servidor amb un consum energètic baix. Per exemple, a 1,4 GHz consumeix aproximadament uns 72 W. Anteriorment, l'empresa Sun ja havia introduït dues arquitectures multinucli: l'UltraSPARC IV i el IV+. No obstant això, el T1 va ser el primer microprocessador que era multinucli i multifil. Es poden trobar versions amb quatre, sis o vuit nuclis i cadascun pot encarregar-se de quatre fils concurrentment. Això implica que es poden executar fins a un màxim de 32 fils concurrentment.

La figura 10 mostra el *pipeline* que té cadascun d'aquests nuclis. Com es pot observar, conté un selector que decideix quin fil s'executa en cada cicle. Aquest selector va canviant de fil a cada cicle. En qualsevol cas, només escull sobre el conjunt de fils disponibles en cada moment. Quan hi ha un esdeveniment de llarga latència (com per exemple, una fallada de memòria cau que desencadena una petició a memòria), el fil que l'ha causat es treu de la cadena de rotació[9]. Un cop aquest esdeveniment de llarga durada acaba, el fil es torna a afegir a la rotació per a accedir a les unitats funcionals. El fet que el *pipeline* es comparteixi amb diferents fils cada cicle fa que cada fil vagi més lent, però la utilització del processador és més elevada. Un altre efecte molt interessant és que l'impacte de fallades de les memòries cau és molt més reduïda: si un dels fils en causa una, els altres poden seguir usant els recursos i progressar.

(9)En anglès, *round robin*.

Figura 10. *Pipeline* UltraSPART T1



Com mostra la figura anterior, donat un cicle, tots els elements del *pipeline* són usats per només un fil. No obstant això, algunes de les estructures estan replicades pel nombre de fils que el nucli té, com, per exemple, la lògica de gestió del comptador de programa (CP). En aquest cas, el nucli ha de ser capaç de distingir en quina part del flux es troba cada fil. Per tant, necessita tenir aquesta estructura per a cadascun dels fils. Contràriament, la resta de lògica, com per exemple la lògica de descodificació, és única i només usada per un sol fil donat un cicle.

4.1.2. Compartició a nivell gruixut

En la compartició a nivell gruixut els canvis de fils es fan quan, en el fil que s'executa, hi ha un bloqueig de llarga durada.

Bloqueig de llarga durada

Un exemple d'aquest tipus de bloqueig és una fallada a la memòria cau de darrer nivell. En aquest cas caldria anar a memòria, fet que implicaria molts cicles de bloqueig.

L'avantatge d'aquesta opció és que no es redueix el rendiment del fil individual, perquè només canvia de fil quan aquest es bloqueja per un esdeveniment que requereix una latència llarga per a ser processat. El desavantatge és que no és capaç de treure rendiment quan els bloquejos són més curts. Com que el nucli només genera instruccions d'un sol fil en cada cicle, quan aquest es bloqueja en un cicle concret (per exemple per una dependència entre instruccions) totes les etapes següents del processador es queden bloquejades o congelades i es tornen a emprar tan bon punt el fil pugui tornar a processar instruccions.

Un altre desavantatge important és que els fils nous que comencen al processador han de passar per totes les etapes abans que el processador comenci a retirar instruccions per aquest fil. En el model anterior, com que cada cicle es canvia de fil, el rendiment de la productivitat no es veu tan afectada. Per exemple, suposem un processador amb 12 cicles d'etapes i 4 fils d'execució. En el millor dels casos, un fil nou tardarà 12 cicles a retirar la primera instrucció. Però durant aquest 12 cicles, en el millor dels casos, els altres 3 fils han pogut retirar 12 instruccions. En canvi en el gra gruixut hauríem estat 12 cicles sense retirar-ne cap.

IBM AS/400

L'IBM AS/400 (IBM, 2011) és un exemple d'aquest tipus de processador.

4.2. Arquitectures *simultaneous multithreading*

Les arquitectures amb multifil simultani[10] (Tullsen, Eggers i Levy, 1995) són una variació de les arquitectures tradicionals multifil. Aquestes arquitectures noves permeten escollir instruccions de qualsevol dels fils que el processador està executant en cada cicle de rellotge. Això vol dir que diferents fils estan compartint en un mateix instant de temps diferents recursos del processador. Aquesta compartició és tant horitzontal (per exemple, etapes de successives del *pipeline*) com vertical (per exemple, recursos d'una mateixa etapa del processador).

(10)En anglès, *simultaneous multithreading* (SMT).

El resultat d'aquestes tècniques és una alta utilització dels recursos del processador i molta més eficiència. Cal recordar que, a diferència de les arquitectures paral·leles anteriors, en un mateix instant de temps dos fils poden estar compartint els mateixos recursos d'una de les etapes del processador. No obstant això, aquesta eficiència no és gratuïta, és a dir, per a donar suport a aquesta compartició el processador conté una lògica extra i força complexa. Cal fer notar que, per defecte, els fils de diferents processos no s'han de poder veure entre si, tant per temes de seguretat com de funcionalitat, l'execució d'una instrucció A d'un fil X no pot modificar el comportament funcional d'un fil B. Cal recordar que, en un sistema operatiu, cada procés té el seu propi context i que aquest és independent dels contextos dels altres processos, sempre que no s'usin tècniques explícites per a compartir recursos.

Les arquitectures que són totalment SMT i que són capaces de treballar amb molts fils són extremadament costoses en termes de complexitat. És important tenir en compte que les estructures necessàries per tal de suportar aquest tipus de compartició creix proporcionalment amb el nombre de fils disponibles. Per tant, si bé és cert que amb més paral·lelisme s'obté més rendiment, com més paral·lelisme, més àrea i més consum energètic. En aquests casos cal aconseguir un balanç de rendiment enfront de consum energètic, complexitat i àrea.

Exemples

L'HyperThreading d'Intel implementa només contextos i dos fils. Un altre exemple és l'Alpha 21464, que disposa de fins a 4 fils paral·lels.

La resta de l'apartat tracta del disseny d'aquest tipus d'arquitectures, com també d'algunes de les arquitectures SMT més rellevants de la literatura.

4.3. Convertint el paral·lelisme a nivell de fil a paral·lelisme a nivell d'instrucció

Com ja s'ha comentat, les arquitectures SMT permeten que diferents fils d'execució comparteixin diferents unitats funcionals. Per a permetre aquest tipus de compartició, s'ha de desar de manera independent l'estat de cadascun dels fils.

Per exemple, cal tenir per duplicat els registres que els fils fan servir, el seu comptador de programa i també una taula de pàgines separada per fil. Si no es tenen duplicades aquestes estructures, l'execució funcional de cadascun dels fils interferiria amb la d'un altre fil. D'altra banda, hi podria haver problemes de seguretat greus. Per exemple, compartir la taula de pàgines implicaria que un fil compartiria el mapatge d'adreces virtuals a física. Això implicaria que un fil podria accedir a la memòria de l'altre fil sense cap tipus de restricció. No obstant això, hi ha altres recursos que no cal replicar, com per exemple, l'accés a les unitats funcionals per tal d'accedir a memòria (ja que els mecanismes de memòria virtual ja donen suport per multiprogramació).

La quantitat d'instruccions que un processador SMT pot generar per cicle està limitada pels desbalancejos en els recursos necessaris per a executar els fils i la disponibilitat d'aquests recursos. No obstant això, també hi ha altres factors que limiten aquesta quantitat, com el nombre de fils que hi ha actius, possibles limitacions en la mida de les cues disponibles, la capacitat de generar prou instruccions dels fils disponibles o limitacions del tipus d'instruccions que es poden generar des de cada fil i per a tots els fils.

Les tècniques SMT assumeixen que el processador facilita un conjunt de mecanismes que permeten explotar el paral·lelisme a nivell de fil. En particular, aquestes arquitectures tenen un conjunt gran de registres virtuals que poden ser usats per a desar els registres de cadascun dels fils de manera independent (assumint, evidentment, diferents taules de *renaming* per a cada fil).

Renaming

El *renaming* de registres facilita identificadors únics de registre. Sense aquest tipus de *renaming* dos fils podrien tenir interferències entre les seves execucions.

Exemple de flux d'instruccions multifil

El flux d'execució dels dos fils que es presenten a continuació no funcionaria correctament. Si els registres r2, r3 i r4 no fossin reanomenats per cadascun dels fils, l'execució funcional de les diferents instruccions seria errònia. En els instants 1 i 4 els valors que tots dos fils llegirien serien incorrectes. En el primer cas el fil 2 estaria emprant el valor del registre r3 modificat pel fil 1 en el cicle anterior. De manera similar també succeiria en el cicle 3, en què el fil 1 estaria sumant un valor r3 modificat per un altre fil.

```
cicle(n)fil 1-->"LOAD  #43,r3"
cicle(n+1)fil 2-->"ADD   r2,r3,r4"
cicle(n+2)fil 1-->"ADD   r4,r3,r2"
cicle(n+3)fil 1-->"LOAD  (r2),r4"
```

Gràcies al *renaming* de registres, les instruccions de diferents fils poden ser barrejades durant les diferents etapes del processador sense confondre fonts i destins entre els diferents fils disponibles. És important fer notar que aquest tipus de tècnica és la que s'empraria en els processadors fora d'ordre SISD. En aquest darrer cas el processador tindria una sola taula de *renaming*.

Així doncs, el procés de *renaming* de registres és exactament el mateix procés que fa un processador fora d'ordre. Per això un SMT es pot considerar com una extensió d'aquest tipus de processadors (afegint, és clar, tota la lògica necessària per a suportar els contextos dels diferents fils). Com es pot deduir, es pot construir una arquitectura fora d'ordre i SMT a la vegada.

El procés de finalització d'una instrucció[11] no és tan senzill com en un processador no SMT (en què només té en compte un fil). En aquest cas es vol que la finalització d'una instrucció sigui independent per a cada fil. D'aquesta manera cadascun pot avançar independentment dels altres. Això es pot dur a terme amb les estructures que permeten aquest procés per a cadascun dels fils, per exemple, tenint un *reoderbuffer* per fil.

(11)En anglès, *commit*.

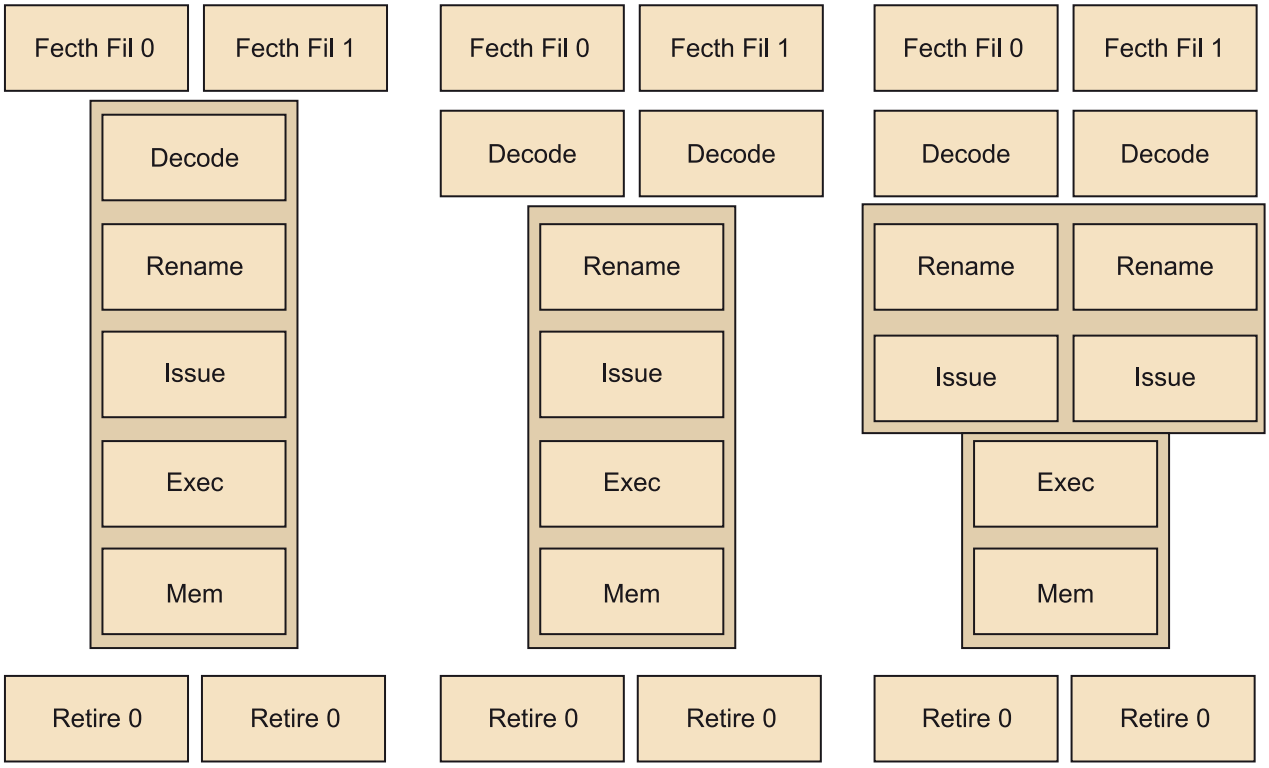
4.4. Disseny d'un SMT

En termes generals, els SMT segueixen la mateixa arquitectura que els processadors superescalars. Això inclou tant els dissenys generals de les etapes que els componen (etapa de cerca d'instrucció, etapa de descodificació i lectura de registres, etc.), com les tècniques o els algoritmes emprats (per exemple, "Tomasulo").

No obstant això, moltes de les estructures han de ser replicades per a donar suport als diferents contextos que el processador ha de gestionar. Algunes són les mínimes necessàries per tal d'evitar interferències entre fils i assegurar la correcció de les aplicacions, com tenir comptadors de programes separats o taules de pàgines separades. De totes maneres, es poden trobar variants arquitectòniques que no són estrictament necessàries, però que poden donar més rendiment en certes situacions.

La figura següent mostra algunes de les opcions que es poden tenir en compte quan es considera l'arquitectura global d'un processador SMT. Aquesta figura mostra diferents possibilitats de la manera com els diferents fils comparteixen o no les diferents etapes del processador (s'han assumit les etapes típiques d'un processador superescalar fora d'ordre). Per exemple, la primera de totes assumeix que només la cerca d'instrucció està dividida per fil, la resta d'etapes són compartides.

Figura 11. Possible disseny d'una arquitectura SMT



Com més a la dreta ens movem en la figura, les etapes es troben més dividides per fil. Cal remarcar que el fet que una etapa es trobi separada per fils exemplifica que el processador realitza l'etapa dividida per fils i que cada fil té estructures separades per a dur-la a terme. No obstant això, és lògic pensar que tots els fils tenen una lògica compartida, ja que el mecanisme és comú entre tots.

En tots els casos, les etapes d'execució i accés a memòria es troben compartides per tots els fils. En un estudi acadèmic es podria considerar que aquestes es troben replicades per fil, però ara com ara, en un entorn real, això és molt costós en termes d'espai i de consum energètic. D'altra banda, com és lògic, la utilització d'aquests recursos serà molt més elevada en els casos en què tots els fils els comparteixin. Així doncs, quan uns estiguin bloquejats els altres l'usaran, i viceversa, o bé quan uns estiguin fent accessos a memòria els altres poden usar les unitats aritmeticològiques. Per tant, per tal de maximitzar l'eficiència energètica del processador cal que es trobin compartides.

En aquests exemples l'etapa de cerca d'instrucció se separa per fil. De totes maneres, això no és comú a tots els dissenys possibles. Com es veurà més endavant, l'HyperThreading d'Intel conté una etapa de cerca d'instrucció compartida per tots els fils. En la cerca s'hi inclou la lògica de selecció de fil, com també el predictor de salts. També cal considerar que l'accés a la memòria cau d'instruccions és independent per fil. Aquesta memòria ha de tenir ports de lectura suficients per tal de satisfer la necessitat d'instruccions dels fils.

Les etapes de descodificació i *renaming* identifiquen les fonts i destins de les operacions, com també calculen les dependències entre les instruccions en vol. En aquest cas, pel fet que les instruccions dels diferents fils són independents, podrien tenir la lògica corresponent de manera separada. No obstant això, tenir les estructures de *renaming* compartides fa que el processador pugui ser més eficient en la majoria de casos.

Per exemple, si s'assumeix que es disposa d'un total de 50 registres de *renaming* per a tots dos fils i les estructures estan separades, cada fil podrà tenir accés a 25 registres com a màxim. En els casos en què un dels fils només en pugui emprar 10, però l'altre en necessiti 40, el sistema estarà infrautilitzat, de manera que el rendiment del segon fil es reduirà substancialment.

4.5. Complexitats i reptes en les arquitectures SMT

Les arquitectures SMT incrementen notòriament el rendiment del sistema augmentant la finestra d'instruccions que aquest pot gestionar. Així, en un mateix cicle, el processador pot escollir un ventall ampli d'instruccions dels diferents fils disponibles al sistema. La resta d'etapes es troben també més utilitzades pel mateix motiu. No obstant això, cal tenir en compte que aquestes millores es donen en contra del rendiment individual del fil. En aquest cas, el rendiment que un sol fil pot aconseguir pot ser menor del que hauria obtingut en un processador superescalar fora d'ordre sense SMT.

Per tal d'evitar aquest detriment individual en els fils, alguns d'aquests processadors introdueixen el concepte de *fil preferit*. La unitat encarregada de generar o començar instruccions donarà preferència als fils preferits[12]. *A priori* pot semblar que aquest aspecte pot afavorir el fet que alguns dels fils tinguin un rendiment més alt i que no se sacrifiqui el rendiment global del sistema.

(12) En anglès, *preferred threads*.

