Arquitectures d'altes prestacions

Ivan Rodero Castro
Francesc Guim Bernat

PID_00191901

UOC
Universitat Oberta de Catalunya

Els textos i imatges publicats en aquesta obra estan subjectes –llevat que s'indiqui el contrari– a una llicència de Reconeixement-NoComercial-SenseObraDerivada (BY-NC-ND) v.3.0 Espanya de Creative Commons. Podeu copiar-los, distribuir-los i transmetre'ls públicament sempre que en citeu l'autor i la font (FUOC. Fundació per a la Universitat Oberta de Catalunya), no en feu un ús comercial i no en feu obra derivada. La llicència completa es pot consultar a http://creativecommons.org/licenses/by-nc-nd/3.0/es/legalcode.ca

Índex

Introducció 5

Objectius 8

1. Descomposició funcional i de dades 9

2. Taxonomia de Flynn 13

3. Arquitectures de processador SIMD 16
- 3.1. Propietats dels processadors vectorials 18
- 3.2. Exemple de processador vectorial 19

4. Arquitectures de processador multifil o MIMD 21
- 4.1. Arquitectures *superthreading* 23
  - 4.1.1. Compartició a nivell fi 23
  - 4.1.2. Compartició a nivell gruixut 25
- 4.2. Arquitectures *simultaneous multithreading* 25
- 4.3. Convertint el paral·lelisme a nivell de fil a paral·lelisme a nivell d'instrucció 26
- 4.4. Disseny d'un SMT 27
- 4.5. Complexitats i reptes en les arquitectures SMT 29
- 4.6. Implementacions comercials de l'SMT 31
  - 4.6.1. L'HyperThreading d'Intel 31
  - 4.6.2. L'Alpha 21464 34
- 4.7. Arquitectures multinucli 36
  - 4.7.1. Limitacions de l'SMT i arquitectures *superthreading* 36
  - 4.7.2. Producció 38
  - 4.7.3. El concepte de multinucli 38
  - 4.7.4. Coherència entre nuclis 40
  - 4.7.5. Protocols de coherència 43

5. Factors determinants en el rendiment en arquitectures modernes 46
- 5.1. Factors importants per a la llei d'Amdahal en arquitectures multifil 47
- 5.2. Factors vinculats al model de programació 48
  - 5.2.1. Definició i creació de les tasques paral·leles 48
  - 5.2.2. Mapatge de tasques a fils 50
  - 5.2.3. Definició i implementació de mecanismes de sincronització 52
  - 5.2.4. Gestió d'accés concurrent a dades 56

5.2.5. Altres factors que cal considerar 62
5.3. Factors lligats a l'arquitectura 62
5.3.1. Compartició falsa 63
5.3.2. Penalitzacions per a fallades a la L1 i tècniques de *prefetch*... 65
5.3.3. Impacte del tipus de memòria cau 67
5.3.4. Arquitectures multinucli i multiprocessador 70

Resum.. 73

Activitats.. 75

Bibliografia... 76

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 601, els Intel Pentium o bé els AMD K5 o AMDK6. 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 tercer 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 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

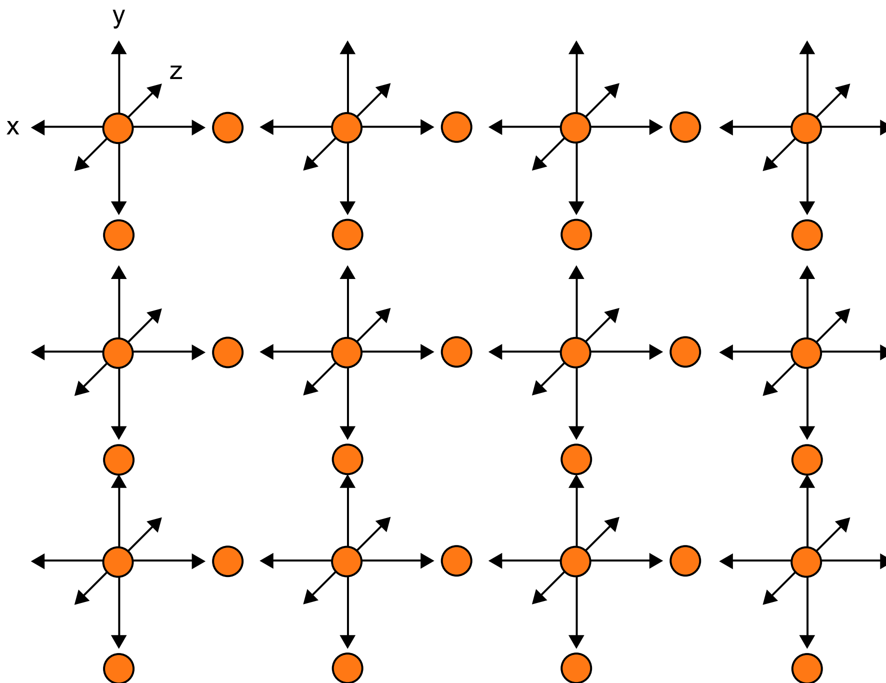
Com es mostra al llarg d'aquesta unitat didàctica la majoria d'arquitectures d'altes prestacions que avui en dia es poden trobar al mercat permeten executar més d'un fil d'execució de manera paral·lela. D'altra banda, algunes també proporcionen accés a unitats de procés especifiques que són capaces d'aplicar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

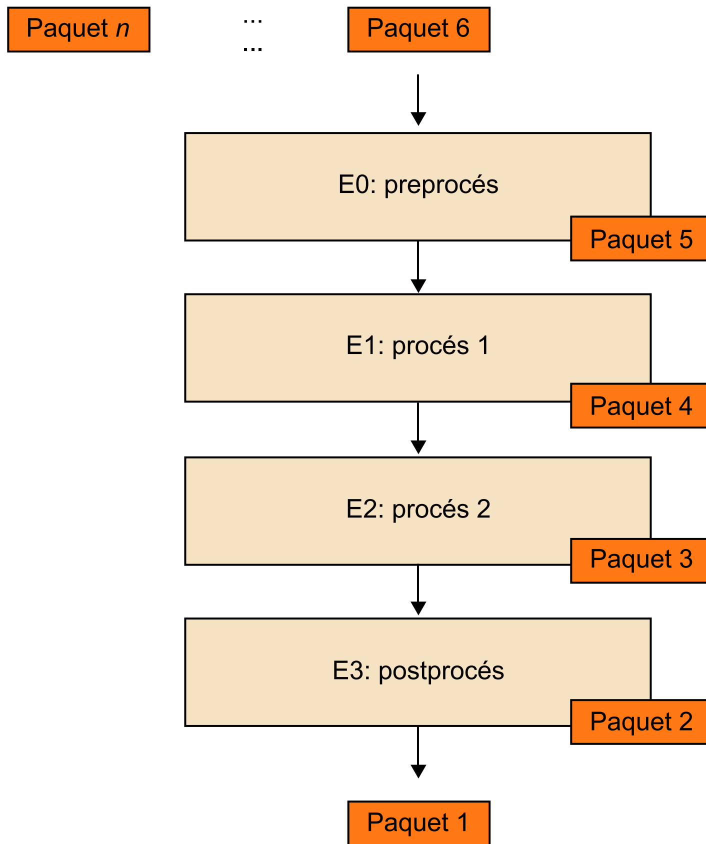
M. S. Lam, E. E. Rothberg i M. E. Wolf (1991).