Ara bé, això no és del tot cert, perquè donant preferències a un subconjunt de fils es provoca una disminució en l'ILP del flux d'instruccions que circulen pel *pipeline* del processador. El rendiment de les arquitectures SMT es maximitza quan hi ha prou fils independents que permetin esmorteir els bloquejos que cadascun experimenta quan s'executa.

Cal comentar que també hi ha algunes arquitectures en què només es consideren els fils preferits, sempre que no es bloquegin. Si un no pot seguir endavant, el processador considera els altres fils. En aquests casos el que s'està fent és causar un desbalanceig de rendiment als diferents fils que el processador executa. Aquest factor s'ha de tenir en compte a l'hora de planificar l'execució dels diferents fils que s'executen sobre el sistema operatiu.

A banda del repte que les arquitectures SMT mostren vers la millora del rendiment individual dels fils, hi ha una varietat d'altres reptes que cal afrontar en el disseny d'un processador d'aquestes característiques, com són els següents:

- Mantenir una lògica simple en les etapes que són fonamentals i que cal executar en un sol cicle. Per exemple, en la tria d'instrucció simple cal tenir present que, com més fils hi ha, la bossa d'instruccions que es poden escollir és més gran. Passa el mateix en l'etapa de finalització d'instrucció en què el processador ha d'escollir quines de les instruccions acabades finalitzaran en el proper cicle.

- Tenir diferents fils d'execució implica que cal tenir un banc de registres prou gran per tal de desar cadascun dels contextos. Això té implicacions tant d'espai com de consum energètic.

- Un dels problemes de tenir diferents fils d'execució compartint els recursos d'un mateix processador pot ser l'accés compartit a la memòria cau. Pot passar que els mateixos fils facin el que s'anomena *falsa compartició*, que és quan fils diferents estan compartint els mateixos sets de la memòria cau, tot i que estan accedint a adreces físiques diferents. En els casos amb molta falsa compartició, el rendiment del sistema es degrada de manera substancial.

Els subapartats següents presenten dues tecnologies comercials que van incorporar el concepte de multifil compartit en els seus dissenys: l'HyperThreading d'Intel i el processador 21464 d'Alpha.

Exemples de processadors

El Memory Logix MLX1 (Song, 2002), el Clearwater Netwroks CNP810SP (Melvin, 2000) o el Flow Strom Porthos (Melvin, 2003) van ser altres exemples de processadors.

4.6. Arquitectures multinucli

4.6.1. Limitacions de l'SMT i arquitectures *superthreading*

En tots els casos anteriors, l'explotació del paral·lelisme es porta a terme afegint el concepte de fil a l'arquitectura del processador. En aquest cas, l'augment de paral·lelisme s'aconsegueix incrementant la lògica del processador, analitzant el diferent nombre de fils que es volen considerar. Per exemple, si es volen tenir vuit fils, s'han de replicar vuit vegades les estructures necessàries (més o menys vegades segons el disseny SMT que s'està considerant). Aquest model, tot i ser adequat i emprat en l'actualitat per molts processadors, té certes limitacions. A continuació se'n discuteixen les més representatives.

Escalabilitat i complexitat

Els models SMT són adients per a un nombre relativament petit de fils (2, 4 o 8 fils). No obstant això, per a un nombre més gran de fils pot portar certs problemes d'escalabilitat. Com ja hem vist, per a cadascun dels fils de què disposa un processador hi ha molta lògica que es troba replicada o compartida.

Aquest fet podria implicar només un problema d'espai. És a dir, duplicar el nombre de fils implica duplicar el nombre d'entrades de l'*store buffer*, o duplicar el nombre de taules de *renaming*. Tot i així, l'increment en la quantitat i mida del nombre de recursos també té associat un increment exponencial en la lògica de gestió d'aquests recursos.

A tall d'exemple, s'estudia el cas del *reorder buffer*. Aquesta estructura és l'encarregada de finalitzar totes les instruccions que ja han passat per totes les etapes del processador i que resten pendents de ser finalitzades (etapa de *commit*). En el cas de tenir dos fils d'execució i assumint que es genera una instrucció per cicle per fil, cal poder finalitzar o "commitejar" dues instruccions per cicle. Altrament, el sistema no és sostenible. Poder finalitzar dues instruccions per cicle és realista.

No obstant això, si s'incrementa el nombre de fils de manera lineal, no és realista esperar que es pugui construir un sistema real que sigui capaç de finalitzar el nombre proporcional d'instruccions necessàries per tal de mantenir el rendiment.

La lògica i els recursos necessaris per a gestionar la finalització d'un nombre tan elevat d'instruccions en vol serien extremadament costosos i probablement no assolibles (si es tinguessin 128 fils disponibles i 32 instruccions en vol per fil en caldrien 4.096). Si més no, considerant l'estat actual de la tecnologia de processadors.

Consum energètic i àrea

Per tal de donar suport a un nombre elevat de fils, cal incrementar les estructures proporcionalment. En alguns casos, aquests increments no són costosos en termes de complexitat i àrea, però algunes de les estructures són altament costoses d'escalar. Altra vegada, podem posar com a exemple l'accés a les memòries cau.

L'increment en el nombre de ports de lectura o escriptura de les diferents memòries cau és altament costós (tant pel que fa a complexitat com a l'àrea). Afegir un port de lectura nou en una memòria cau pot equivaldre a un increment d'un 50 % d'àrea (Handy, 1998).

Com ja s'ha dit, si es vol incrementar el rendiment de manera més o menys proporcional al nombre de fils, també cal tenir en compte com s'accedeix a la memòria cau. Per tant, si es volgués augmentar el nombre de fils, per exemple a 256, caldria redimensionar (tant en àrea com en lògica) tota la jerarquia de memòria coherentment. Com ja es pot veure, i amb la tecnologia actual, això és impracticable.

El consum energètic d'un processador SMT donant suport a un nombre molt elevat de fils és alt. En les situacions en què no es fessin servir tots els fils disponibles o l'ús dels recursos corresponents fos ineficient, el consum en watts del processador seria molt elevat comparat amb el rendiment que se n'estaria obtenint.

Com s'analitza a continuació, en altres arquitectures i en aquestes situacions, es pot aplicar *dynamic voltage scaling* (Yao, Demers i Shenker, 1995), és a dir, reduir la freqüència i voltatge d'algunes parts del processador, ja que això permet reduir substancialment el consum del processador en situacions com la plantejada.

4.6.2. Producció

Durant les darreres dècades, donada una gama de processadors que segueixen un disseny arquitectònic similar (per exemple, els SandyBridge d'Intel), es treuen diferents versions d'un mateix processador. En una mateixa família es poden trobar versions orientades als clients (ordinadors d'ús domèstic), versions orientades a dispositius mòbils i versions orientades a servidors.

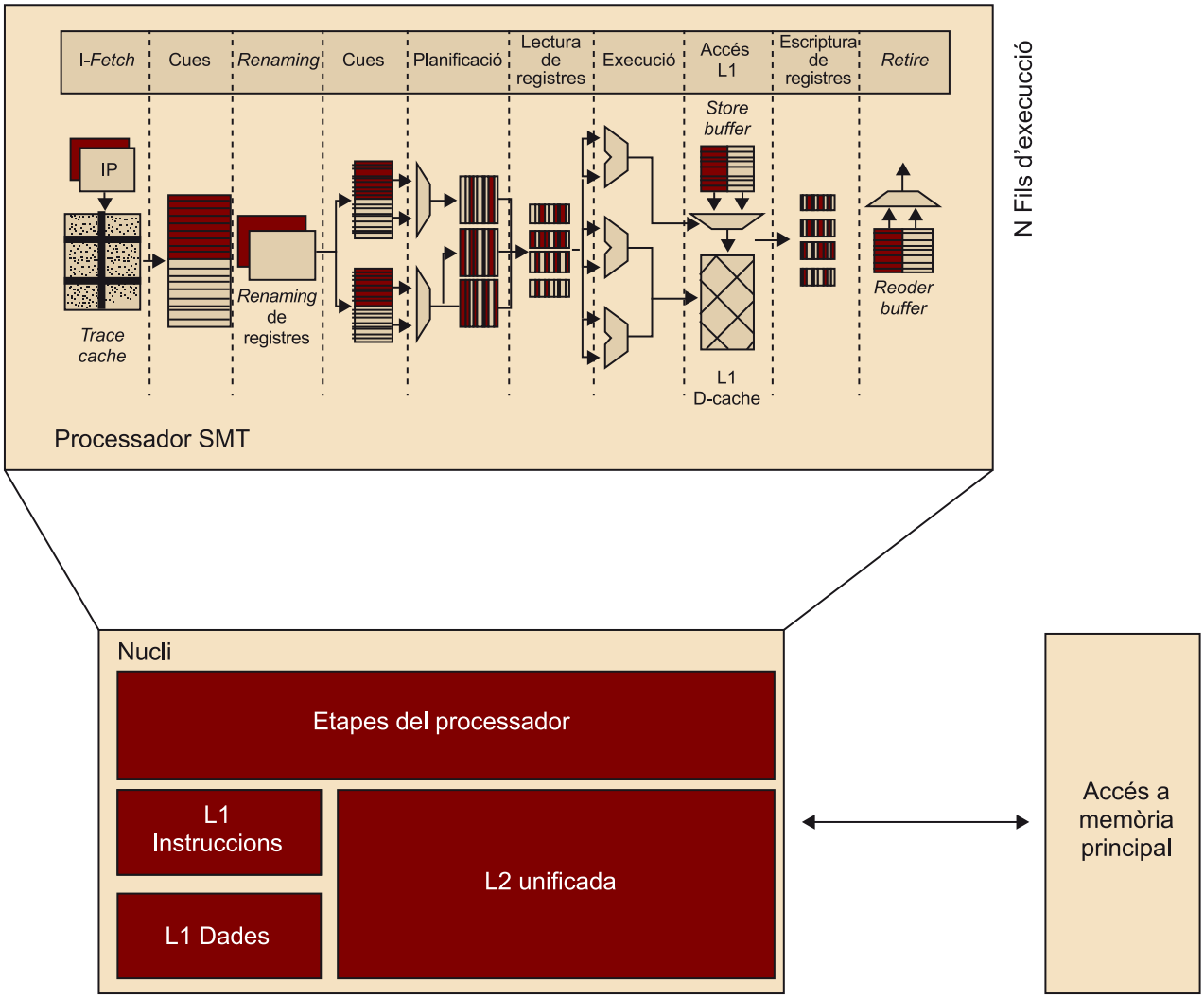
Per exemple, en el cas de la família SandyBridge, es poden trobar els processadors domèstics amb 12 fils i 130 watts de consum, processadors per a dispositius mòbils de 8 fils i 55 watts de consum i processadors per a servidors amb 16 fils i 150 watts de consum. Val a dir que no només varia el nombre de fils entre els diferents tipus de processadors, sinó que també ho fan les mides de memòries cau i prestacions específiques (per exemple, la capacitat de connectivitat amb altres processadors). Dins d'un mateix tipus de processadors (per exemple, els clients) hi ha moltes variants (per exemple, dins de la família SandyBridge de tipus client hi ha més de trenta variants).

Mirant de donar suport a tota aquesta varietat de nombre de fils limitant-se en escalar la quantitat de fils que l'arquitectura SMT suporta, seria extremadament costós des del punt de vista de producció. És a dir, la complexitat de tenir tants fils diferents encariria molt més el procés de disseny, producció i validació dels processadors. Com veurem a continuació, usant tecnologies multinucli aquest procés esdevé menys costós i més factible.

4.6.3. El concepte de multinucli

Durant els darrers subapartats, hem parlat de diferents estructures multifils. Cadascuna implementava un processador superescalar fora d'ordre afegint-hi el concepte de fil. Independentment del tipus d'arquitectura multifil (*superthreading* o SMT), aquests processadors es podrien veure com un element de computació, amb n fils i una jerarquia de memòries cau. Aquesta abstracció (com mostra la figura següent) es pot anomenar *nucli*. Cal remarcar que en aquest cas no inclou altres elements que un processador superescalar sí que inclouria: sistema de memòria, accés a l'entrada i sortida, etc.

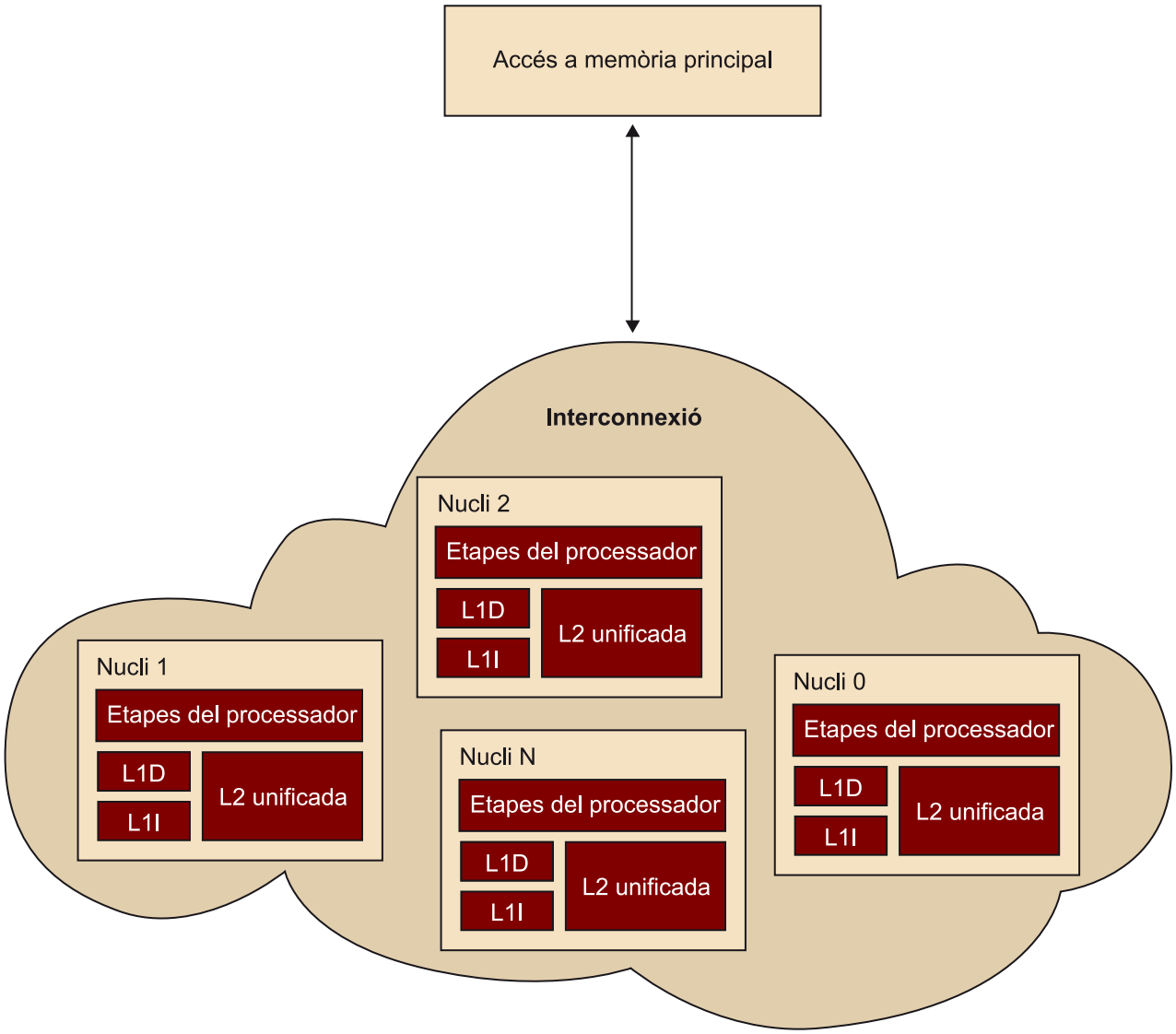
Figura 12. Abstracció de processador multifil



De fet, el concepte de *multinucli*, com indica el nom, consisteix a replicar *m* nuclis diferents dins d'un processador (figura 16). Cada nucli acostuma a tenir una memòria cau de primer nivell (de dades i instrucció), i pot tenir una memòria de segon nivell (acostuma a ser unificada: dades més instruccions). A part dels nuclis, el processador acostuma a tenir altres components especialitzats i ubicats fora d'aquests. Habitualment hi ha els següents: una memòria de tercer nivell, un controlador de memòria, components per a fer processament de gràfics, etc.

Tots aquests components (inclosos els nuclis) estan connectats per una xarxa d'interconnexió, que és el mitjà físic i lògic que permet enviar peticions d'un component a un altre (per exemple, una petició de lectura d'un nucli a la L3).

Figura 13. Abstracció de multinucli



Com ja s'ha dit, un dels components d'un multinucli fonamental és el controlador de memòria. Aquest gestiona les peticions d'accés al subsistema de memòria que fa la resta de components (tant lectures com escriptures).

Per si mateixa, una arquitectura multifil pot semblar senzilla. No obstant això, darrere d'aquest tipus d'arquitectures hi ha molta complexitat amagada que no es veu directament: protocols de coherència, escalabilitat en la interconnexió, sincronització, desbalancejos entre fils, etc.

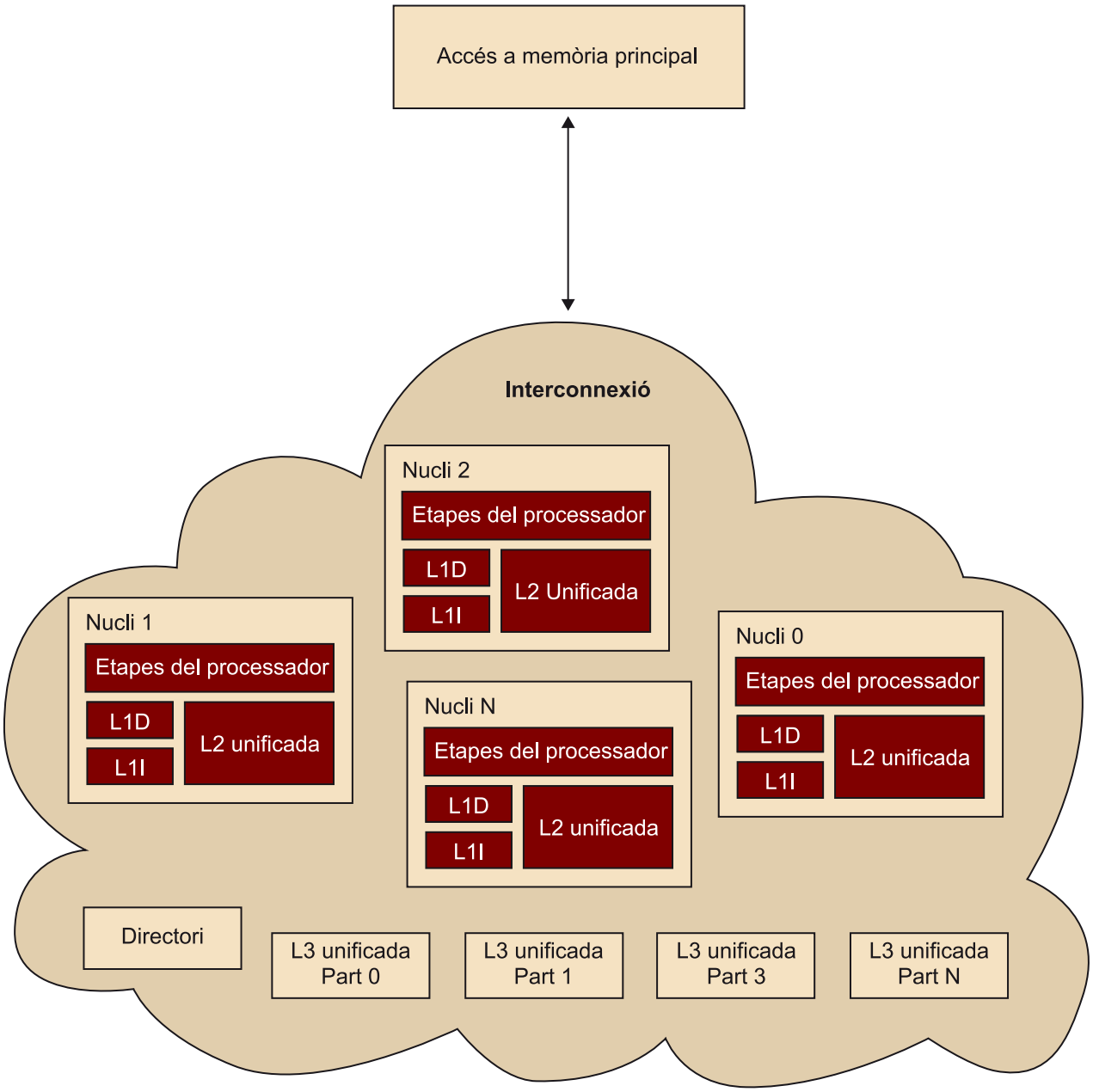
4.6.4. Coherència entre nuclis

El model que acabem de veure descriu l'estructura més senzilla d'un multinucli: nuclis, interconnexió i accés a memòria principal. No obstant això, com ja es pot deduir, hi ha moltes variants d'aquest model.

La figura 17 mostra una arquitectura sovint emprada en el món de la recerca acadèmica i també present en models comercials. De la mateixa manera que el model anterior, s'hi troben un conjunt de nuclis que proporcionen accés a recursos computacionals (amb un fil o més) i a dues memòries cau (una de primer i una de segon nivell). A part d'aquests components, n'hi ha dos de nous: una memòria de darrer nivell (o memòria cau de tercer nivell) i un directori.

A diferència de la L1 o L2, la L3 està dividida en blocs de la mateixa mida i ubicada en components separats. Com veurem a continuació, el directori és l'encarregat de saber quines línies té cada bloc de la L3 i en quin estat es troben. D'altra banda, també s'encarrega de coordinar i de gestionar totes les peticions que els diferents nuclis fan a aquest darrer nivell de memòria cau.

Figura 14. Arquitectura multifil amb L3 i directori



En aquest model, quan una petició de lectura o escriptura no encerta cap línia ni de la L1 ni de la L2, la petició es reenvia a la memòria de tercer nivell. Conceptualment això és equivalent al flux de fallada de la L1 que demana la mateixa petició a la L2. No obstant això, com ja s'ha comentat, aquesta petició es reenvia cap al directori.

El directori, donada l'adreça que s'està demanant, té informació per tal de saber quin dels components de la L3 té aquesta línia i en quin estat es troba. En cas que cap component la tingui, s'encarrega de demanar-la a memòria, desar-la en el bloc corresponent de L3 i enviar-la al nucli. Això es fa per mitjà de protocols concrets que asseguren l'ordre i finalització en el tractament d'aquestes peticions.

En el cas anterior, s'ha assumit l'escenari en què ni la L1 ni la L2 del nucli contenen l'adreça que el fil del mateix nucli està demanant. També s'ha assumit que el nucli genera una petició de lectura al directori i que aquest gestiona la petició a la L3. Tot i així, què passaria si algun altre nucli tingués la mateixa adreça a la seva L2/L1?

En aquest cas, podria passar el que mostra l'exemple següent. El primer nucli llegeix l'adreça @X, la modifica i l'escriu de tornada a la memòria cau de darrer nivell amb el valor nou. Si durant tot el procés d'aquesta transacció un altre nucli llegeix el valor de @X de la L3, rep un valor incorrecte. En el cas de l'exemple, el nucli 2 rebria un valor erroni.

Exemple de flux de lectures i escriptures incoherent

```
instant(n)     nucli 1-->lectura          @X =11     (L3-->L1/L2)
instant(n+1)   nucli 1-->escriptura       @X=0       (L2)
instant(n+2)   nucli 2-->lectura          @X=11      (L3-->L1/L2)
instant(n+3)   nucli 1-->escriptura       @X=0       (L2-->L3)
```

Per a evitar aquests problemes es fan servir protocols de coherència, que estan dissenyats per a evitar situacions com la presentada i mantenir la coherència entre tots els nuclis del processador.

Cada processador implementa models i mecanismes de coherència específics. Per tant, quan es desenvolupen aplicacions per processadors multinucli cal conèixer-ne bé les característiques. El rendiment de l'aplicació depèn en gran mesura de la manera com s'adapta a les característiques del processador.

Interconnexions

Dins de l'àmbit de computació d'altes prestacions[13] es poden trobar molts articles acadèmics que han tractat aquest problema.

(13)En anglès, *high performance computing* (HPC).

Activitat

Busqueu treballs relacionats amb topologies usades per a connectar sistemes de supercomputació.

En aquest àmbit, la problemàtica de connectar diferents components també apareix, però a una escala més gran. És a dir, en comptes de connectar nuclis, es connecten processadors entre si, o clústers. L'entorn HPC té molt més rodatge en la recerca d'aquest tipus de problemes, ja que és una ciència que treballa en aquests problemes des de ben entrats els anys vuitanta, quan es van dissenyar els primers computadors HPC.

De xarxes d'interconnexió, se n'han proposat de molts tipus (Agrawal, Feng i Wu, 1978): des de simples busos, fins a estructures tridimensionals molt complexes com les topologies Torus o el *crossbar*. No obstant això, per temes de complexitat o cost, no totes són aplicables al món dels multinuclis, en què l'escala i les restriccions de consum i àrea són més grans. Val a dir que, com més avança la tecnologia, més complexes són les xarxes i més a prop d'aquests models complexos es poden apropar els multinuclis.

Xarxes en els multinuclis

Alguns exemples de xarxes que es poden trobar en el món dels multinuclis són els busos (Ceder i Wilson, 1986), multibusos (Reed i Grunwald, 1987), anells de comunicació (Hong i Payne, 1989) o malla (Bell i altres, 2008).

4.6.5. Protocols de coherència

Els protocols de coherència han de garantir que en tot moment la visió global de l'espai de memòria és coherent entre tots els nuclis. És important saber quin tipus de protocol implementa una arquitectura quan se'n vol desenvolupar una aplicació. Donada una línia de memòria @X:

- Si un dels nuclis en vol fer una lectura, en rep la darrera còpia. No pot donar-se el cas que un altre nucli l'estigui modificant mentre aquest en té una còpia.

- Quan un nucli demana una línia de memòria, la pot demanar en exclusiva o en estat compartit. El nucli només pot modificar la línia quan aquesta la tingui en estat exclusiu. Llavors la línia es troba en estat modificat.

- Depenent del model de coherència que es faci servir, dos nuclis poden tenir @X en les memòries cau respectives en estat compartit. Ara bé, no poden modificar aquesta línia (perquè llavors tindríem dos valors diferents de @X en dos nuclis).

- Si un dels nuclis vol modificar la dada, abans cal avisar la resta de nuclis que han d'invalidar aquesta línia. Si un altre nucli la torna a voler, primer s'ha d'escriure a la L3 o a memòria i aquest l'ha de llegir de L3. Depenent del protocol podem trobar certes variants.

Alguns dels punts anteriors depenen del tipus de model de coherència que estiguem considerant, especialment el tercer, ja que, per exemple, si el processador no suporta l'estat de línia compartit, cada vegada que un nucli demana una línia, forçosament ha d'invalidar aquesta línia de tots els nuclis que la tinguin.

En aquest subapartat es considera un model MESI. En aquest cas, es considera que les línies de memòria que un nucli conté poden estar en un d'aquests quatre estats: *modified*, *exclusive*, *shared* i *invalid*. No obstant això, hi ha altres tipus de models (Stenström, 1990): MESIF, MOSI, MEOSI, etc.

Activitat

Us recomanen mirar les diferències entre els diferents tipus de models de coherència esmentats.

Els models de coherència es poden implementar sobre diferents tipus de protocols de coherència. En termes generals, es poden diferenciar en dues categories diferents:

1) Els protocols de tipus Snoop. En aquests protocols (Katz, Eggers, Wood, Perkins i Sheldon, 1985) quan un nucli vol fer alguna acció sobre una adreça de memòria, de lectura o d'escriptura, ha de notificar-ho de manera pertinent a la resta de nuclis per tal d'obtenir-ne l'estat desitjat.

Per exemple, si vol tenir una adreça de memòria en exclusiva per tal de modificar-la, primer el nucli ha d'invalidar totes les còpies que tinguin els altres nuclis. Quan tots els altres nuclis hagin respost notificant que han processat la petició, el nucli ja pot fer servir la línia. Abans, però, l'ha de llegir de memòria o de la darrera memòria cau (L3).

2) Els protocols basats en directori. A diferència dels protocols de tipus Snoop, aquí els nuclis no s'encarreguen de gestionar la coherència quan volen usar una adreça de memòria (Chaiken, Fields, Kurihara i Agarwal, 1990). En aquest cas, apareix un component nou que s'encarrega de gestionar-la: el directori. Quan un nucli vol una línia en un estat concret, aquest la demana al directori. El directori acostuma a tenir una llista concret dels nuclis que tenen l'adreça en qüestió i en quin estat la tenen. Per tant, quan processa una petició ja sap quins nuclis ha d'avisar i quins no.

El primer dels dos protocols és menys escalable que el segon i necessita molts més missatges per a gestionar la coherència entre nuclis. En el segon cas, el directori conté la informació de cada línia que tenen els diferents nuclis i en quin estat la tenen. Per tant, si un nucli llegeix una línia que no té cap altre nucli, el directori no envia cap missatge a la resta, sinó que directament li dóna la línia en estat exclusiu. En canvi, en el cas de protocols de tipus Snoop, el nucli envia missatges per a invalidar la línia en concret als diferents nuclis i en rep la notificació. En aquest segon cas, el nombre de missatges que circulen per la xarxa és molt més elevat que en cas del directori.

En qualsevol cas, el rendiment dels protocols basats en directori no és gratuït. Les estructures que desen l'estat i els propietaris de les diferents línies és costosa tant pel que fa a l'àrea com al consum energètic. Per tant, per a sistemes relativament petits és més eficient un sistema basat en Snoop.

Fins ara hem presentat els diferents tipus d'arquitectures de computadors més representatius dins l'àmbit de computació d'altes prestacions. Com s'ha vist, cadascun ofereix cert tipus de prestacions específiques que poden ser més adequades depenent del tipus d'aplicacions que es vulguin executar.

> Per exemple, en el cas de voler fer còmputs amb matrius en què les diferents operacions es poden aplicar sobre blocs grans (per exemple, 512 bytes), seria adequat utilitzar processadors o unitats vectorials. D'altra banda, si el tipus de problema que s'ha de resoldre es caracteritza per un subconjunt de problemes independents, és interessant emprar una arquitectura de processador que faciliti accés a un conjunt elevat de fils d'execució.

No obstant això, tot i que les arquitectures esmentades donen accés a una capacitat de computació elevada poden ser extremadament complexes de programar. Per aquest motiu, durant les darreres dècades han aparegut models de programació que faciliten l'accés a les seves funcionalitats.

> Per exemple, en el cas de les arquitectures vectorials han aparegut biblioteques com les Intel Advanced Vector Exetensions Instrinsics (també conegudes com a *AVX*). Un altre exemple molt clar són els diferents models de programació que s'han proposat per tal d'accedir a arquitectures de computadors que donen accés a més d'un fil d'execució, com són Pthreads, OpenMP, MPI, Intel TBB o Intel Cilck Plus.

Tot i que aquests models de programació són extremadament potents, cal considerar certes restriccions o situacions que poden disminuir-ne substancialment el rendiment. Aquest tipus de situacions apareixen més habitualment en arquitectures multifil. El fet de tenir diferents fils d'execució accedint de manera simultània i en alguns casos compartida a recursos de computació fa que en algunes situacions derivin en una lluita que perjudica de manera important el rendiment global de les aplicacions. Els propers dos apartats presenten alguns dels factors més importants que cal considerar a l'hora de dissenyar i implementar una aplicació multifil.

5. Arquitectures *many-core*: el cas de l'Intel Xeon Phi

En aquesta secció ens centrarem en arquitectures *many-core* utilitzant una de les famílies de processadors d'Intel, Many Integrated Core d'Intel, com a fil conductor. Com es veurà durant les properes subseccions, aquests processadors es caracteritzen per donar accés a un nombre molt elevat de fils d'execució. D'aquesta manera aplicacions que són altament paral·leles poden treure un rendiment força més elevat respecte d'altres arquitectures disponibles en el mercat. Per altra banda, un aspecte que les fa molt atractives respecte de l'ús de GPU (que també donen accés a un nombre molt elevat de fils d'execució) és que manté una visió coherent de l'espai de memòria, mentre que la majoria de GPU actuals no ho fan.

Primerament es presentarà l'evolució històrica (fins al moment d'escriure aquest material) de la família Xeon Phi i els seus orígens. A continuació es presentaran les característiques més important dels dos processadors que s'han presentat fins ara (Knights Ferry i Knights Corner). Finalment es donaran un conjunt de referències recomanades per a aprofundir en les arquitectures Xeon Phi.

5.1. Història dels Xeon Phi

L'any 2010, la companyia de processador Intel va anunciar el primer dels processador Intel que inclouria un gran nombre de nuclis integrats. Aquest nombre de nuclis seria molt més elevat que el nombre dels processadors dissenyats fins aleshores. Aquest concepte, ja conegut en la literatura de computació d'altes prestacions, és conegut com a *many integrated cores* (molts nuclis integrats).

Aquesta família de processadors, anomenada Intel Xeon Phi, heretava el disseny conceptual del projecte Larrabee. L'objectiu d'aquest era dissenyar una GPU (*graphical processing unit* o targeta gràfica) en què els còmputs es fessin en unitats de procés de propòsit general (x86). El projecte en qüestió no va sortir a la llum. No obstant això, tota la recerca feta es va emprar per a transformar aquest sistema multinucli en un processador de propòsit general que donés accés a un nombre molt elevat de fils d'execució. Per a més detalls sobre el projecte Larrabee es recomana la lectura de l'article de Selier, Carmean i Sprangle (2008). Un dels altres avantatges més importants d'aquestes arquitectures era el baix consum energètic respecte dels competidors d'altres companyies.

Després del 2010, Intel va anunciar tres nous processadors dins de la família dels Xeon Phi: Knights Ferry (KNF), Knights Corner (KNC) i Knights Landing (KNL).

El primer de tots, KNF, va ser un prototip que tan sols es va lliurar a centres de recerca o centres de supercomputació per a començar a fer proves i avaluacions de rendiments amb aquesta nova família de processadors. D'aquesta no se'n va fer cap venda comercial. L'objectiu era que els potencials usuaris finals d'aquest comencessin a veure com calia adaptar les aplicacions (si era necessari) i fer projeccions de quin rendiment en traurien. Com a característiques més importants cal destacar que estava compost per trenta-dos nuclis amb quatre fils per nucli, estava construït sobre un procés de 45 nm i venia amb 2 GB de memòria integrada. Aquest era el primer dels processadors amb objectius comercials que Intel llançava amb 128 fils d'execució.