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



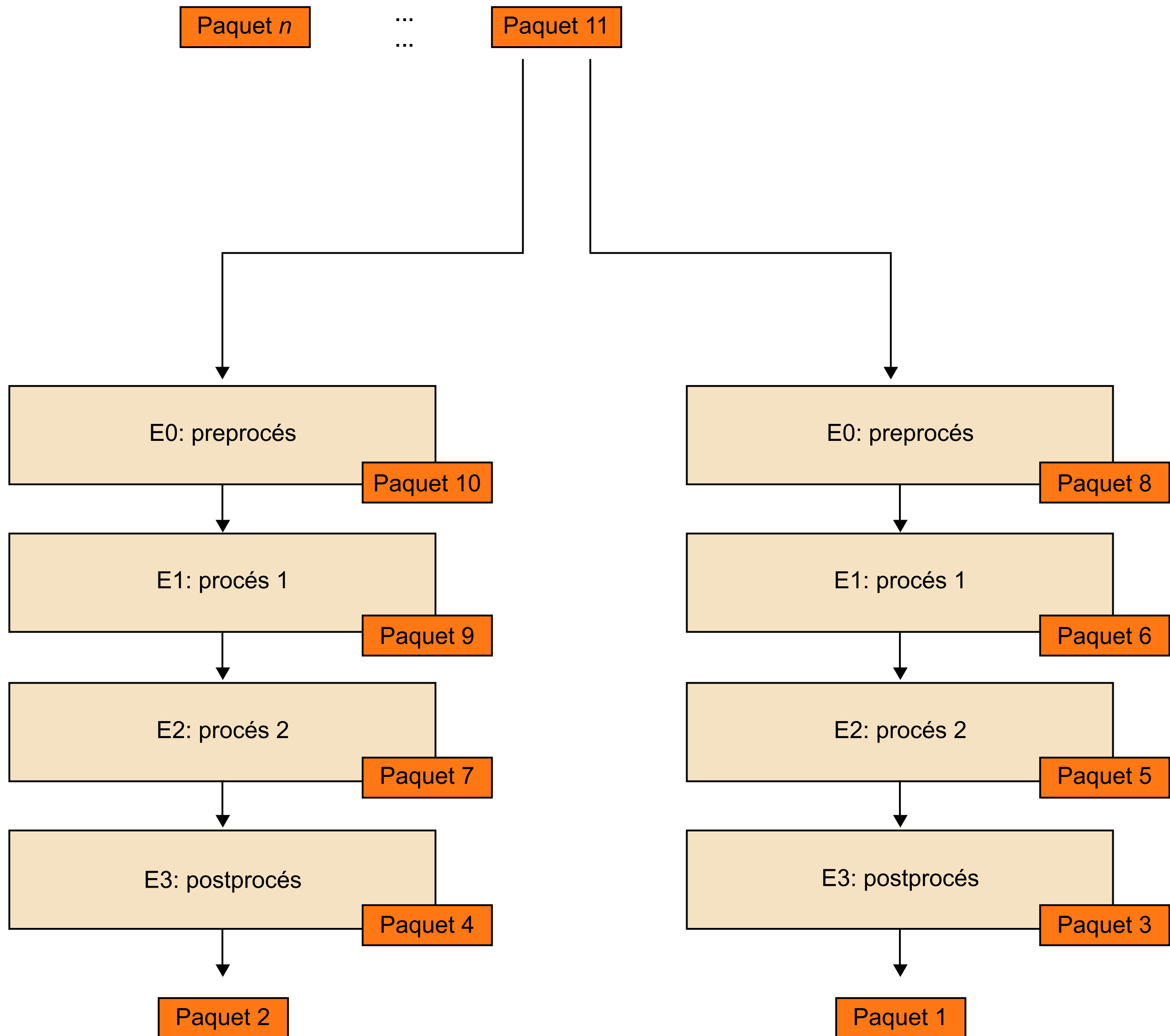
Tot i que els dos exemples anteriors tracten la descomposició de dades i funcional separadament, a la realitat aquest dos problemes estan directament relacionats. A l'hora d'analitzar la paral·lelització d'una aplicació, se n'estudien tant la descomposició funcional com de dades. En general es busca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

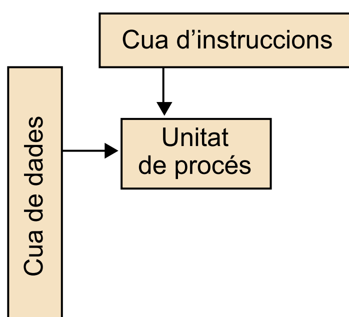
1) Una instrucció, una dada[1]: són les arquitectures tradicionals formades per una sola unitat de procés. Com es pot observar a continuació, no exploten ni el paral·lelisme a nivell de fil, ni el paral·lelisme a nivell de dades. En aquest cas, tenim un sol flux de dades i d'instruccions. La figura següent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

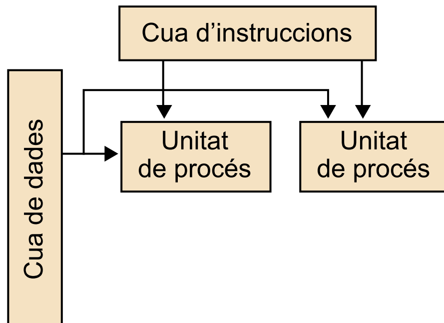
(2)En anglès, *single instruction, multiple data stream* (SIMD).

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



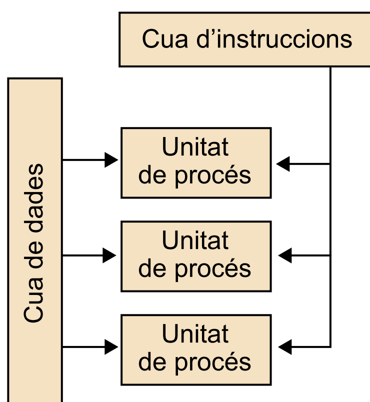
3) Múltiples instruccions, una dada[3]: permeten executar diferents fluxos d'instruccions sobre un mateix bloc de dades. En aquest cas, tenim diferents unitats de procés que operen sobre la mateixa dada. Aquest tipus d'arquitectures és molt específic i no se'n troba cap dins del món de l'arquitectura 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

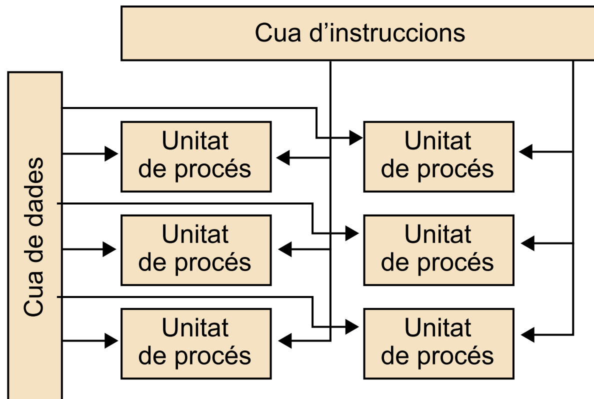


4) Múltiples instruccions, múltiples dades[4]: estan formades per múltiples unitats de procés que treballen sobre diferents fluxos de dades de manera paral·lela. Com mostra la figura següent, l'arquitectura està formada per diferents unitats de procés que tenen diferents fluxos d'instruccions que treballen

(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 < 128; 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(a,B[i]);
   c = suma_vectorial(a,b);
   desa(C[i],a);
}
```

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

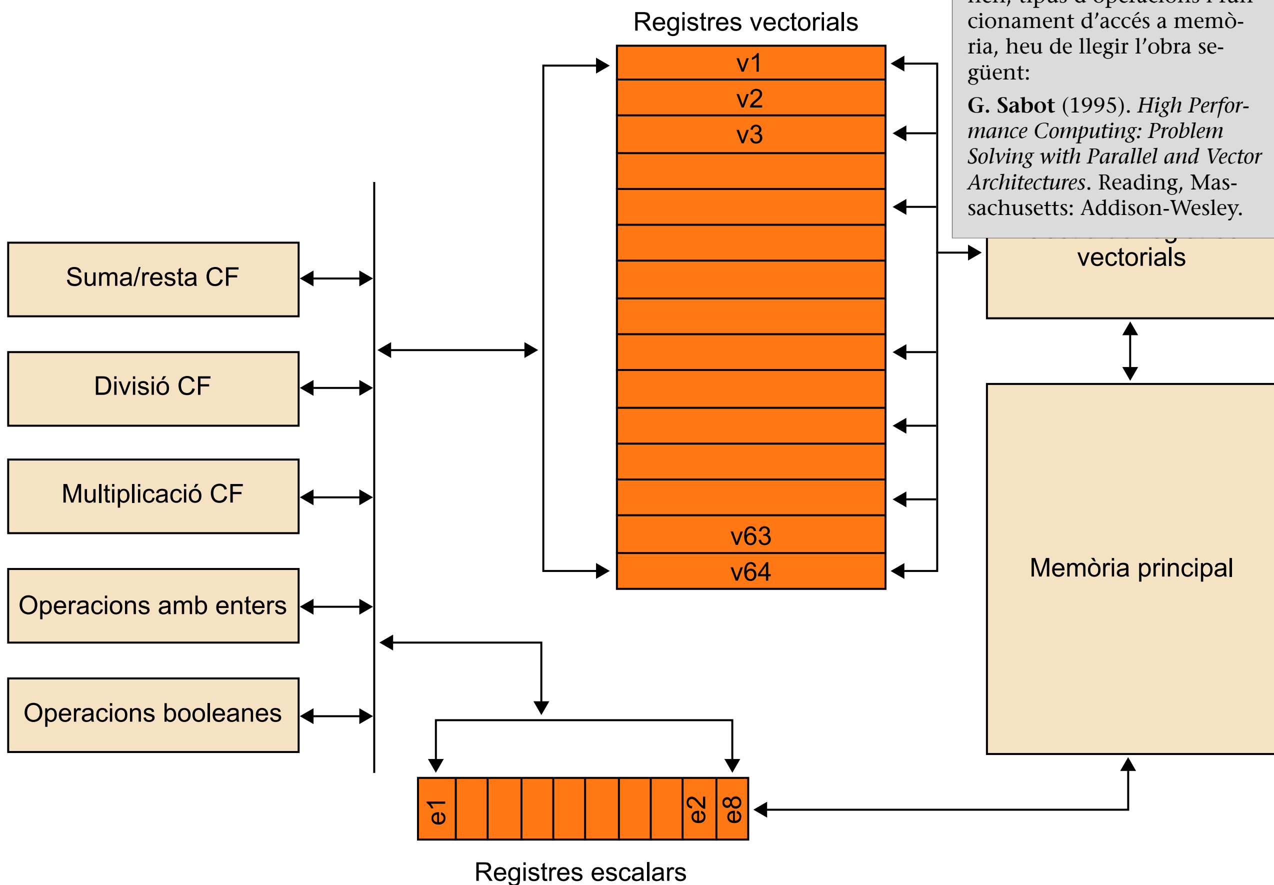
Un exemple en són els *broadcasts* o els *shuffles*. Alguns dels jocs d'instruccions més coneguts són les *advanced vector extensions* (AVX; I. Corp, 2011) o les *streaming SIMD extensions* (SSE; I. Corp, 2007).

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



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.

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.

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.

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.

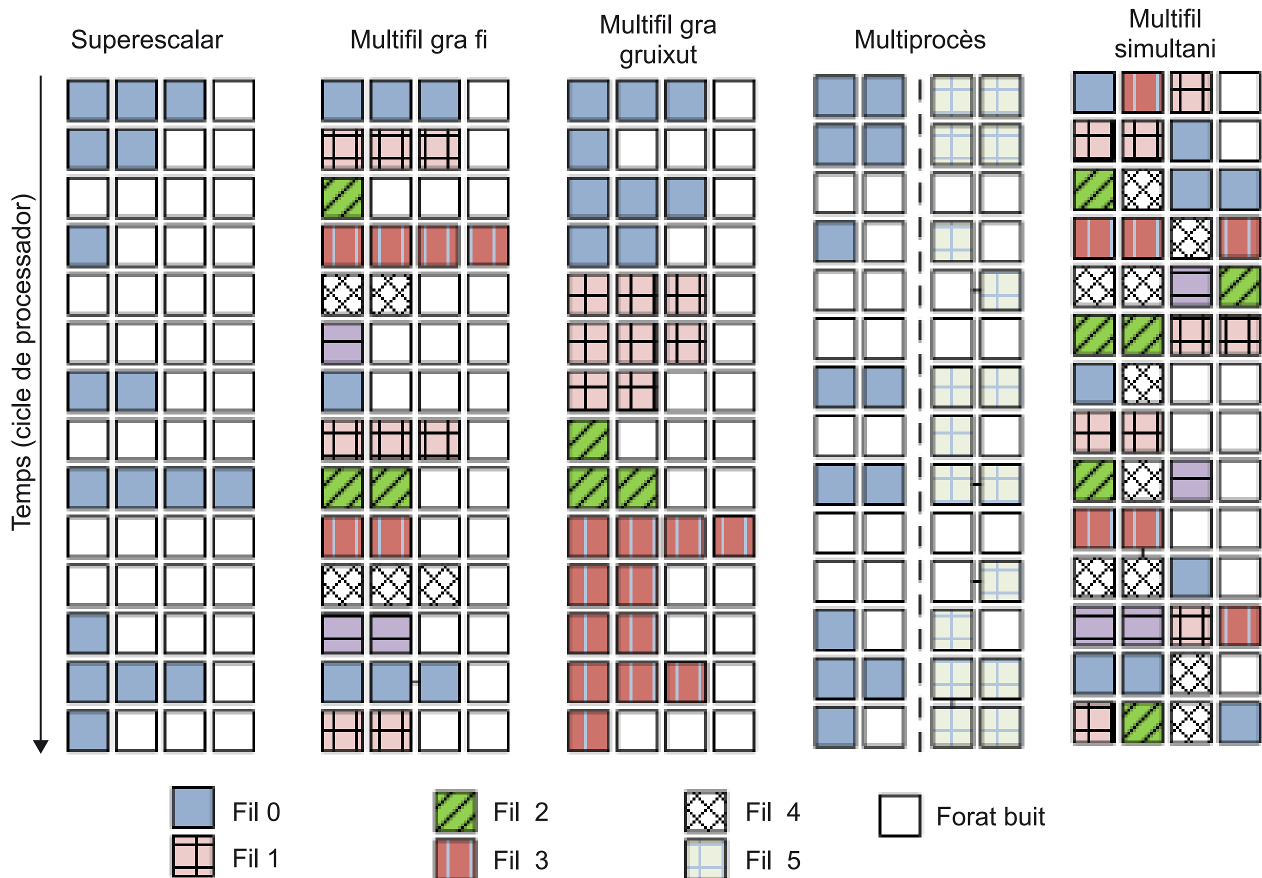
Aplicacions paral·leles

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

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 *superthreading*. 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

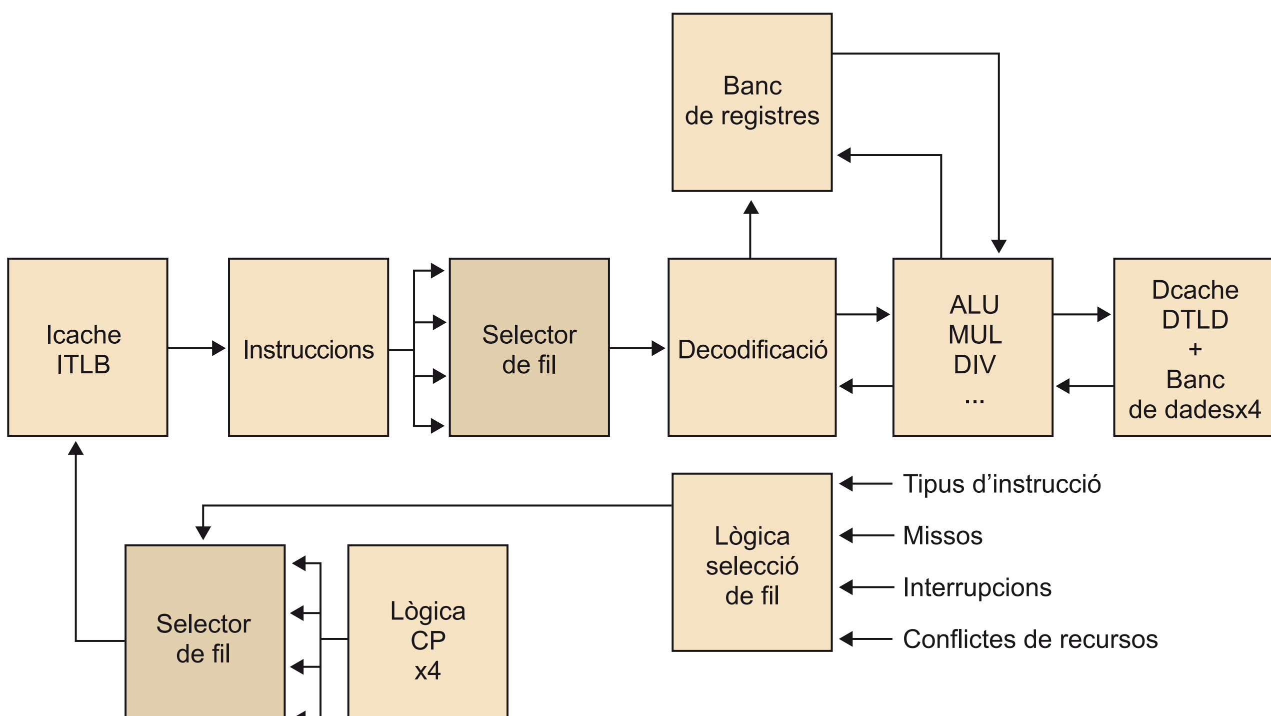
L'UltraSPARCT1 (Kongetira, Aingaran i Olukotun, 2005) és un exemple d'aquest tipus de processador.

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*.

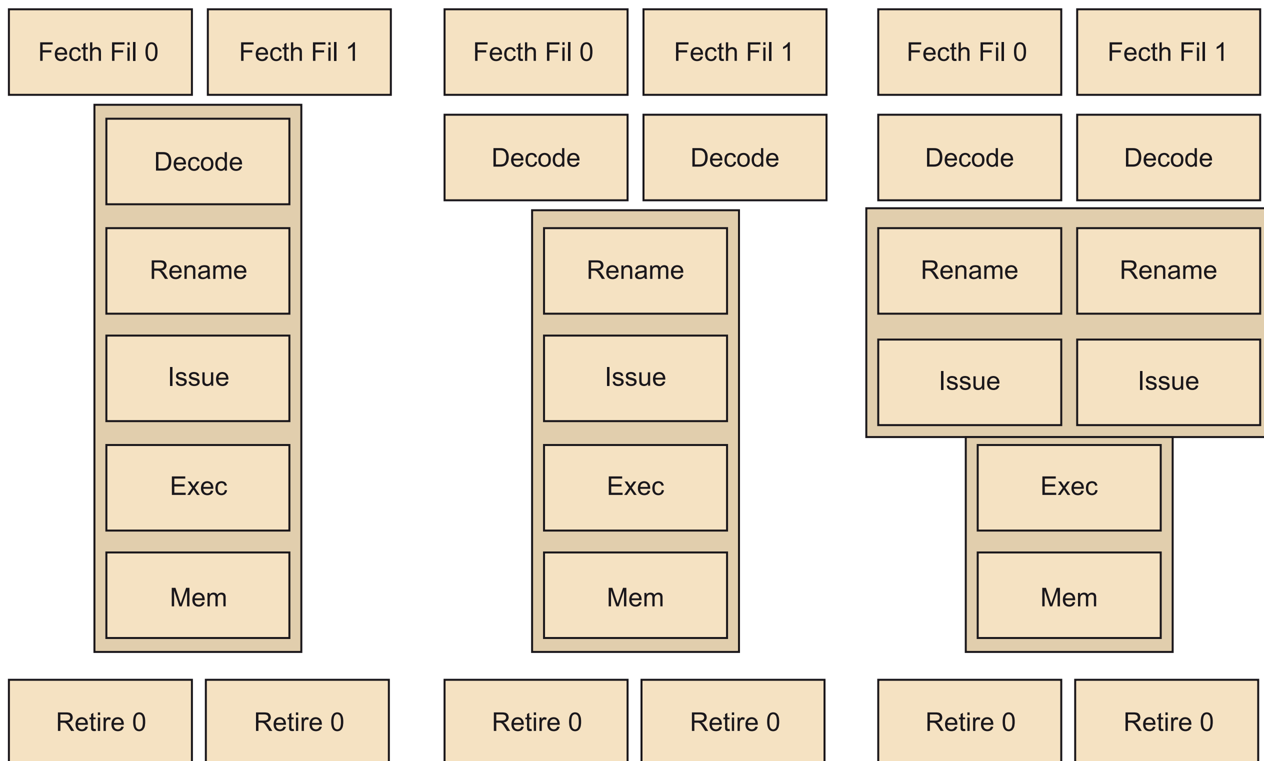
4.4. Disseny d'un SMT

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. Implementacions comercials de l'SMT

4.6.1. L'HyperThreading d'Intel

HyperThreading va ser el nom comercial amb el qual l'empresa Intel va introduir al mercat les tecnologies SMT en els seus productes. El novembre d'aquest any Intel va llançar el processador IntelPentium 4. Aquest va ser el primer que va incloure SMT en el seu disseny.

El 2010, Intel va llançar el processador Atom, que també inclou la tecnologia SMT. No obstant això, aquest producte està orientat al mercat dels dispositius de baix consum. Això inclou portàtils de baix consum, tauletes, telèfons mòbils, etc. Per aquest motiu, tot i ser una tecnologia SMT aquest no és fora d'ordre. Per tant, no inclou tècniques de reordenament d'instruccions, execució especulativa o *renaming* de registres.