El segon, KNC, va ser la versió ja comercial de KNF i es va presentar l'any 2012. Aquesta arquitectura es fonamentava en el mateix disseny que KNF. No obstant això, donava accés a un nombre més elevat de nuclis (cinquanta nuclis en la primera versió), incorporava una memòria integrada molt més gran (fins a 16 GBS) i usava un procés de producció més eficient (22 nm). Usant aquesta nova arquitectura, centres de computació com el Texas Advanced Computing Center o el Centre Guangzhou van construir supercomputadors que es van col·locar en les primeres posicions del Top 500 (Top500) i del Green Top 500 (500). El primer va construir Stampede (TACC), sistema format per un total de 102,400 nuclis i capaç de 9.5 petaflopts. El segon va construir el computador anomenat Thiane-2 (Top500, China's Tianhe-2 Supercomputer Takes No. 1 Ranking on 41st TOP500). Aquest és capaç d'arribar a un rendiment màxim de 33.8 petaflops, i en el moment del llançament va ocupar la primera posició dels computadors més potents del món (Top 500).

Finalment, el darrer de tots, KNL es va anunciar al final del 2013. Cal remarcar que KNC havia estat una evolució de KNF.

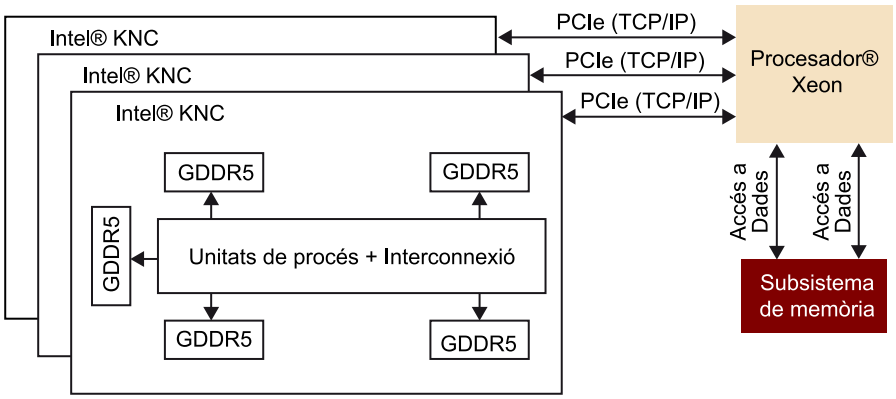
El disseny intern de KNL és força diferent del que es veurà en la propera subsecció. Com a característica més remarcable, aquest processador du integrat un sistema de memòria anomenat *high memory bandwidth*. Aquest, a part de donar accés a una quantitat de memòria més gran que els seus predecessors, permet accedir-hi mitjançant una amplada de banda molt més gran. Això és especialment important tenint en compte que moltes de les aplicacions HPC es troben limitades, per una banda, per la quantitat de memòria que poden fer servir, però, per una altra banda (i en alguns casos més important), per la quantitat de dades per segon a la qual el processador pot seguir.

Com a exercici de lectura i recerca, es recomana que un cop acabada la lectura d'aquesta secció se cerquin en la Xarxa les novetats d'aquesta família de processadors: nous models, prestacions, etc. Cal tenir present que estem parlant d'un mercat molt recent i en evolució constant. Per tant, el que es cobreix pot no incloure novetats més recents o canvis importants incorporats en noves generacions.

5.2. Presentació dels Xeon KNC i KNF

Com ja s'ha comentat en la darrera subsecció, l'arquitectura d'un KNC és una extensió comercial de la de KNF que augmenta la quantitat de nuclis i de memòria integrada. En aquesta subsecció es cobreixen les característiques arquitectòniques comunes a totes dues arquitectures. Com ja s'ha esmentat, en el moment d'escriure d'aquesta documentació, l'arquitectura dels Xeon KNL no s'havia fet pública. Per tant, no se'n discutiran els detalls.

Figura 15. Visió global d'un KNC



La figura 15 mostra una visió global de com un sistema de computació pot ser construït emprant KNC (a partir d'aquest punt s'emprarà KNC en referència a KNC o KNF indistintament, en cas contrari s'especificarà). Com es pot observar, els KNC es troben connectats a un processador Xeon tradicional (com podria ser un Sandy Bridge Corporation, Sandy Bridge Server Products) o Haswell (Corporation, Haswell Server Products) mitjançant PCIexpress. Això és degut al fet que tant KNC com KNF es van dissenyar com a coprocessadors. Aquests no són capaços d'arrancar un sistema operatiu convencional (per exemple, Linux o Windows). Per tant, necessiten un processador complet que permeti arrancar un sistema operatiu i permeti a l'usuari interactuar amb el sistema (executar aplicacions, gestionar fitxers, etc.).

De la mateixa manera que es fa amb els ordenadors convencionals i les targetes gràfiques (GPU), els KNC es col·loquen als PCIexpress del computador. Quan s'executa una aplicació, aquesta s'instancia en el processador principal (anomenat també *host*). El fil principal de l'aplicació s'executa en aquest. No obstant això, usant un conjunt de biblioteques que Intel facilita (*Corporation, Intel® Xeon Phi™ Coprocessor Developer's Quick Start Guide*, 2013), l'aplicació és capaç de moure dades al KNC i d'instanciar-hi fils d'execució.

```
float reduction(float *data, int size)
{
    float ret = 0.f;
    #pragma offload target(mic) in(data:length(size))
    for (int i=0; i<size; ++i)
    {
        ret += data[i];
    }
    return ret;
}
```

Codi 1. Exemple d'*offload*

El codi 1 mostra un exemple de com el programador pot especificar quina part del codi es vol executar en el *host* i quina s'executa en el KNC. Com es pot observar, per defecte, el codi s'executarà en el processador principal (*host*). Ara bé, si se'n vol executar una part en el coprocessador cal especificar-ho usant el que en el món de la programació es coneix com a *pragmas*. Aquests permeten especificar al compilador i a les biblioteques quin conjunt de línies es volen executar fora i com s'han d'executar. En l'exemple anterior es demana d'executar el bucle en el KNC i s'especifica que cal transferir-hi el contingut apuntat per la variable *data* que té una mida *size*.

Tal com es pot observar en la figura 15, es poden construir sistemes compostos per un sol processador *host* i tants coprocessadors KNC com entrades PCI-express tingui la placa base que s'està emprant. En cap cas es poden comunicar dos KNC connectats a la mateixa placa. Sempre s'ha de fer per mitjà del processador principal. Aquest és l'encarregat d'interaccionar amb tots els KNC que formen part del sistema.

Una de les novetats importants de KNL respecte als seus predecessors, és que aquest sí que serà *bootable*. És a dir, aquest serà capaç d'arrancar un sistema operatiu complet i executar aplicacions per si mateix. Aquesta és una propietat molt interessant, ja que es podran construir sistemes d'altes prestacions sense haver d'usar forçosament processadors de servidor convencionals. Aquests sistemes podran estar compostos únicament per processadors KNL.

Durant els darrers paràgrafs s'ha esmentat que les dades es transmeten del processador principal al coprocessador. Això té tres implicacions importants. La primera és que els KNC han de tenir memòria pròpia per a poder emmagatzemar aquestes dades. Tal com es pot observar, els KNC empren prou memòria GDDR5 (Jedec) per a emmagatzemar dades. Les diferents versions disponibles en el mercat comprenen versions des de 6 GB fins a 16 GB les més potents.

La segona implicació important és que el *host* i el KNC no comparteixen espai de memòria virtual. És a dir, les variables que s'instancien el codi executat en el *host* no seran visibles per les parts de codi executades dins el KNC i viceversa. Si es volen compartir dades serà necessari especificar dins el codi quines variables es volen moure del *host* cap al KNC i de les KNC cap al *host*.

La tercera implicació pot afectar l'eficiència de les aplicacions que es dissenyin. És a dir, cal minimitzar, optimitzar al màxim les comunicacions que es fan del *host* al KNC. Cal estudiar quina és la millor manera de moure les dades (per exemple: moure-les totes de cop, a trossos, etc.). Cal tenir present que l'amplada de banda que un KNC pot donar de memòria principal és de 352 GB/s en la versió de 16 GBS (*pcwire*, 2012), i el que pot donar el PCIe pot ser de 8 GB/s en un PCIexpress x16 (anandtech). Per tant, el subsistema de memòria d'un KNC donarà una amplada de banda quaranta vegades més ràpida que el que dóna el PCIexpress.

5.3. Arquitectura i sistema d'interconnexió

En la subsecció anterior s'han donat les característiques més importants del funcionament d'un sistema de computació construït amb processadors de la MIC. Com s'ha vist, la comunicació entre diferents nodes KNC i el processador principal es fa amb una connexió PCIexpress. Ara bé, com està dissenyat internament un KNC? Com estan interconnectats els diferents nuclis, la memòria interna d'aquest i com aquests es connecten amb el món anterior ?

La figura 16 mostra l'arquitectura interna d'un KNC. Com es pot observar, aquest conté quatre tipus diferents d'agents:

- L'agent PCIe és l'encarregat de comunicar-se amb el *host*. Cada vegada que un dels nuclis accedeix a l'espai de memòria que es troba mapat al *host*, la petició del nucli acaba arribant al *host* per aquest agent. De la mateixa manera, cada vegada que el *host* vol enviar dades o peticions al KNC, ho fa usant aquest agent.

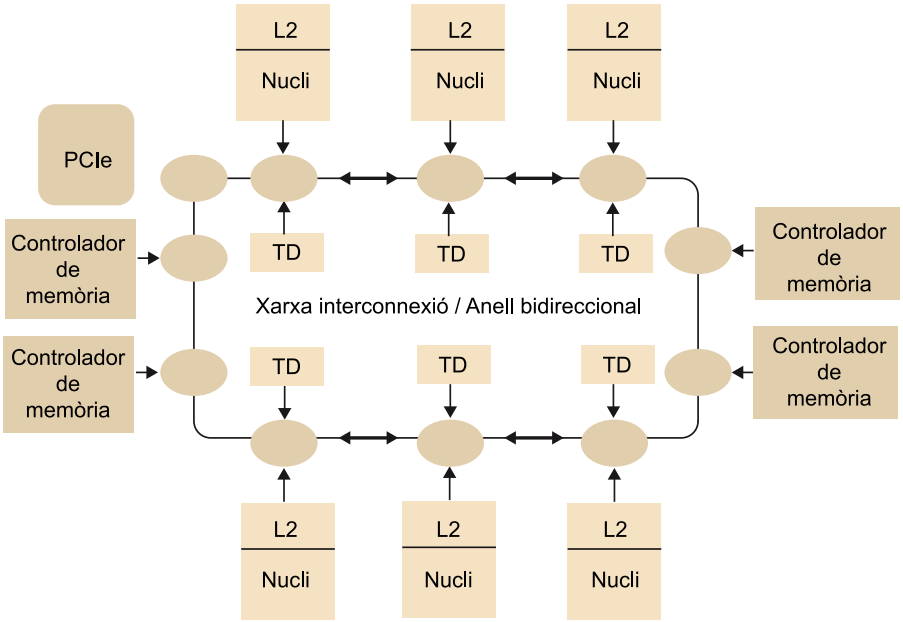
- L'agent Nucli és l'encarregat de dur a terme els càlculs i accions dels diferents fils que l'aplicació ha instanciat. Com ja s'ha esmentat anteriorment, cada nucli té quatre fils d'execució i conté dos nivells de memòria cau (més endavant se'n presenten més detalls).

- L'agent TD o *tag directory* és l'agent encarregat de mantenir la coherència de memòria. Cada vegada que un nucli vol accedir a una adreça de memòria ho ha de demanar al TD, ja que és l'encarregat de gestionar la coherència de l'adreça en qüestió. Cada adreça de l'espai de memòria és gestionada per un i tan sols un TD. Com es veurà més endavant, quan aquest rep una petició per a accedir a una dada ha de controlar que cap altre nucli no la tingui abans de demanar-la a memòria. En cas contrari haurà de notificar-ho al nucli que la tingui perquè aquest actualitzi el seu estat.

- Finalment, el darrer agent és el controlador de memòria. Aquest agent és l'encarregat de satisfer les peticions d'accés a la memòria principal que rep del TD i que han estat originades per un dels nuclis. Quan aquest rep una petició de lectura o escriptura, el controlador tradueix aquesta petició a les tecles d'ordre específiques que el dispositiu de GDDR5 entén. Per

a més informació de com un controlador de memòria interacciona amb un dispositiu GDDR5 es recomana la lectura de la introducció a aquesta tecnologia (Jedec).

Figura 16. Arquitectura interna d'un KNC



Els diferents agents es troben connectats per un anell bidireccional. Aquest és el responsable de realitzar la comunicació entre els diferents nuclis, TD, controladors de memòria i agent PCIexpress necessària per a poder mantenir la coherència de l'espai de memòria i comunicar-se amb el *host*. És important remarcar que tot i semblar una peça senzilla de maquinari, aquesta és una peça altament complexa que ha de tenir en compte molts aspectes importants: mecanismes de balanceig, mecanismes de descongestió en cas de molta càrrega, mecanismes per a evitar bloquejos (també anomenats *deadlocks*) etc.

Aquest anell bidireccional ha de ser capaç de transmetre informació de tres tipus diferents:

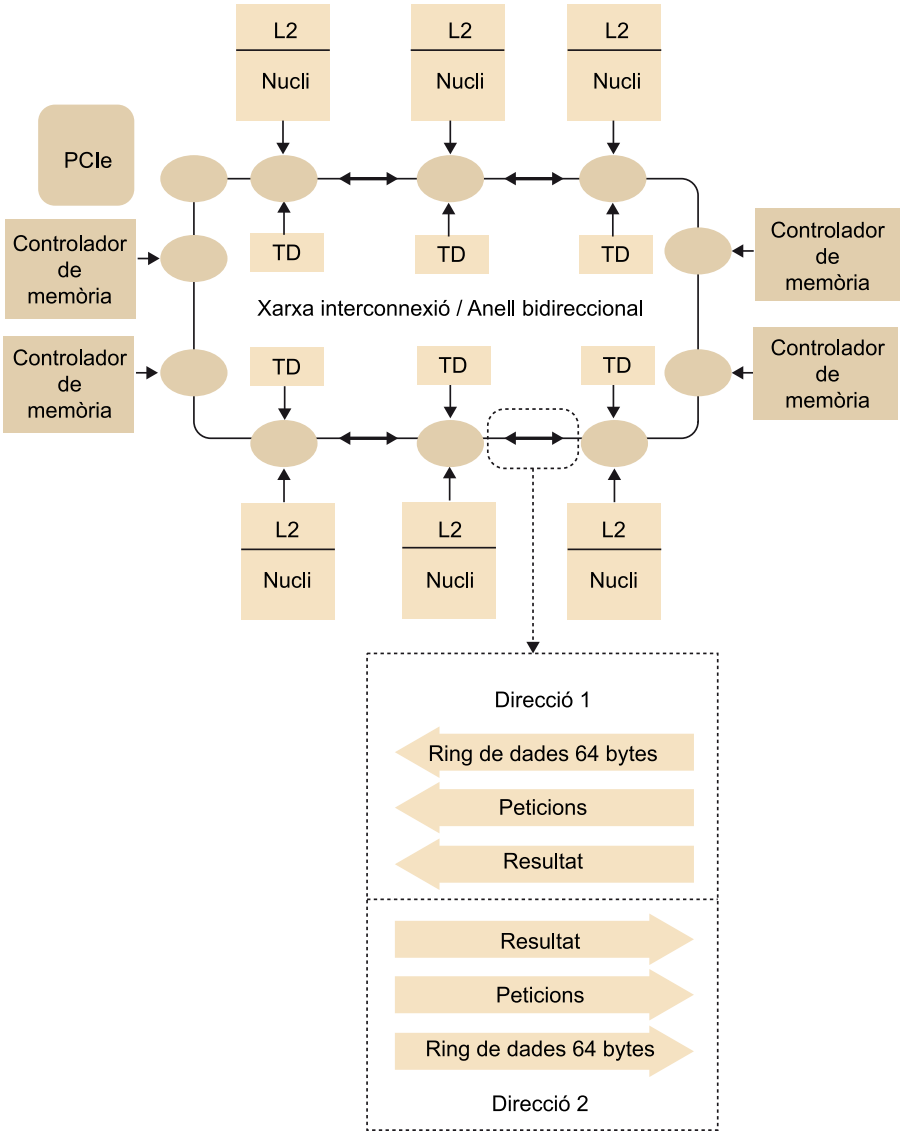
- Peticions entre els diferents agents. Per exemple la petició d'una línia de memòria en exclusiva d'un nucli a un TD; o bé una petició de lectura d'una línia de memòria d'un TD a un controlador de memòria.

- Dades que s'envien entre els diferents agents generades com a conseqüència d'una petició que s'ha iniciat anteriorment. Per exemple: després d'una de lectura d'una línia de memòria per part d'un nucli, algun dels controladors de memòria acabarà enviant les dades demanades al nucli que ha enviat la petició.

- Respostes de confirmació o resultat que, com en el cas de les dades, han estat generades com a conseqüència d'una petició. Per exemple: quan un TD rep una petició de lectura d'una línia de memòria, aquest ha de res-

pondre al nucli que la transacció s'està processant correctament i ha de dir en quin estat es retornarà la línia en qüestió (exclusiva o compartida).

Figura 17. Anell del *ring* bidireccional



Tal com es mostra en la figura 17, l'anell bidireccional està format per tres subanells diferents (tres en cada direcció). El primer de tots, anomenat AD, és el responsable de transportar les peticions. El segon de tots anomenat, BL (nom abreujat de ***bl****ock*), és el responsable de transportar les dades. I finalment el tercer, anomenat ACK (nom abreujat de ***ack****nowledgement*), és el responsable de transmetre confirmacions i resultats de les peticions.