Dins de la gamma dels productes Xeon (Intel) orientada a servidors també es pot trobar aquesta tecnologia. En aquest cas, com que el segment de mercat destinat són els servidors, els processadors ofereixen un nivell de paral·lelisme molt més elevat i totes les funcionalitats d'un processador fora d'ordre. En aquest cas, ens movem del processador Bloomfield, amb 4 nuclis i un total de 8 fils d'execució, fins al processador Beckton, amb 8 nuclis i un total de 16 fils d'execució. Aquí cal fer notar que la tecnologia SMT s'inclou dins de cada nucli.

Com es pot observar, la tecnologia SMT és aplicable a molts tipus de segments: segment de dispositius mòbils o segments de computació d'altes prestacions. El concepte de base en tots els casos és el mateix, el de potenciar el rendiment dels sistemes augmentant el nivell de paral·lelisme de les aplicacions. En uns casos es necessita un rendiment molt elevat (per exemple, per servidor de bases de dades) i en altres només poder tenir diferents fils executant-se en paral·lel (per exemple, el navegador i el gestor de correu). El consum i complexitat en tots dos casos també és força diferent.

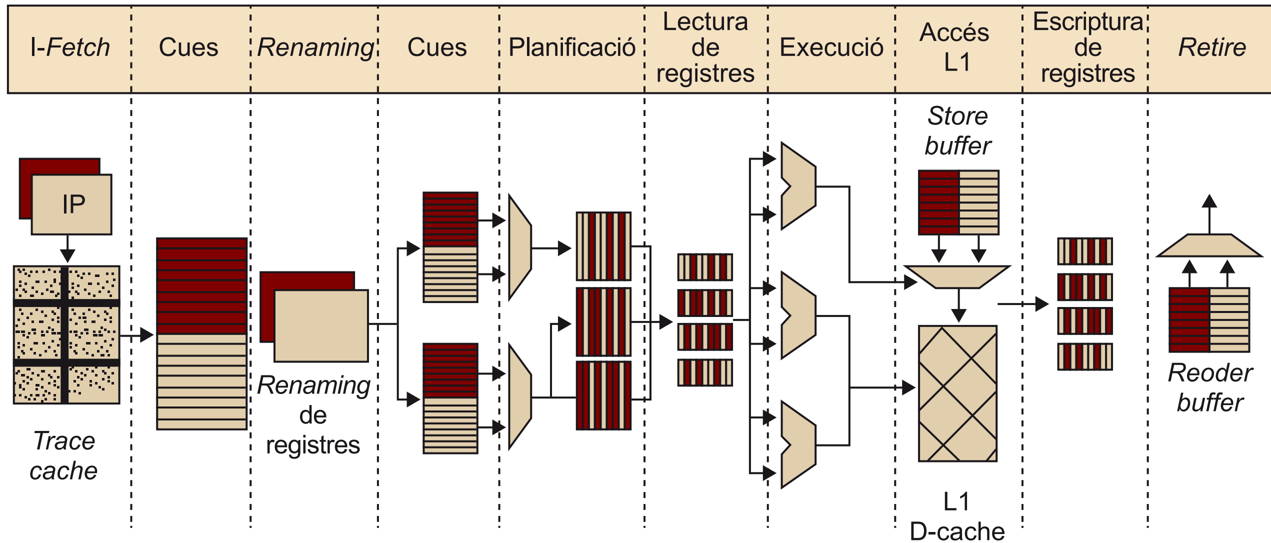
Consum

Un processador Atom pot consumir uns 10 watts, mentre que un processador Beckton pot consumir fins a 130 watts.

L'arquitectura

La figura següent presenta les diferents etapes de la microarquitectura del Pentium IV, coneguda també com a *Netburst Micro-Architecture* (Intel, Sandy Bride Intel). El *pipeline* consta de deu etapes diferents. Les quatre primeres són en ordre i són les relacionades amb la cerca i preparació de les instruccions (cerca, *renaming* i traducció a microoperacions). Les cinc següents poden ser fora d'ordre i són les encarregades de dur a terme l'execució funcional de les operacions. Finalment, la darrera s'executa en ordre i és on les instruccions es finalitzen.

Figura 12. *Pipeline* del Pentium IV



Durant les diferents etapes del processador, hi ha recursos que es troben replicats per processador lògic, n'hi ha que es troben compartits però dividits per processador lògic i, finalment, n'hi ha que es troben totalment compartits.

En primera instància, cada processador lògic manté una còpia separada del seu estat arquitectònic necessari per a una execució funcional correcta. Els recursos encarregats de desar aquest tipus d'informació són els que es troben replicats per a cada context:

- El punter a la instrucció següent.

- El *buffer* d'instruccions del fil, és a dir, el flux d'instruccions que el fil potencialment executarà.

- El *Translation Look-Aside buffer* (TLB) usat per a fer la traducció d'adreces virtuals a físiques. Cada fil ha de tenir un mapatge d'adreça virtual a una adreça física diferent. Altrament hi hauria problemes de seguretat potencials.

- El predictor d'adreça de retorn[13], que, com el nom indica, prediu les adreces de retorn de les funcions.

- L'*advanced programmble interrupt controller* (APIC) orientat a la gestió d'interrupcions.

- La taula de *renaming*, necessària per a poder dur a terme el fora d'ordre i poder fer el remapatge de registres virtuals a registres físics.

(13)En anglès, *return stack predrictor*.

Tot i que hi ha altres recursos que es troben replicats, els més importants són els que acabem d'esmentar. També cal fer notar que aquesta descripció equival a una arquitectura *hyperthreading* general. Cada implementació concreta, per exemple la del processador Atom, pot tenir variacions segons el tipus de requisits que el processador té i del disseny que els arquitectes han dut a terme.

En segona instància, trobem certs recursos que es troben dividits entre els diferents fils. En general, la majoria són *buffers*.

Buffers

Com, per exemple, el *reorder buffer*, els *load* i *store buffers*, cues entre les diferents etapes, etc.

Finalment, trobem altres recursos que es troben totalment compartits entre tots els fils del processador:

- La *trace cache* o memòria cau d'instruccions (i els *buffers* corresponents). Aquest és un mecanisme que es va dissenyar per tal d'augmentar l'amplada de banda de l'etapa de cerca d'instruccions i reduir el consum del processador. Consisteix a desar traces d'instruccions que ja s'han seleccionat prèviament i de les quals ja es té una descodificació.

- Les memòries cau.

- Els mecanismes d'execució fora d'ordre.