Cadascun d'aquests subanells té una mida força diferent. El més ample de tots és el de BL. Aquest té una amplada total de 512 bits (mida de la línia de 64 bytes de memòria) més la capçalera (necessària per a incorporar informació de ruta: destinatari, origen, identificador de transacció, etc.). El següent, l'AD, és menys ample que aquest primer. Aquest només ha de transportar el tipus de petició, l'adreça física de la línia a la qual fa referència (48 bits) i, com en

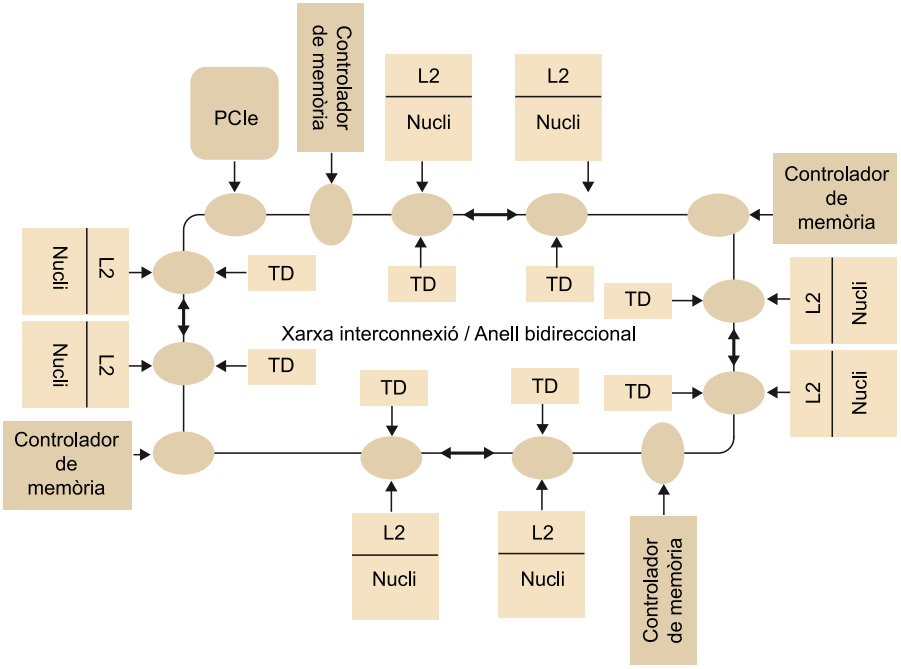
el cas anterior, la informació de ruta. Finalment, l'anell d'AK és el que menys amplada necessita. Aquest tan sols ha de dur la resposta, l'identificador de la transacció a la que fa referència i la informació de ruta.

És important remarcar que el disseny d'aquest anell bidireccional té com a objectiu donar prou amplada de banda perquè es puguin satisfer totes les transferències entre nuclis i l'amplada de banda que cadascun dels dispositius de GDDR5 dóna. És molt important tenir en compte que si cadascun dels dispositiu pot donar un total de 80 GB/s bidireccional (escriptura i escriptura) a memòria, el disseny del sistema ha de ser capaç de suportar tota l'amplada de banda que tots els controladors poden donar (en la versió més potent fins a 350 GB/s). En cas contrari, no s'estaria usant tota la potència que el sistema de memòria dóna. Això no seria acceptable tenint en compte que aquest és un dels components més cars d'un sistema.

Tal com mostra la figura 18, els KNC tenen un total de quatre controladors de memòria. Cada controlador dóna accés a una quarta part de la memòria total disponible. Per exemple, en cas de tenir la versió amb 8 GB, cada controlador donarà accés a dispositius de GDDR5 de 2 GB amb una amplada de banda de 80 GB/s. Tal com s'ha esmentat, el disseny de la xarxa d'interconnexió haurà de ser capaç de poder saturar els quatre controladors.

En aquest apartat no s'aprofundirà en els detalls d'aquest disseny. En qualsevol cas, si el lector hi està interessat, es recomana la lectura del mòdul de xarxes d'interconnexió i ampliar els coneixements mitjançant lectures de treballs relacionats disponibles en la Xarxa. Com ja s'ha esmentat, un disseny d'un sistema de comunicació d'aquestes característiques requereix tenir en compte molts factors decisius a l'hora de treure'n el rendiment esperat.

Figura 18. Arquitectura d'un KNC



5.4. Nuclis dels KNC

Tal com ja s'ha esmentat anteriorment, la primera generació de Xeon Phi incorporava un nombre més limitat de nuclis respecte de KNC (un total de trenta-dos nuclis). No obstant això, l'arquitectura de tots dos era exactament la mateixa. En aquesta secció es presentaran les característiques més generals d'aquest i les seves prestacions.

Abans de discutir les prestacions d'aquests nuclis, cal remarcar que la segona generació, els processadors KNC, ofereix un ventall més obert de configuracions. Com es pot observar en la taula 1, l'opció més simple incorpora un total de cinquanta-set nuclis que poden córrer a una freqüència d'1.1 GHz, una memòria amb una amplada de banda màxima de 240 GB/s amb una capacitat de 6 GB i un rendiment màxim (en doble precisió) de 1003 gigaflops. Per altra banda, l'opció més agressiva d'aquest model incorpora un total de seixanta-un nuclis que poden córrer a 1.2 GHz oferint un rendiment màxim de 1208 gigaflops. Aquest també inclou una memòria amb una amplada de banda màxima de 352 GB/s i capacitat de 16 GBS.

Taula 1. Opcions de KNC disponibles (finals de 2013).

Model	Consum (Wats)	Nombre nuclis	Frequencia (GHZ)	Rendiment màxim (GFLops)	Ample de banda de memòria màxim (GB/s)	Capacitat memòria
3120P	300	57	1.1	1003	240	6
3120A	300	57	1.1	1003	240	6
5110P	225	60	1.053	1011	320	8
5120D	245	60	1.053	1011	352	8

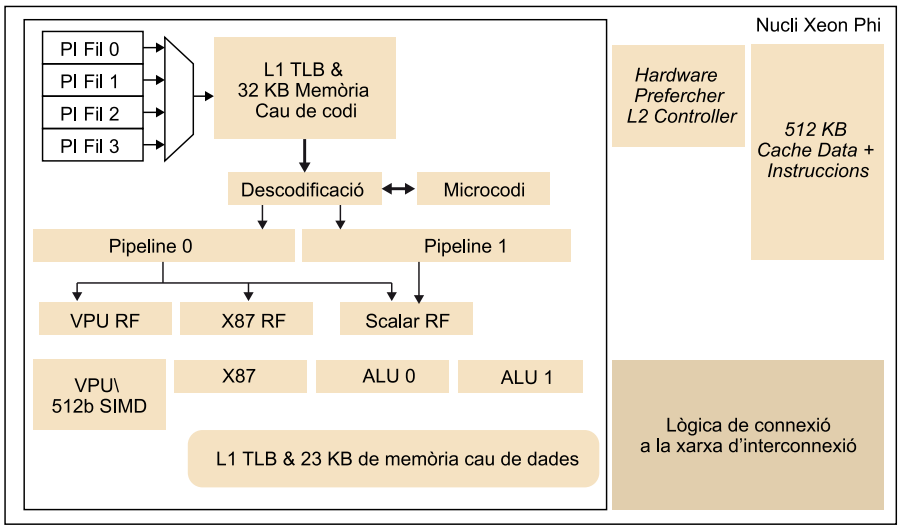
Model	Consum (Wats)	Nombre nuclis	Frequencia (GHZ)	Rendiment màxim (GFLops)	Ample de banda de memòria màxim (GB/s)	Capacitat memòria
7110P	300	61	1.238	1208	352	16
7120X	300	61	1.238	1208	352	16

La diferència fonamental entre cadascuna de les opcions que hem mostrat en el punt anterior pel que fa als nuclis és a la freqüència a què aquests s'executen i la quantitat de memòria a la qual donen accés. La seva arquitectura interna és exactament la mateixa.

La figura 19 mostra els elements fonamentals que en defineixen el disseny. Com es pot observar en la part superior, aquest conté una estructura que emmagatzema la informació necessària per a poder executar quatre fils d'execució. De manera simbòlica, només es mostra el punter d'instrucció (PI), no obstant això, aquest emmagatzema altres dades necessàries per a poder gestionar el seu flux d'execució. Com ja s'ha vist en altres mòduls, això permet que les aplicacions puguin instanciar fins a quatre *hardware threads* o fils per cadascun dels nuclis.

A continuació podem veure que cadascun dels fils que s'executa s'extreu d'una memòria cau de primer nivell (només d'instruccions) de 32 kB. Com ja és sabut, les etapes d'un processador no treballen amb adreces virtuals, sinó físiques. Per aquest motiu, també en aquesta primera part trobem la TLB (*translation lookaside buffer*). Aquesta és una estructura que permet traduir les virtuals a físiques. Un cop la instrucció s'extreu de l'L1 d'instruccions i s'ha fet la TLB, les instruccions entren en la fase de descodificació. En aquesta fase, el nucli s'encarrega de dur a terme totes les comprovacions necessàries per a poder executar la instrucció (dependències, riscos estructurals, etc.) i de traduir les instruccions (anomenades *macroinstruccions*) en instruccions més simples, anomenades *microinstruccions*. Aquesta és una característica comuna a totes les arquitectures x86 (Corporation, Introduction to x64 Assembly).

Figura 19. Arquitectura d'un nucli de KNC



Les etapes següents ja són pròpiament les etapes de càlcul i d'accés a memòria en cas de ser necessari. Una de les característiques més remarcables d'aquesta arquitectura són les unitats vectorials que aquesta incorpora, anomenades *vector processing unit* (VPU). Aquestes unitats vectorials permeten executar setze operacions de precisió simple o vuit de doble precisió per cicle i permeten executar instruccions FMA. Les FMA (*fuse-multiply and add*) permeten sumar un element més un segon element multiplicat per un tercer element (A = A + B*C). Com que estem parlant d'una unitat vectorial, tant A com B com C són vectors. Per tant, les FMA permeten fer fins a trenta-dues operacions de precisió simple o bé setze de doble precisió per cicle. Aquestes operacions, molt emprades en el món de les aplicacions d'altes prestacions, permeten obtenir un bon rendiment de càlcul.

A part de la memòria cau d'instruccions de primer nivell, cada nucli conté una memòria de dades de primer nivell de 32 kB (juntament amb la seva TLB). Com a segon nivell de memòria, conté una L2 de 512 kB. En aquest cas l'L2 és unificada. És a dir, conté tant dades com instruccions.

Un dels elements interessants que un nucli KNC té és el *hardware prefetcher*. Aquest, tal com ja s'ha introduït en altres mòduls, és l'encarregat de demanar dades de memòria abans que un fil les demani. Això es fa mitjançant algorismes que miren de predir a quines adreces de memòria accedirà el fil en qüestió en un futur. D'aquesta manera, quan aquest les demani, ja estaran emmagatzemades en la memòria cau i no caldrà anar-les a buscar a la memòria principal (la qual cosa implicaria una quantitat de cicles molt superior).

5.5. Sistema de coherència

Els KNC implementen un sistema de coherència MESI (*modified, exclusive, shared i invalid*) que implementa un sistema GOLS (*globally owned locally shared*). Com ja s'ha discutit anteriorment, el fet de tenir un sistema de coherència permet assegurar que en tot moment la visió de memòria per part de tots els

nuclis és coherent. Per tant s'assegura que dos nuclis accedeixen a la línia de memòria @X, i aquests tindran un visió coherent d'aquesta (el mateix valor). També s'assegura que, si un nucli vol modificar el seu valor, cap altre nucli en tindrà una còpia que sigui diferent d'aquesta.

Taula 2. Definició de MESI

Estat	Definició	Propiteris de la linia	Estat respecte memòria
M	Modificada	Només un nucli és propiteri la línia	Modificada
E	Exclusiva	Només un nucli és propiteri la línia	No modificada
S	Compartida	Un conjunt de nuclis contenen la línia	Pot estar modificada o no
I	Invalida	Cap nucli conté la línia	-

La taula 2 mostra els diferents estats en els quals es pot trobar una línia de memòria en un moment determinat de l'execució dins d'una L1 o L2 d'un nucli. Pel fet de ser un sistema multinucli, una línia de memòria es pot trobar ubicada en diferents memòries cau de diferents nuclis. El sistema GOLS, definit en la taula 3, estén la definició de MESI afegint quatre estats més que es fan servir globalment en el sistema. Cada nucli guarda l'estat de la línia seguint un protocol MESI, i els TD, encarregats de mantenir una visió global coherent, implementen el protocol GOLS.

L'estat GOLS permet saber al TD que tot i que els diferents nuclis tenen una línia en estat compartit o S, aquesta havia estat prèviament modificada per un nucli (quan només aquesta la tenia). Per tant, quan el darrer nucli que contingui la línia la vulgui invalidar caldrà escriure-la en la memòria. Per exemple:

1) El nucli 1 demana la línia @Y al TD 1 i la modifica. Aquest passa d'estat E al nucli M.

2) El nucli 2 demana la línia @Y al TD 1. El TD primer enviarà una notificació al nucli 1 perquè passi en estat S i aquest li respondrà que la tenia modificada. Al final d'aquesta transacció, els nuclis 1 i 2 tindran la línia en estat S (per tant no la podran modificar) i el TD 1 tindrà apuntat que es troba en mode GOLS.

3) El nucli 1 invalida la línia i ho notifica al TD 1.

4) El nucli 2 invalida la línia i ho notifica al TD 1. Aquest cop, com que el nucli 2 és el darrer a tenir la línia i el TD sap que es trobava en mode GOLS, demanarà al nucli 2 que enviï les dades a memòria.

Taula 3. Definició de GOLS

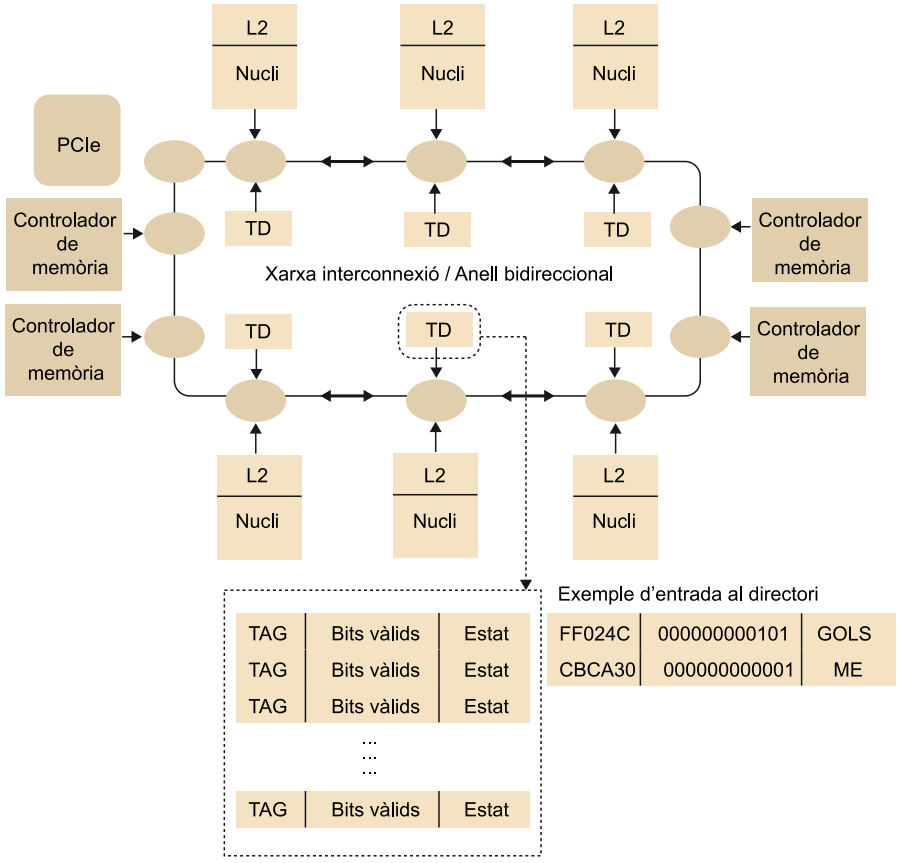
Estat	Definicio	Propiteris de la linia	Estat respecte memòria
GOLS	*Globally owned Locally Shared*	Diferents nuclis la poden tenir	Modificada
GM/GE	*Globally Modified / Exclusive*	Només aquest nucli la té	Pot estar modificada o no
GS	*Globally Shared*	Diferents nuclis la poden tenir	No ha sigut modificada
GI	*Globally Invalid*	Cap nucli conté la línia	-

Al principi d'aquesta secció s'ha explicat que els TD són els agents encarregats de gestionar la coherència de memòria. Com ja s'ha esmentat, cada línia de memòria és gestionada per un i solament un TD. Cada nucli implementa un funció que li permet calcular quin és el TD que gestiona una adreça en qüestió td_id = f_calcul_td(@X). Tots els nuclis implementen la mateixa funció, d'aquesta manera el sistema s'assegura que les peticions sobre una adreça aniran sempre al mateix TD.

Cada TD conté una estructura que li permet seguir l'estat de cada línia i saber quin dels nuclis la tenen. La figura 15 n'és un exemple. Cada línia té associats, a part de l'estat, el que s'anomenen *els bits vàlids*. Cada bit correspon a un dels nuclis del sistema. Si un nucli té la línia, el bit corresponent estarà a 1. En aquest exemple, la línia amb TAG FF024C es troba en estat GOLS i la tenen els nuclis 0 i 2. Cal remarcar que el TD no guarda la adreça de la línia sencera, sinó el TAG (com ja s'ha discutit anteriorment en mòduls que introdueixen la gestió de memòries cau).