- Els predictors de salt.

- Lògica de control i busos de comunicació.

Un dels punts clau en un disseny SMT és la complexitat i àrea que s'afegeix al disseny d'aquest pel fet de considerar els diferents fils d'execució. Un dels avantatges de l'HyperThreading és la capacitat de proporcionar una millora de rendiment important respecte d'un augment d'àrea reduït. Un dels motius principals és que els processadors lògics comparteixen gairebé tots els recursos del processador físic. L'augment d'àrea ve bàsicament pels estats arquitectònics extres, lògica de control addicional i la replicació d'alguns dels recursos.

La perspectiva del sistema

L'HyperThreading, tal com hem dit, feia que un sol processador es veiés des del punt de vista del sistema com si fossin diferents processadors lògics (o fils d'execució). El processador té una còpia de l'estat arquitectònic per a cadascun dels processadors lògics, i tots els processadors lògics comparteixen un conjunt de recursos físics que permeten executar els diferents fluxos d'instruccions.

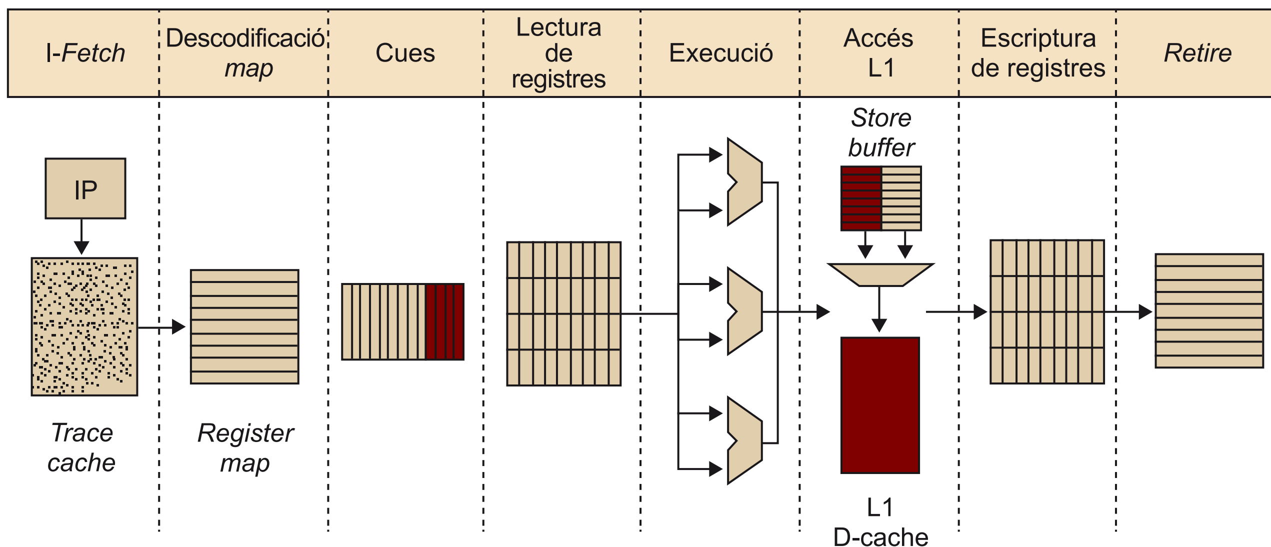
Des de la perspectiva del sistema operatiu i de les aplicacions, el maquinari els ofereix *N* processadors independents. El programari pot gestionar els diferents fils d'execució damunt dels processadors lògics segons les seves polítiques d'administració. És important fer notar que, des del seu punt de vista, és equivalent a tenir a la seva disposició *N* processadors físics independents.

Des de la perspectiva del model de programació, l'HyperThreading es pot veure com una arquitectura multiprocessador amb temps d'accés uniforme a memòria (conegut com a *NUMA*). Tots els fils tenen, de mitjana, un temps d'accés a memòria similar.

4.6.2. L'Alpha 21464

L'Alpha 21464 (Rusu, Tam, Muljono, Ayers i Chang, 2006), també conegut com a *EV8*, va ser l'evolució de l'Alpha 21364 (figura 13). La millora més important de l'EV8 respecte d'aquest darrer va ser que incorporava tècniques SMT. Tot i que tenia un rendiment estimat força més elevat que el seu predecessor, la línia de processador Alpha es va cancel·lar el 2001. Això va incloure l'EV8.