Els sistemes de coherència són una de les peces més importants a l'hora de determinar el rendiment que un sistema pot donar. Aquest és especialment important per a aplicacions en què els diferents fils comparteixen molt sovint dades o bé han d'accedir molt al sistema de memòria. Es recomana la lectura de l'article de Ramos Garea i Hoefler (2013). Aquest presenta una comparativa del sistema de coherència d'un KNC respecte d'un Sandy Bridge. No tan sols en presenta una descripció detallada, sinó que també en presenta un estudi de rendiment força interessant.

Figura 20. Implementació del directori del TD



5.6. Protocol de coherència

El KNC implementa un protocol de missatges entre els diferents agents per mantenir l'estat MESI i GOLS. Aquest protocol s'implementa sobre els tres subanells explicats en les darreres subseccions (AD, BL i AK). Com ja s'ha esmentat, les peticions es generen i s'envien per mitjà d'AD, les respostes per mitjà d'AK, i les dades per mitjà de BL.

El protocol que permet definir l'estat de coherència és complex i preveu moltes situacions i tipus de peticions diferents. Cal tenir present que no solament hi ha peticions d'accés a memòria. Hi podem trobar transaccions que demanen accés a zones de memòries no coherents, a la zona PCIexpress etc. Cada tipus de transacció ha de tenir associats uns conjunts de missatges i dependències perquè aquesta es dugui a terme.

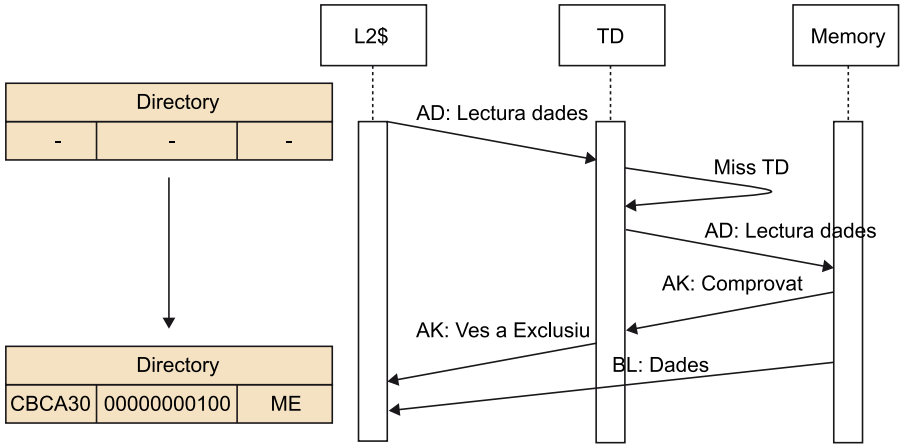
A tall d'exemple, a continuació es descriuen tres possibles transaccions en el cas d'una petició de lectura d'una línia de memòria per part d'un nucli.

Petició d'una línia de memòria no ubicada en cap nucli

El nucli demana la línia en qüestió al TD mitjançant l'anell d'AD. El TD busca en el seu directori i veu que aquesta no és dins de cap de les L2 de la resta de nuclis. A continuació, el mateix TD demana al controlador de memòria que

enviï les dades de la línia al nucli que l'ha demanat. El controlador enviarà una confirmació d'inici de transacció al directori. Aquest, quan la rep, comunica al nucli que la línia que ha demanat la té en mode exclusiu (mitjançant l'anell d'AK). En aquest punt, el director actualitza l'estat de la línia en ME i el nucli l'actualitzarà a Exclusiva un cop rebi, per l'anell de BL, les dades que rep de memòria. Com es pot veure, el TD ha activat el bit corresponent dels bits vàlids.

Figura 21. Petició de línia de memòria que no té cap altre nucli

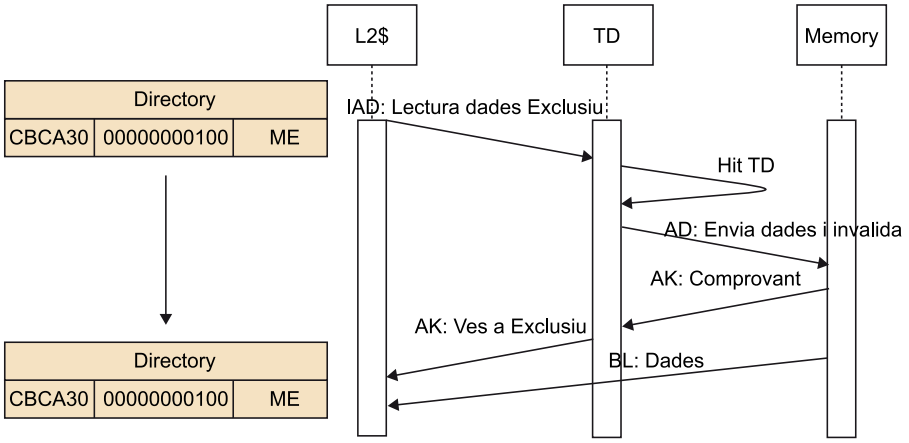


Petició d'una línia en estat exclusiu que ja té un altre nucli

A diferència de l'exemple anterior, en aquest cas s'assumeix que la línia ja estava ubicada en un altre nucli. Per altra banda, també s'assumeix que el nucli demana la línia en exclusiu. Això és necessari en tots el casos en què el nucli ha de modificar el contingut de la línia.

En aquest cas, quan el TD processa la petició del nucli, la cerca dins els TAGS és positiva. Com es pot veure en la figura 22, la cerca retorna que la línia ja està ubicada en el nucli 1 i que està en estat ME. Com que el nucli l'està demanant en exclusiva, el TD envia una petició al nucli que té la línia d'invalidar-la i enviar les dades al nucli que l'ha demanat. Com es pot veure, els bits vàlids són modificats de tal manera que el bit corresponent al primer nucli passa a ser zero i el bit del tercer nucli (el que ha demanat la línia) passa a ser 1.

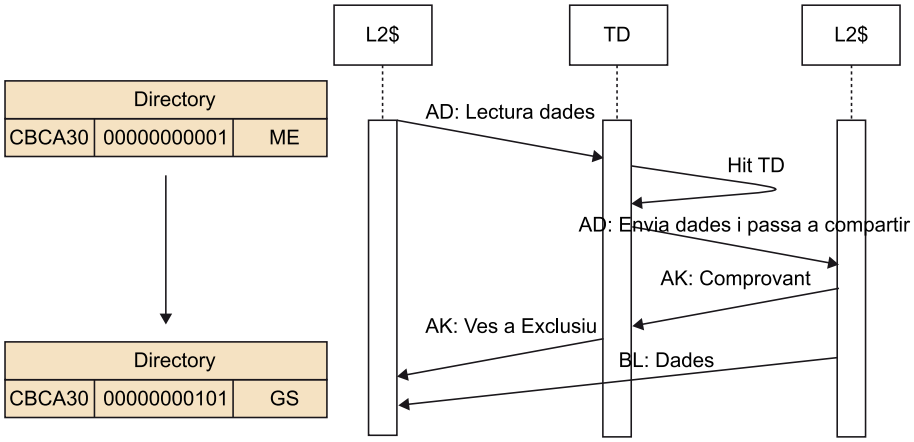
Figura 22. Petició d'una línia de memòria que ja té un altre nucli



Petició d'una línia que ja té un altre nucli

En aquest darrer exemple, a diferència del cas anterior, la línia no demana l'exclusivitat del nucli. Per tant, el TD demana al nucli que ja té la línia que la reenviï al nucli que l'ha demanat. En aquest cas, però, notifica al nucli que tenia la línia en estat exclusiu que passi a compartit o *shared*. Tal com es pot veure en la part esquerra de la figura 23, els bits vàlids passen de tenir el bit del nucli 1 a 1, i mantenen l'estat del bit del nucli 1 a 1. D'aquesta manera, el TD s'apunta que els dos nuclis tenen la línia en qüestió i que aquesta es troba en estat GS.

Figura 23. Petició d'una línia de memòria que ja té un altre nucli



5.7. Conclusions

Durant aquesta secció s'han presentat les pinzellades generals i més representatives de les primeres arquitectures Many Integrated Core que Intel va treure al mercat sota la família anomenada Xeon Phi. Se n'han discutit les característiques generals com a sistema, arquitectura, coherència i nucli.

No obstant això, com ja s'ha esmentat, aquestes contenen una quantitat substancialment més elevada de detalls i definicions. És per aquest motiu que es recomana la lectura d'altres fonts que aprofundeixin més en els detalls esmentats en aquest document, com per exemple: *Corporation, Intel® Xeon Phi™ Coprocessor - the Architecture, The first Intel® Many Integrated Core (Intel® MIC) architecture product* (2013). També es recomana cercar en la Xarxa nous documents que donin més detalls d'aquesta família o bé que expliquin arquitectures de les quals encara no s'havien donat detalls a l'hora d'escriure aquesta documentació (per exemple: el processador Knights Landing).

En aquesta introducció a la família Xeon Phi, no s'han cobert detalls del seu model de programació. Tot i haver introduït el model general en la primera subsecció, Intel ha estès models de programació ja existents com, per exemple, OpenMP (OpenMP), Cilck (Corporation, Intel® Cilk™ Plus) o Intel-TBB (Corporation, Threading Building Blocks). Per altra banda, ha donat noves directives per poder treballar amb aquestes noves arquitectures. Aquestes extensions (majoritàriament accessibles mitjançant directives de tipus *pragma*) permeten, per una banda, interaccionar amb els KNC, per exemple, moure dades al MIC, crear o destruir fils, fer *map o reduce* de variables, etc.; i, per altra banda, permeten usar noves funcions proporcionades per aquesta arquitectura, per exemple (algunes ja disponibles en altres sistemes), *software prefetchs*, FMA, etc. És per aquest motiu que també es recomana la lectura de documents que donen més detalls del model de programació: *Corporation, Intel® Xeon Phi™ Coprocessor Developer's Quick Start Guide* (2013); Corporation, *An Overview of Programming for Intel® Xeon® processors and Intel® Xeon Phi™ coprocessors*, Roth (2013).

5.8. Knights Landing Xeon Phi

Tal com s'ha comentat en seccions anteriors, KNC i KNF varen ser de les primeres arquitectures que la gamma de processadors Intel (amb el nom d'Intel Xeon Phi) va treure al mercat per a computació d'altes prestacions. A mode de resum, les característiques més rellevants d'aquests respecte als altres processadors existents eren:

- De la mateixa manera que les GPU, KNC i KNF estaven basades en una connexió PCIe. És a dir, eren acceleradors connectats a uns processadors als quals es podia fer *offloading* de certes parts del codi, les quals es marcaven amb directives (*pragmes*).

- Pel fet que estan connectades via PCIe, no són *bootables*, és a dir, que no poden arrencar una aplicació que podria córrer sobre un servidor x86 estàndard. Per tant, binaris que estiguin compilats per a ser executats sobre KNC o KNF haurien de ser recompilats per altres arquitectures x86 si fos necessari.

- Tal com ja s'ha comentat en relació amb el model de les GPU, els KNF i KNC (com a model d'acceleració) tenen una memòria pròpia que no és

compartida amb el processador principal. Per tant, cada vegada que es volen moure dades del processador principal al KNF o KNC cal moure les dades de la memòria de l'amfitrió (Xeon) al KNC o KNF.

- Pel que fa als models de coherència i xarxa d'interconnexió feta servir per a connectar els diferents components del processador (directoris, nuclis, etc.), són similars als que havien fet servir generacions anteriors de processadors Intel. El model de coherència era un MOESIF amb GOLS i la xarxa d'interconnexió era un anell bidireccional connectat per diferents tipus de canals (un per a dades, un per a peticions, etc.).

- El nombre de nuclis inclosos dins KNF i KNF era substancialment més elevat que els processadors Xeon que hi havia en aquella època per a intentar maximitzar al màxim el nivell de TLP que el processador ofereix a l'aplicació. Un Xeon estàndard incloïa prop de 8-12 nuclis, mentre que un Xeon Phi podia arribar a 61 nuclis. Ara bé, la complexitat del nucli era força més senzilla que un nucli d'un Xeon estàndard (per exemple, essent un processador en ordre).

Malgrat que els KNF i KNC van ser un primer pas cap a un processador específic per a càlculs científics i per a altes prestacions, aquest tenia un conjunt d'elements que es podien millorar per a fer-lo més competitiu respecte a altres productes (com les GPU) disponibles en el mercat:

- Tal com s'ha esmentat, calia portar aplicacions HPC existents a KNC/KNF. En termes d'esforç, això implicava que molts usuaris es veiessin obligats a migrar a aquests nous tipus de sistemes. Ara bé, en molts casos això no era possible. Per exemple, algunes de les aplicacions més rellevants en entorns HPC podien tenir milions de línies de codi que caldria adaptar a aquesta nova arquitectura. Per tant, evolucionar KNC/KNF cap a una arquitectura compatible amb x86 estàndard seria un aspecte important a tenir en compte.

- El fet de tenir un model discret implica que l'aplicació ha de moure les dades de l'amfitrió cap al dispositiu cada vegada (de manera similar al codi que és executat). En molts casos, això pot afegir un cost extra que té un impacte en el rendiment. Per altra banda, probablement tenir un model *bootable* que permetés executar directament aplicacions permetria córrer de manera més senzilla i natural aplicacions OpenMP i MPI (les més freqüentment emprades en entorns HPC).

- La quantitat de memòria proporcionada (16 GB) pot ser emprada per a fer càlculs per a un determinat tipus de problemes. Ara bé, per a moltes aplicacions HPC el problema emprat (anomenat en anglès *working set*) acostuma a ser major de 16 GB. Per tant, desenvolupar nous esquemes per permetessin tenir una solució HPC amb més memòria seria interessant.

- Usar una interconnexió de tipus anell té l'avantatge que és una solució més senzilla de desenvolupar. No obstant, pot mostrar dues limitacions importants:
  - **Latència variable**. Depenent de «qui» estableixi una comunicació amb «qui», la latència pot ser 16 cops més llarga. Per exemple, assumint un cicle per a parlar amb l'element següent, un element X tarda 1 cicle a parlar amb l'X+1 i 16 per a parlar amb l'element x+16.

  - **Colls d'ampolla**. Pel fet que tots els elements estan connectats amb un anell bidireccional, es donen situacions en què determinats components connectats a aquest poden esdevenir veïns molestos (en anglès *noisy neighbors*). Per exemple, si un nucli N està enviant moltes peticions a memòria o a un altre nucli, el nucli N+1 pot tenir problemes per a usar l'anell, ja que sempre està ocupat per peticions del nucli N.

- Tenir molts nuclis per tal d'obtenir rendiment incrementant el TLP és important. Ara bé, el fet que estiguin en ordre els fa més complexos de programar. Per exemple, quan s'envien peticions a memòria, el fil en qüestió queda bloquejat fins que les dades tornen de memòria. En aquest cas anar cap a arquitectures de nucli més complexes seria una manera de millorar la seva programabilitat i rendiment.

Activitats

1. Busqueu a la xarxa comparatives entre KNC i KNL. Identifiqueu quins són els elements més importants que mencionen altres fonts i com es relacionen amb el ja esmentat en aquesta secció.

2. Per què l'ús de tècniques de *prefetching* poden ajudar a millorar el rendiment d'arquitectures de nucli en ordre?

Per tal d'adreçar els punts esmentats anteriorment i d'altres no discutits, Intel va dissenyar la següent generació de processadors Intel anomenada Intel Knights Landing. En les subseccions següents s'expliquen les característiques més rellevants de l'arquitectura.

La figura 24 mostra els detalls generals de l'arquitectura del processador KNL. Com es pot veure, l'arquitectura d'interconnexió, el tipus de memòria i altres elements arquitectònics són força diferents respecte de KNF o KNC.

Abans d'entrar a discutir cadascun dels elements rellevants de l'arquitectura, una de les coses importants a mencionar és que una de les novetats importants respecte a KNC és que KNL és una arquitectura *bootable*. És a dir, és un processador que pot córrer sistemes operatius i binaris compatibles amb arquitectures x86 i, per tant, que pot ser l'únic processador del servidor o de la placa on està ubicat. Tot i semblar una diferència força petita, pel que fa a l'arquitectura implica moltes més complexitats que un coprocessador o accelerador no cal

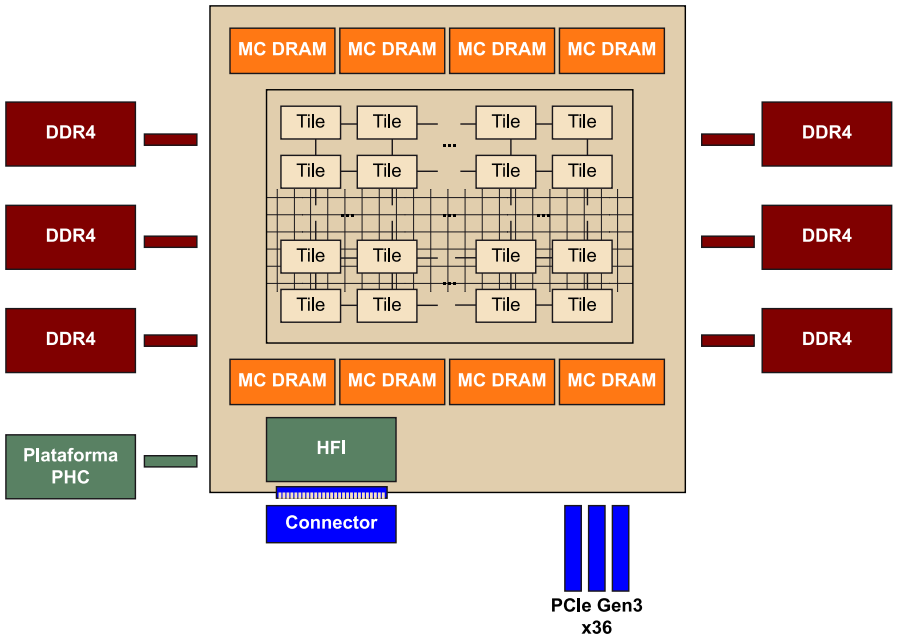
que tingui en compte. Per exemple, ha de donar suport a un conjunt de mapatges de memòria (anomenats en anglès *system address mapping*) que un coprocessador no caldria que suportés.

Per altra banda, no és tan sols un tema de complexitat. El fet que un KNL sigui *bootable* i pugui actuar com a processador principal d'una placa (en anglès *platform* o *board*) permet que es puguin connectar directament a un KNL diferents tipus de dispositius que en una targeta discreta no es podrien connectar. Per exemple, es poden usar els canals de PCIe per a connectar discos de NVM (*none volatile memory*) directament al processador.

Activitat

Us recomanem cercar què és el *system address decoder* de les arquitectures d'Intel. És important parar atenció als diferents tipus d'adreces que un processador x86 ha de suportar.

Figura 24. Arquitectura KNL



Durant les properes subseccions es presentaran algunes de les característiques més importants de KNL. No obstant, algunes d'aquestes (per exemple, el tipus de connexions del processador: PCIE, Ominpath, etc.) no es discuteixen en detall. Es recomana a l'estudiant buscar més fonts si vol aprofundir més en els detalls d'aquesta arquitectura.

5.8.1. Xarxa d'interacció

Una de les millores substancials respecte a KNC és la xarxa d'interconnexió que els KNL usen per a connectar els diferents elements que aquest conté (nuclis, memòries, directoris, etc.). Tal com es pot observar en la figura anterior, el tipus de xarxa que es va escollir per a un KNL és una *mesh* o malla 2D.

La diferència fonamental entre una *mesh* 2D i una 3D és que en una 2D, quan un component que està ubicat al final o inici d'una columna o una fila es vol comunicar amb un component que està ubicat justament al final oposat (columna o fila respectivament), aquest haurà d'enviar un missatge que travessarà tota la columna o fila fins a arribar al seu destinatari. En cas d'una *mesh* 3D, els extrems de les columnes i files estan interconnectats.

Tot i que podria semblar obvi que escollir una *mesh* 3D seria molt millor en termes de latència i amplada de banda, hi ha certes implicacions que ho fan molt més complex quant a implementació. Tal com s'ha comentat en capítols anteriors, una de les coses que es busquen amb arquitectures de processadors és que siguin al més regulars possible. D'aquesta manera, a l'hora de fer el disseny físic (en anglès anomenat *floorplant*) fa que les opcions o implementacions siguin molt més factibles. En aquest cas, usar una *mesh* 3D hauria implicat una sèrie de complicacions importants a l'hora d'enrutar els cables que creuessin tota una columna fins a arribar a l'extrem oposat. De manera similar, pel que fa a l'arquitectura, una *mesh* 3D tindria implicacions importants sobre com fer-ne l'arquitectura. Per exemple, les latències de comunicar els components en extrems oposats (tot i ser contigus quant a lògics) serien força més altes que les latències de comunicar components contigus lògicament i físicament.

Respecte a un KNC, les millores més importants que la nova xarxa d'interconnexió ha portat són:

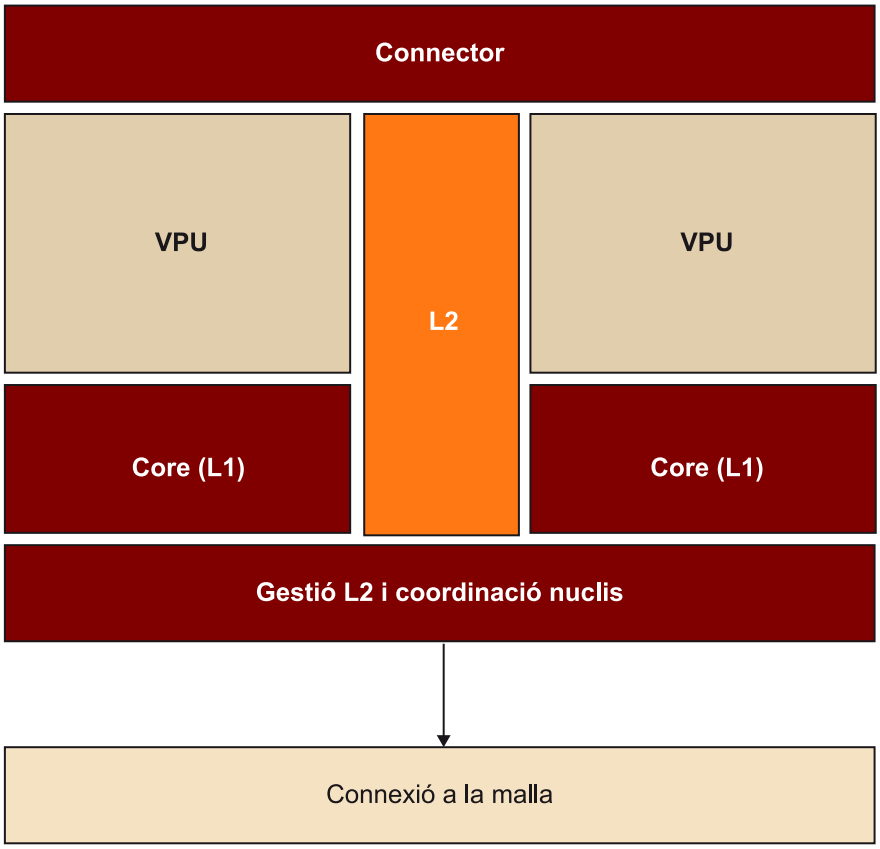
1) Menys latència. A diferència de KNC, en què les latències podien ser més altes per a comunicar-se amb elements que estiguessin més llunyans de l'anell (per exemple, 16 salts), en el cas d'una *mesh* les latències són sempre més baixes. En aquest cas, si teniu un total de 36 components que voleu comunicar podeu fer una *mesh* de 6x6. Per tant, la latència màxima que tindreu sempre serà de 6 salts com a màxim, mentre que en un anell serà de 18 salts.

2) Millor amplada de banda. De manera similar, el fet que un component pugui comunicar-se amb N components diferents fent servir rutes potencialment separades permet que el nombre mitjà de paquets enviats sigui més elevat. Cal recordar que, com que tots els components comparteixen un anell, en un anell hi poden haver situacions de contenció més freqüents.

5.8.2. Nuclis i models de coherència

Una de les novetats importants respecte a KNC és com són estructurades les unitats o elements de computació . Tal com s'ha discutit anteriorment, KNF i KNF usaven uns nuclis simples que executaven les instruccions en ordre. KNL introdueix nuclis SMT amb quatre fils per nucli fora d'ordre. A part de ser fora d'ordre, el nou tipus de nucli és força més potent (té més instruccions en vol, més quantitat de peticions a memòria, millors *prefetcher* a memòria, etc.). De manera similar a KNC, el nombre de fils per nucli és 4.

Figura 24. Visió general d'un *tile*



Activitat

Busqueu entre els diferents processadors del mercat quines són la quantitats de fils que acostumen a tenir els seus nuclis. Per a quin motiu creieu que aquest nombre no acostuma a ser molt elevat? En el casos en què és molt elevat, per què?

No obstant, la diferència probablement més rellevant és que ara els nuclis són agrupats en el que s'anomena *tile* (*rajola* en català). Un *tile* agrupa dos nuclis que comparteixen una L2 (cadascun té una L1 de dades i instruccions pròpia). Aquesta L2 és gestionada per una lògica que implementa les funcionalitats següents:

- Processa peticions dels agents de coherència (que usen una estructura de directoris) per a invalidar o canviar l'estat de línies que té l'L2.

- Processa peticions dels nuclis per a accedir a les línies que potencialment són en l'L2 i per a enviar-ne víctimes. Si les línies demanades pels nuclis no són en l'L2, la lògica genera una petició al directori.

- Manté la coherència entre les L1 dels dos nuclis. És a dir, si aquests estan compartint dades la lògica s'encarregarà que les dades siguin coherents entre ambdós, per exemple, invalidant el nucli 1 si té una línia de memòria que demana el nucli 2.

Altres elements importants nous que KNL té respecte a KNC són les dues unitats vectorials que són adjuntes a cadascun dels nuclis. Aquestes unitats vectorials operen amb registres de 512 bytes i tenen un total de 32 registres de precisió simple i 16 de doble precisió (ambdós tipus de registres de 512 bytes, tal com s'ha mencionat). Aquest tipus d'unitat vectorial implementa instruccions vectorials del tipus X87, SSE1, AVX1, AVX2 i EMU.

Activitat

Busqueu i analitzeu algunes de les instruccions vectorials que cadascun dels tipus esmentats anteriorment exposa en el programari.

El darrer element important que conté el *tile* és el directori, anomenat també *caching home agent* (*CHA*). Aquest element és responsable de mantenir la coherència de memòria entre els diferents nuclis del sistema usant un sistema de coherència MESIF. De manera similar al TD de KNF/KNC, cada CHA és responsable de mantenir la coherència d'un subespai d'adreces de memòria.

5.8.3. Subsistema de memòria

Una de les novetats més importants de KNL és el subsistema de memòria. A diferència de KNF/KNC i de tots els processadors que Intel havia fabricat fins al moment, KNL inclou un sistema de memòria híbrid: dos tipus de memòries diferents que poden ser configurades en diferents tipus de modes.

5.8.4. MCDRAM

La MCDRAM, o memòria d'alt amplada de banda, és un nou tipus de memòria de menys capacitat (de prop de 16 GB) que es caracteritza per ser més petita que la memòria principal però per donar una amplada de banda molt més elevada que la primera. Per tenir-ne una referència, mencionem que per al *benchmark* Stream Triad KNL dona una amplada de banda de 400 GB/s (es deixa a l'estudiant esbrinar les característiques d'aquest *benchmark* i per què es representatiu a l'hora d'avaluar el subsistema de memòria).

Una característica important d'aquest tipus de memòria és que es diu *in-package memory* (memòria dins el processador). Tal com es pot observar en la figura següent (una fotografia d'un processador KNL de ®Intel), la memòria està soldada dins el processador. Com es veurà a continuació, en processadors tradicionals la memòria (de tipus DRAM) és ubicada a la placa i connectada al processador mitjançant canals especials.

Figura 25. Memòria MCDRAM dins del xip

5.8.5. DRAM (DDR)

El segon tipus de memòria que KNL ofereix és de tipus DRAM (*dyamic random access memory*). Aquesta memòria, la més tradicional i habitual en arquitectures de processadors, es caracteritza per donar una accés de latència similar al de la MCDRAM però amb una amplada de banda molt més baixa. En el cas del *benchmark* Stream Triad KNL, dona aproximadament una amplada de banda de 90 GB/s (més de quatre vegades menys que l'MCDRAM).

Ara bé, l'avantatge primordial que la DRAM té respecte a l'MCDRAM és la seva capacitat. Mentre que una MCDRAM està limitada a 16 GB/s (recordeu que és soldada dins del processador), la DRAM pot arribar a més de 300 GB/s (depenent de la placa). El preu de poder tenir més capacitat és que ja no es pot ubicar dins el processador sinó que cal posar-la a la placa. Tal com es mostra en la figura següent, el quadrat blanc és on es connecta el processador i les ranures negres (part esquerra superior de la figura) és on es posen les peces de memòria DDR. Els cables que connecten aquestes ranures i el processador són el factor limitant més important quant a amplada de banda.

Figura 26. Exemple de placa amb ranures de DDR a la part superior esquerra (en negre)

5.8.6. Modes de memòria

Una de les novetats més importants associada a tenir dos tipus de memòria diferent és que cal facilitar mecanismes per tal que les aplicacions les puguin fer servir. En aplicacions i plataformes tradicionals, quan es reserva memòria (per exemple, mitjançant la funció *malloc*), el sistema operatiu reserva una part de la memòria física (per exemple, DDR), l'associa a l'espai d'adreces virtual de l'aplicació i en retorna el punter virtual a l'aplicació.

Una de les qüestions que l'estudiant s'ha de plantejar pel que fa a KNL és: «Com reservaran les aplicacions memòria física a la DDR o l'HBM?». Tal com es pot intuir, un dels mecanismes obvis és estendre els ja existents per tal que l'aplicació pugui especificar al sistema operatiu quin tipus de memòria es vol usar quan es reserva memòria. De la mateixa manera, el processador ha d'exposar interfícies al sistema operatiu per tal que aquest pugui accedir, configurar i assignar qualsevol de les dues memòries. És important esmentar que en aquest cas l'aplicació ha de ser conscient quan reserva i demana memòria d'un tipus, les característiques de la qual són: HBM (més amplada de banda i menys capacitat); DDR (menys amplada de banda però més capacitat). Per tant, les variables que es mapin en l'HBM podran ser accedides amb més velocitat (però se n'hi podran assignar menys) i les que es mapin la en la DDR podran ser accedides amb menys velocitat (però se n'hi podran assignar moltes més). Aquest tipus d'accés és el que s'anomena *flat memory mode*.

KNL, per tal de facilitar la programació de les aplicacions i que aquestes puguin ser agnòstiques dels tipus de memòria, ofereix el tipus d'accés anomenat *memory side cache mode*. En aquest mode, la memòria HBM es comporta com una memòria cau de la DDR. Per tant, quan l'aplicació demana una dada, el sistema de memòria de KNL primer mirarà si es troba en l'HBM. Si no hi és, es generarà una víctima a l'HBM (per fer espai per a la dada que demana

l'aplicació) i la demanarà a la DDR. Un cop la DDR rebi la petició, aquesta retornarà les dades a l'HBM, que la guardarà (actuant com a memòria cau), i al nucli on l'aplicació que demani les dades s'estigui executant. Com es pot observar, el sistema de memòria es comportarà com una cau tradicional.

Activitat

Busqueu quines són les interfícies que exposa el sistema operatiu i quines llibreries facilita Intel per tal d'accedir a les diferents memòries.

Resum

En aquest mòdul didàctic s'han presentat els fonaments arquitectònics de processadors necessaris per a poder entendre la computació d'altes prestacions. Primerament, després d'una breu introducció que tractava de la descomposició funcional i de dades, s'ha presentat la taxonomia de Flynn, de la qual es deriva una caracterització de les diferents arquitectures de computadors que es poden trobar en l'àmbit de computació. A continuació s'han tractat les arquitectures *single instruction, multiple data* (SIMD) i les *multiple instruction, multiple data* (MIMD).

Tot seguit s'ha presentat l'arquitectura d'un dels processadors més estesos dins el món de la computació: les arquitectures vectorials. Més endavant, en tractar de les MIMD, s'ha fet una anàlisi detallada de les diferents variants. Començant per les arquitectures *superthreading* i acabant per les arquitectures multinucli.

Com s'ha pogut veure, millorar el rendiment que les diferents arquitectures proporcionen no és una tasca senzilla. En molts casos hi ha factors que en limiten l'escalabilitat o que en fan les millores extremadament costoses. Per exemple, en un processador multinucli, depenent de quin tipus de xarxa d'interconnexió s'usi, el nombre de nuclis que pot suportar, no escala fins a un nombre gaire elevat (per exemple, una xarxa d'interconnexió de tipus bus).

Després s'han estudiat alguns dels factors més importants que cal tenir en compte a l'hora d'avaluar l'escalabilitat i rendiment d'un processador d'altes prestacions (com ara el protocol de coherència). Per tal de poder treure el màxim rendiment a les aplicacions, cal tenir en compte certes limitacions o restriccions que aquest tipus d'arquitectures presenten. En el darrer apartat hem presentat un conjunt d'aspectes importants que cal tenir en compte a l'hora de dissenyar-les i desenvolupar-les. Per exemple, s'ha presentat l'impacte que la compartició falsa té en l'accés a dades. En aquest cas, diferents fils d'execució accedeixen a dades que es troben ubicades a diferents direccions físiques, però que comparteixen la mateixa entrada de la memòria cau. Això fa que competeixin pel mateix recurs de manera continuada, cosa que en degrada molt el rendiment. Tal com ja s'ha esmentat, la compartició falsa es pot evitar afegint el que s'anomena *padding*, és a dir, una de les dues dades es declara desplaçada per tal que es mapegi en una adreça física i, per tant, no coincideixi amb la mateixa entrada de memòria cau.

Tal com s'ha fet palès durant tot el mòdul didàctic, la complexitat d'aquest àmbit de recerca és força elevat. Les generacions noves de computadors que el mercat demana requereixen cada vegada més capacitat de procés. Aquesta capacitat va tan lligada a la quantitat de processament que aquestes faciliten

(el nombre de nuclis i fils d'execució) com a la capacitat de processar dades (per exemple, per mitjà d'unitats vectorials). Això és especialment important en la computació d'altes prestacions.

És recomanable, doncs, que l'alumne llegeixi algunes de les referències proporcionades al llarg del mòdul. Una lectura en profunditat dels articles recomanats dóna una perspectiva més detallada dels factors que es consideren més importants dins de la recerca feta.

Activitats

1. En el subapartat 5.1 s'han tractat diferents factors que cal tenir en compte a l'hora de programar arquitectures multifil. No obstant això, les arquitectures vectorials també tenen reptes i factors que cal tenir en compte a l'hora de programar-les. Per aquesta activitat es proposa aprofundir en les arquitectures vectorials amb la lectura de l'article següent: "Intel® AVX: New Frontiers in Performance Improvements and Energy Efficiency" (Firasta, Buxton, Jinbo, Nasri i Kuo, 2008).

2. "L'exemple de suma vectorial" de l'apartat 3 mostra una comparativa d'un mateix algorisme de suma usant un codi escalar i un codi vectorial. Assumint que fer una *load* i un *store* tarden 10 cicles i 15 cicles en escalar i vectorial, respectivament, i que una suma escalar i vectorial tarden 1 i 2 cicles, respectivament, quina *speedup* obtindríem en la versió vectorial respecte de l'escalar?