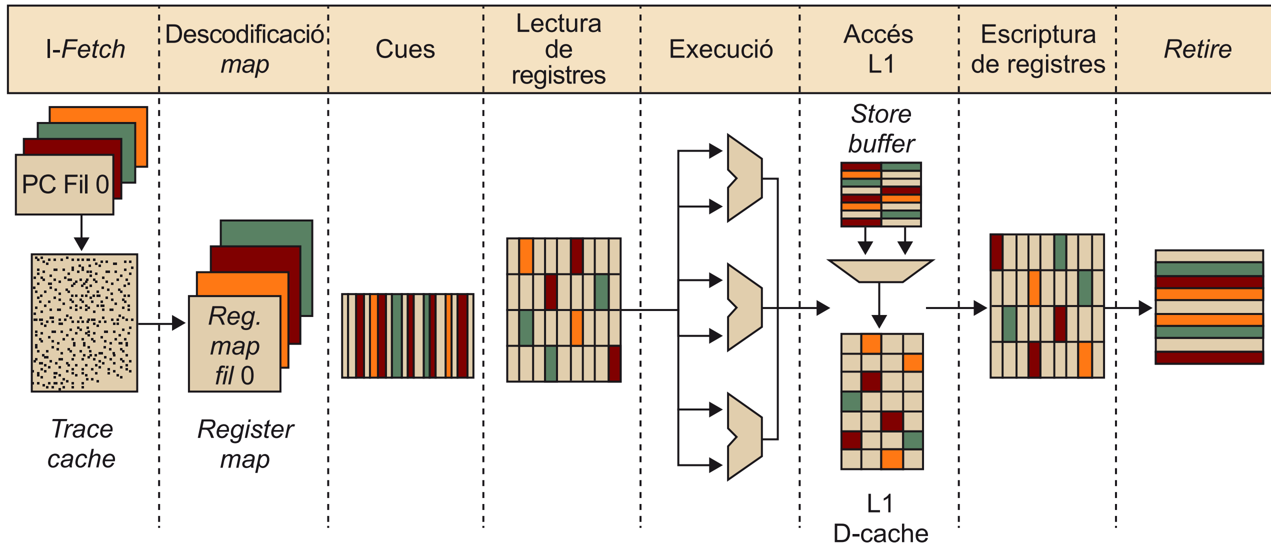
Figura 13. *Pipeline* de l'Alpha 21364



L'arquitectura

La figura 14 mostra les diferents etapes de què es componia. Per tal d'afegir SMT a l'arquitectura Alpha, l'EV8 inclou una lògica nova per a mantenir quatre comptadors de programes independents (un per fil). A cada cicle, l'etapa de *fetch* selecciona un sol dels fils disponibles de la memòria cau d'instruccions. Un cop la instrucció és seleccionada i carregada, es marca amb l'identificador del fil al qual pertany. Aquest identificador va aparellat amb la instrucció durant totes les etapes del processador (etapa de descodificació, unitats funcionals, etc.).

Figura 14. *Pipeline* de l'Alpha 21464



La lògica que gestiona la selecció fora d'ordre de les instruccions que poden ser executades pot escollir una instrucció de qualsevol dels quatre fils. De manera similar al que ja hem vist amb l'HyperThreading, la complexitat afegida a la lògica fora d'ordre i la lògica de *renaming* és força reduïda. En el cas de l'EV8, això implicava només un increment del 6 % de l'àrea.

El canvi més important introduït en les arquitectures Alpha per a donar suport a l'SMT va ser que incorporaven 32 registres enters i 32 coma flotants per a cadascun dels quatre fils d'execució. Per tant, per a desar tot l'estat arquitectònic de tot el processador calen 256 registres. D'altra banda, per tal de poder dur a terme el *renaming* i de poder suportar un total de 256 instruccions en vol, l'EV8 disposa de 256 registres extra. En total inclou un total de 512 registres.

A diferència de la microarquitectura NetBurst, l'Alpha 21364 només replica el comptador de programa i els *registermap*. Tota la resta de recursos es troben compartits:

- La cua d'instruccions.
- El *Register File*, que, com ja s'ha esmentat, es troba incrementat notòriament respecte de l'Alpha 21364.
- El primer i segon nivell de memòries cau.
- El *translation look-aside buffer* (TLB).
- El predictor de salts.

Des del punt de vista del sistema o l'aplicació, l'HyperThreading ofereix quatre fils d'execució totalment independents. Com ja s'ha dit, cada fil es veu com un processador lògic diferent (quatre fils es veuen com quatre processadors independents).