3. Per a aquesta activitat es proposa buscar i explicar dos problemes o algorismes diferents que es puguin beneficiar de la computació SIMD. A l'apartat 3, dedicat a les arquitectures SIMD, s'han enumerat alguns d'aquests algorismes. No obstant això, no s'ha detallat com han adaptat els seus algorismes per tal d'emprar les operacions vectorials per a processar les dades que usen. Cal estudiar i explicar-ne els detalls i fer una estimació de les diferències pel que fa a rendiment de la versió vectorial respecte de l'escalar.

4. Expliqueu quines són les diferències més importants entre un protocol basat en directori i un de basat en *snoops* i en quines situacions creieu que un protocol basat en directori és més adient que un de basat en *snoops* i viceversa. Raoneu la resposta.

5. Una de les mètriques més importants que cal optimitzar en el disseny de processadors és el consum energètic. En cap dels apartats anteriors s'ha detallat l'impacte de la complexitat dels processadors multifil o multinucli en el consum energètic. Enumereu les aportacions més interessants de l'article següent en aquest aspecte: "Effects of Pipeline Complexity on SMT/CMP Power-Performance Efficiency" (Lee i Brooks, 2005).

6. Implementeu un codi multifil en què hi pugui haver un problema de carrera d'accés. Cal que raoneu el problema que pot aparèixer en temps d'execució. Feu el mateix exercici, però en un codi en què el programa es pugui bloquejar (és a dir, acabar en un *deadlock*).

7. Les arquitectures multinucli han agafat molta importància durant els darrers anys. Permeten que aplicacions que poden escalar el seu rendiment si tenen més font de paral·lelisme se'n beneficiïn molt. Larrabee va ser un multinucli que l'empresa Intel va proposar el 2008. Per aquesta activitat cal llegir l'article següent i fer una llista amb 15 característiques positives d'aquesta arquitectura i 15 de negatives: "Larrabee: A Many-Core x86 Architecture for Visual Computing" (Seiler i altres, 2008).

Bibliografia

500, G. T. (n.d.). *Green Top 500*. Retrieved 8 28, 2013, from http://www.green500.org/

Agrawal, D. P.; Feng, T. Y.; Wu, C. L. (1978). "A survey of communication processors systems". *Proc. COMPSAC* (pàg. 668-673).

Alverson, R.; Callahan, D.; Cummings, D.; Koblenz, B. (1990). "The Tera computer system". *International Conference on Supercomputing* (pàg. 1-6).

AMD (2011). *AMD Athlon™ Processor*. Recuperat el 4 de gener de 2012 d'AMD Athlon™ Processor.

anandtech (n.d.). *AMD Radeon HD 7970 Review: 28nm And Graphics Core Next, Together As One*. Retrieved 08 29, 2013, from http://www.anandtech.com/show/5261/amd-radeon-hd-7970-review/10

Andrews, G. R. (1999). *Foundations of Multithreaded, Parallel, and Distributed Programming*. Reading, Massachusetts: Addison-Wesley.

Bailey, D., Barton, J., Lasinski, T., i Simon, H. (1991). "The NAS parallel benchmarks". Technical Report RNR-91-002 Revision 2, 2, NASA Ames Research Laboratory.

Baniwal, R. (2010). "Recent Trends in Vector Architecture: Survey". *International Journal of Computer Science & Communication* (vol. 1, núm. 2, pàg. 395-339).

Bell, S., Edwards, B., Amann, J., Conlin, R., Joyce, K., Leung, V. i altres (2008). "TILE64 Processor: A 64-Core SoC with Mesh Interconnect". *IEEE International Solid-State Circuits Conference*.

Ceder, A., i Wilson, N. (1986). "Bus network design". *Transportation Design* (pàg. 331-344).

Chaiken, D., Fields, C., Kurihara, K., i Agarwal, A. (1990). "Directory-based cache coherence in large-scale multiprocessors". *Computer* (pàg. 49-58).

Chapman, B., Huang, L., Biscondi, E., Stotzer, E., i Shrivastava, A. G. (2008). "Implementing OpenMP on a High Performance Embedded Multicore MPSoC". *IPDPS*.

Chase, J. i Doyle, R. (2001). "Balance of Power: Energy Management for Server Clusters". Proceedings of the 8th Workshop on Hot Topics in Operating Systems.

Chynoweth, M. i Lee, M. R. (2009). "Implementing Scalable Atomic Locks for Multi-Core". Recuperat el 28 de desembre de 2011 d'Implementing Scalable Atomic Locks for Multi-Core.

Corp, I. (2007). "Intel SSE4 Programming Reference". A: *Intel, Intel® SSE4 Programming Reference* (pàg. 1-197). Dender: Intel Corporation.

Corp, I. (2011). *AVX Instruction Set*. Recuperat el 23 de març de 2012: http://software.intel.com/en-us/avx/.

Corporation, I. (2013). *An Overview of Programming for Intel® Xeon® processors and Intel® Xeon Phi™ coprocessors*. Portland: Intel.

Corporation, I. (2013). *Intel® Xeon Phi™ Coprocessor - the Architecture, The first Intel® Many Integrated Core (Intel® MIC) architecture product*. Portland: Intel.

Corporation, I. (2013). *Intel® Xeon Phi™ Coprocessor Developer's Quick Start Guide*. Portland: Intel.

Corporation, I. (n.d). *Haswell Server Products*. Retrieved 08 29, 2013, from http://ark.intel.com/products/codename/42174/Haswell

Corporation, I. (n.d.). *Intel® Cilk™ Plus*. Retrieved 09 1, 2013, from Intel® Cilk™ Plus: http://software.intel.com/en-us/intel-cilk-plus

Corporation, I. (n.d.). *Introduction to x64 Assembly*. Retrieved 09 01, 2013, from Introduction to x64 Assembly: http://software.intel.com/en-us/articles/introduction-to-x64-assembly

Corporation, I. (n.d.). *Sandy Bridge Server Products*. Retrieved 08 29, 2013, from http://ark.intel.com/products/codename/29900

Corporation, I. (n.d.). *Threading Building Blocks*. Retrieved 08 29, 2013, from Threading Building Blocks: www.threadingbuildingblocks.org/

Dixit, K. M. (1991). "The SPEC benchmark". *Parallel Computing* (pàg. 1195-1209).

Culler, D. E. i Pal Singh, J. (1999). *Parallel computer architecture: a hardware/software approach*. Burlington, Massachusetts: Morgan Kaufmann.

Firasta, N.; Buxton, M.; Jinbo, P.; Nasri, K.; Kuo, S. (2008). "Intel® AVX: New Frontiers in Performance Improvements and Energy Efficiency". *Intel White Paper*.

Fisher, J. (1893). "Very Long Instruction Word Architectures and the ELI-512". *Proceedings of the 10th Annual International Symposium on Computer Architecture* (pàg. 140-150).

Flich, J. (2000). "Improving Routing Performance in Myrinet Networks". *IPDPS*.

Gibbons, A.; Rytte, W. (1988). *Efficient Parallel Algorithms*. Cambridge: Cambridge University Press.

Grunwald, D., Zorn, B., i Henderson, R. (1993). "Improving the cache locality of memory allocation". ACM SIGPLAN 1993 Conference on Programming Language Design And Implementation.

Handy, J. (1998). *The cache memory book*. Londres: Academic Press Limited.

Hennessy, J. L. i Patterson, D. A. (2011). *Computer Architecture: A Quantitative Approach*. Burlington, Massachusetts: Morgan Kaufmann.

Herlihy, M. i Shavit, N. (2008). *The Art of Multiprocessor Programming*. Burlington, Massachusetts: Morgan Kaufmann.

Hewlett-Packard (1994). "Standard Template Library Programmer's Guide". Recuperat el 9 de gener de 2012 d'Standard Template Library Programmer's Guide.

Hily, S. i Seznec, A. (1998). "Standard Memory Hierarchy Does Not Fit Simultaneous Multithreading". Workshop on Multithreaded Execution, Architecture, and Compilation.

Hong, Y.; Payne, T. H. (1989). "Parallel Sorting in a Ring Network of Processors". *IEEE Transactions on Computers* (vol. 38, núm. 3).

IBM (2011). "AS/400 Systems". Recuperat el 20 de desembre de 2011: http://www-03.ibm.com/systems/i/.

IBM. "System/370 Model 145". Recuperat el 12 de març de 2012 d'IBM Corporation Web Site.

IEEE (2011). "IEEE POSIX 1003.1c standard". Recuperat l'11 de gener de 2012 d'IEEE POSIX 1003.1c standard.

insideHPC. "InsideHPC: A Visual History of Cray". Recuperat l'1 de març de 2012 d'insideHPC Web Site.

Intel (2007). *Intel® Threading Building Blocks Tutorial*.

Intel (2007). *Optimizing Software for Multi-core Processors*. Portland: Intel Corporation - White Paper.

Intel (2011). *1971-2011. 40 años del microprocesador*. Portland: Intel Corporation.

Intel (2011). "Boost Performance Optimization and Multicore Scalability". Recuperat el 3 de gener de 2012 de Boost Performance Optimization and Multicore Scalability.

Intel (2011). "Intel Cilk Plus". Recuperat el 21 de desembre de 2011 d'Intel Cilk Plus.

Intel (2011). *Intel® Array Building Blocks 1.0 Release Notes*.

Intel (2011). "Nehalem Processor". [Recuperat el 4 de gener de 2012 de Nehalem Processor.]

Intel (2011). *Use Software Data Prefetch on 32-Bit Intel / IA-32 Intel® Architecture Optimization Reference Manual*. Portland: Intel Corporation.

Intel (2012). "Intel Sandy Bridge - Intel Software Network". Recuperat el 8 de gener de 2012 d'Intel Sandy Bridge - Intel Software Network.

Intel (2012). "Intel® Quickpath Interconnect Maximizes Multi-Core Performance". Recuperat el 8 de gener de 2012 d'Intel® Quickpath Interconnect Maximizes Multi-Core Performance.

Intel. "Sandy Bride Intel". Recuperat el 10 de desembre de 2011 de Sandy Bride Intel.

Jedec (n.d.). *Jedec*. Retrieved 08 29, 2013, from www.jedec.org/sites/default/files/docs/JESD212.pdf

Jin, R.; Chung, T. S. (2010). "Node Compression Techniques Based on Cache-Sensitive B+-Tree". *9th International Conference on Computer and Information Science* (ICIS) (pàg. 133-138).

Joseph, D.; Grunwald, D. (1997). "Prefetching using Markov predictors". *24th Annual International Symposium On Computer Architecture*.

Katz, R. H.; Eggers, S. J.; Wood, D. A.; Perkins, C. L.; Sheldon, R. G. (1985). "Implementing a cache consistency protocol". *Proceedings of the 12th Annual International Symposium on Computer Architecture*.

Kim, B. C. i Jun, S. W. H. K. (2009). "Visualizing Potential Deadlocks in Multithreaded Programs". *10th International Conference on Parallel Computing Technologies*.

Kishan Malladi, R. (2011). *Using Intel® VTune™ Performance Analyzer Events/ Ratios & Optimizing Applications*. Portland: Intel Corporation.

Kongetira, P., Aingaran, K., i Olukotun., K. (2005). "Niagara: A 32- Way Multithreaded SPARC Processor". *IEEE MICRO Magazine*.

Kourtis, K., Goumas, G., i Koziris, N. (2008). "Improving the Performance of Multithreaded Sparse Matrix-Vector Multiplication Using Index and Value Compression". *37th International Conference on Parallel Processing*.

Kumar, R., Farkas, K. I., Jouppi, N. P., Ranganathan, P., i Tullsen, D. M. (2003). "Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction". *Microarchitecture, MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on* (pàg. 81-92).

Kwak, H., Lee, B., Hurson, A., Suk-Han, Y., i Woo-Jong, H. (1999). "Effects of multithreading on cache performance". *IEEE Transactions on Computer* (pàg. 176-184).

Lee, B. C. i Brooks, D. (2005). "Effects of Pipeline Complexity on SMT/CMP Power-Performance Efficiency". *Workshop on Complexity Effective Design 2005* (WCED2005, held in conjuction with ISCA-32).

Linux (2010). *Linux Programmer's Manual*. Recuperat el 3 de gener de 2012 de Linux Programmer's Manual.

Lo, J. L. (1997). *ACM Transactions on Computer Systems. Converting Thread-level Parallelism to Instructionlevel Parallelism via Simultaneous Multithreading*.

Lo, J. L., Eggers, S. J., Levy, H. M., Parekh, S. S., i Tullsen, D. M. (1997). "Tuning Compiler Optimizations for Simultaneous Multithreading". *International Symposium on Microarchitecture* (pàg. 114-124).

Mellor-Crummey, J. M. i Scott, M. L. (1991). "Algorithms for scalable synchronization on shared-memory multiprocessors". *ACM Transactions on Computer Systems*.

Marr, D. (2002). "Hyper-Threading Technology Architecture and Microarchitecture: A Hypertext History". *Intel Technology J.* (vol. 8, núm. 1).

Martorell, X., Corbalán, J., González, M., Labarta, J., Navarro, N., i Ayguadé, E. (1999). "Thread Fork/Join Techniques for Multi-level Parallelism Exploitation in NUMA Multiprocessors". *13th International Conference on Supercomputing*.

Melvin, S. (2000). "Clearwater networks cnp810sp simultaneous multithreading (smt) core". Recuperat el 19 de desembre de 2011: www.zytek.com/melvin/clearwater.html.

Melvin, S. (2003). "Flowstorm porthos massive multithreading (mmt) packet processor".

OpenMP (n.d.). *The OpenMP® API specification for parallel programming*. Retrieved 09 01, 2013, from The OpenMP® API specification for parallel programming.

ParaWise (2011). "ParaWise - the Computer Aided Parallelization Toolkit". Recuperat el 27 de desembre de 2011 de ParaWise - the Computer Aided Parallelization Toolkit.

pcwire (2012, 05 12). *Intel Brings Manycore x86 to Market with Knights Corner*. Retrieved 08 29, 2013, from http://www.hpcwire.com/hpcwire/2012-11-12/intel_brings_manycore_x86_to_market_with_knights_corner.html

Philbin, J.; Edler, J.; Anshus, O. J.; Douglas, C.; Li, K. (1996). "Thread scheduling for cache locality". *7th International Conference on Architectural Support for Programming Languages and Operating Systems*.

Prasad, S. (1996). *Multithreading Programming Techniques*. Nova York: McGraw-Hill, Inc.

Ramos Garea, S.; Hoefler, T. (2013). *Modelling Communications in Cache Coherent Systems*.

Reed, D. i Grunwald, D. (1987). "The Performance of Multicomputer Interconnetion Networks". *Computer* (pàg. 63-73).

Reinders, J. (2007). *Intel Threading Building Blocks*. O'Reilly.

Roth, F. (2013). *System Administration for the Intel® Xeon Phi™ Coprocessor*. Portland: Intel.

Rusu, S.; Tam, S.; Muljono, H.; Ayers, D.; Chang, J. (2006). "A dual-core multi-threaded Xeon(r) processor with 16 Mb L3 cache". *Proc. 2006 IEEE Int. Solid-State Circuits Conf.* (pàg. 315-324).

Sabot, G. (1995). *High Performance Computing: Problem Solving with Parallel and Vector Architectures*. Reading, Massachusetts: Addison-Wesley.

Seiler, L.; Carmean, D.; Sprangle, E.; Forsyth, T.; Dubey, Junkins, S. P. i altres (2008). "Larrabee: A Many-Core x86 Architecture for Visual Computing". *ACM Transaction on Graphics*.

Seznec, A. (1993). "A case for two-way skewed-associative caches". *20th Annual International Symposium on Computer Architecture*.

Seznec, A.; Felix, S.; Krishnan, V.; Sazeide, Y. (2002). "Design tradeoffs for the Alpha EV8 conditional branch predictor". *Proceedings of the 29th International Symposium on Computer Architecture*.

Song, P. (2002). "A tiny multithreaded 586 core for smart mobile devices". 2002 Microprocessor Forum (MPF).

Stenström, P. (1990). "A Survey of Cache Coherence for Multiprocessors". *IEE Computer Transactions*.

TACC, T. A. (n.d.). *Dell PowerEdge C8220 Cluster with Intel Xeon Phi coprocessors*. Retrieved 09 01, 2013, from Dell PowerEdge C8220 Cluster with Intel Xeon Phi coprocessors: http://www.tacc.utexas.edu/resources/hpc/stampede

Top500 (n.d.). Retrieved 08 28, 2013, from http://www.top500.org/

Top500 (n.d). *China's Tianhe-2 Supercomputer Takes No. 1 Ranking on 41st TOP500* . Retrieved 09 01, 2013, from China's Tianhe-2 Supercomputer Takes No. 1 Ranking on 41st TOP500: http://www.top500.org/blog/lists/2013/06/press-release/

Tullsen, D. M.; Eggers, S.; Levy, H. M. (1995). "Simultaneous multithreading: Maximizing on-chip parallelism". *22th Annual International Symposium on Computer Architecture*.

Valgrind (2011). "Cachegrind: a cache and branch-prediction profiler". Recuperat el 3 de gener de 2012 de Cachegrind: a cache and branch-prediction profiler.

Villa, O.; Palermo, G.; Silvano, C. (2008). "Efficiency and scalability of barrier synchronization on NoC based many-core architectures". *CASES '08 Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*.

Yao, E.; Demers, A.; Shenker, S. (1995). "A scheduling model for reduced CPU energy". *IEEE 36th Annual Symposium on Foundations of Computer Science* (pàg. 374-382).