La perspectiva del sistema

En el cas de l'EV8, el sistema també té accés a un sol processador físic. No obstant això, en aquest cas també hi ha un sol processador lògic. Cada fil és vist com una unitat de procés[14], que comparteix recursos com ara la memòria cau d'instruccions, memòria cau de dades, la TLB o la memòria cau de segon nivell amb els altres fils.

(14)En anglès, *thread processing unit* (TPU).

4.7. Arquitectures multinucli

4.7.1. 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.

4.7.2. 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.

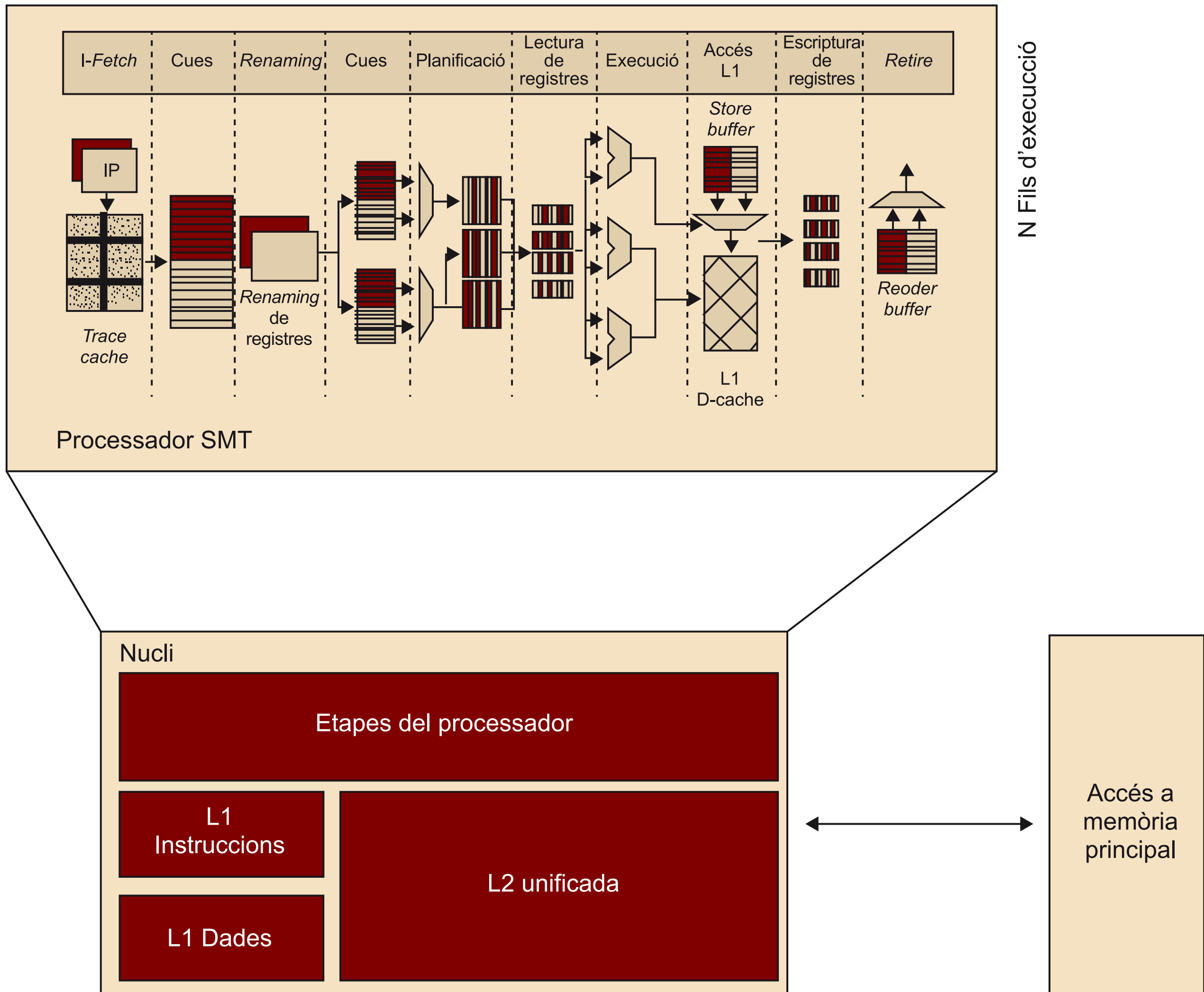
Per exemple, en el cas de la família SandyBridge, es poden trobar els processadors domèstics amb 12 fils i 130 watts de consum, processadors per a dispositius mòbils de 8 fils i 55 watts de consum i processadors per a servidors amb 16 fils i 150 watts de consum. Val a dir que no només varia el nombre 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.

4.7.3. 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.

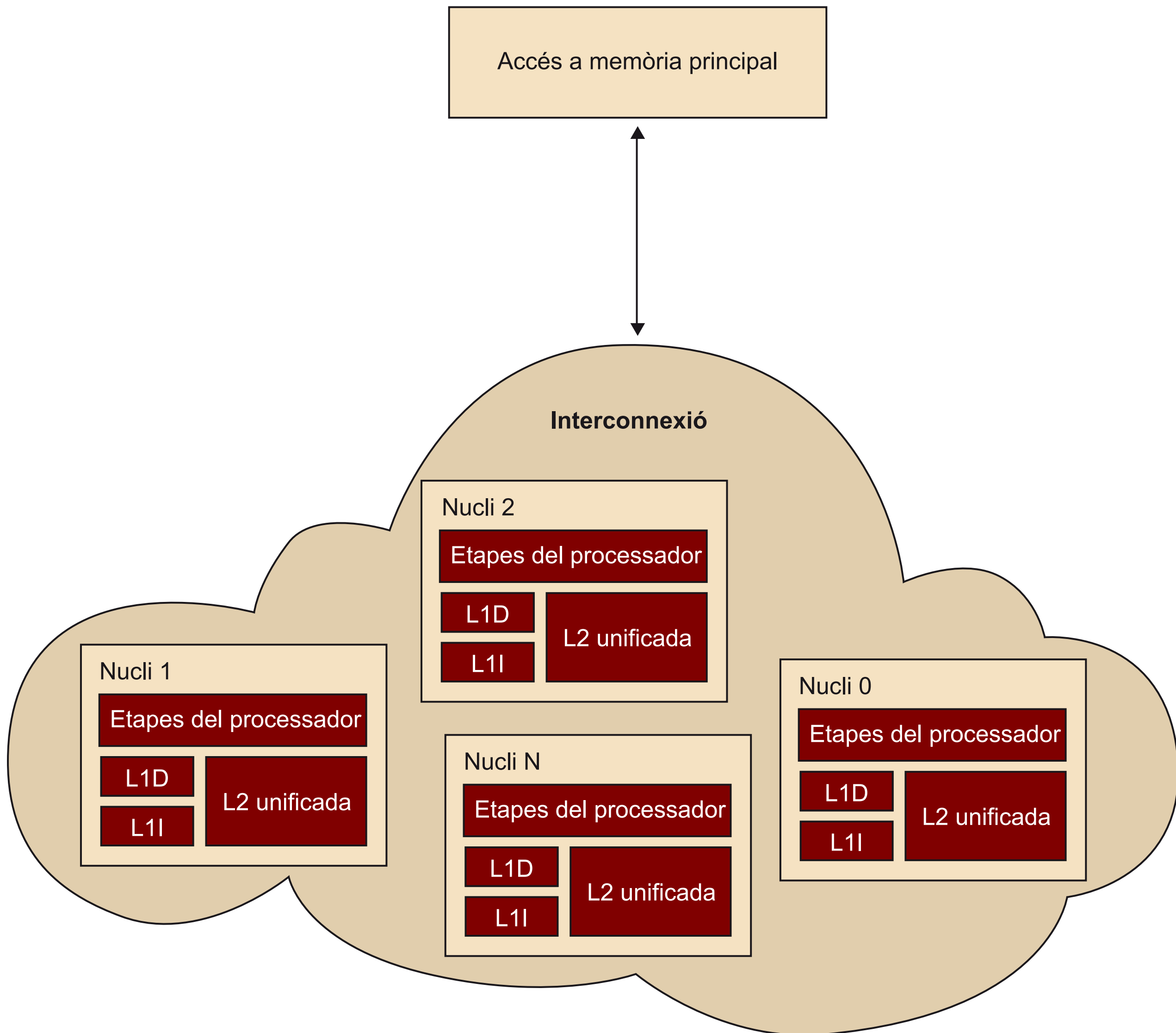
Figura 15. Abstracció de processador multifil



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 16. 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.

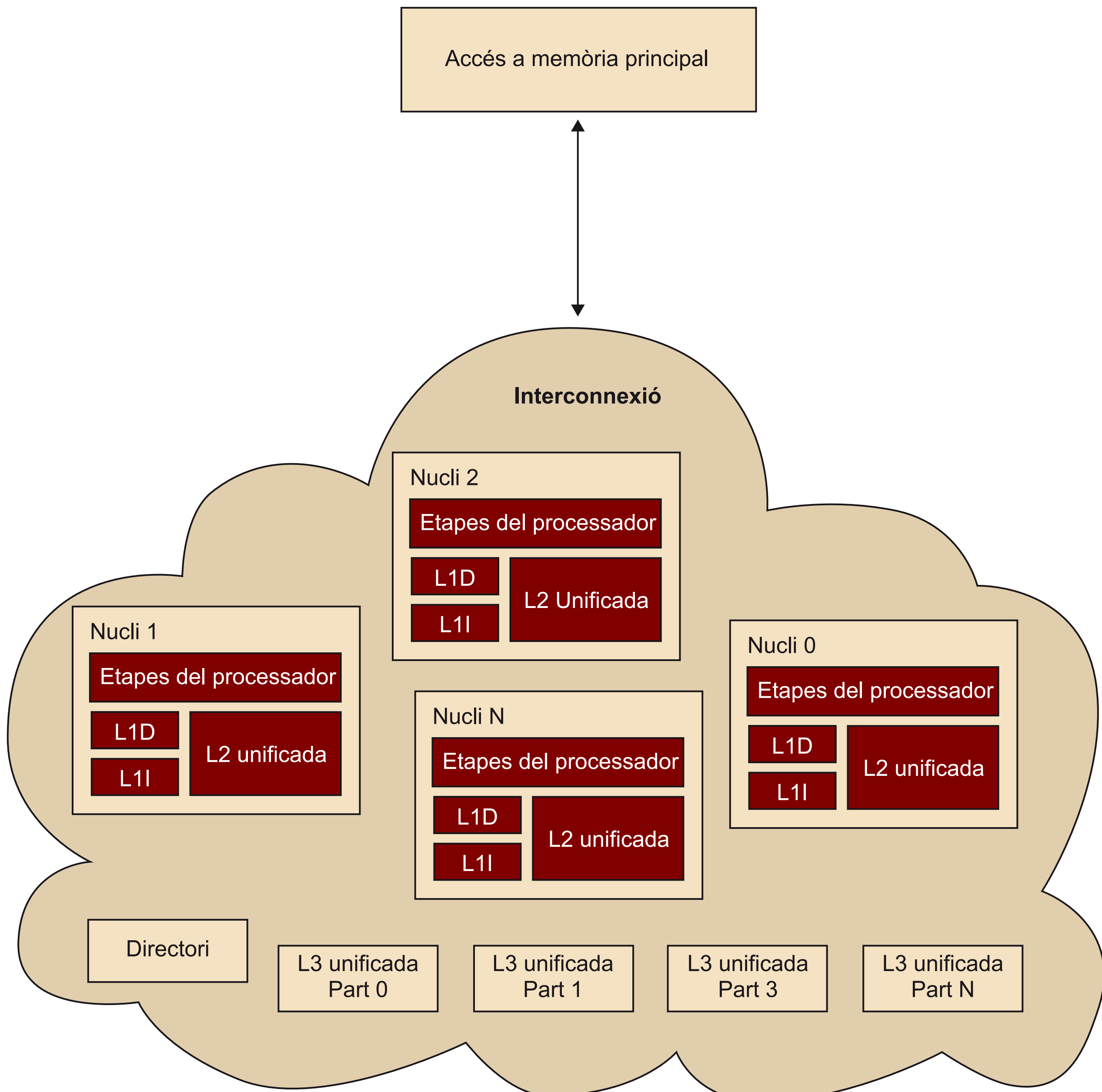
4.7.4. Coherència entre nuclis

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 17. 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 acadèmic, s'han proposat moltes maneres diferents de connectar els elements o components d'un processador multinucli. La majoria d'aquestes propostes vénen de recerca ja realitzada en l'entorn de computació d'altes prestacions[15].

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

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).

4.7.5. 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.

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. Factors determinants en el rendiment en arquitectures modernes

En el darrer apartat s'han presentat diferents arquitectures de processadors que implementen més d'un fil d'execució, com també les arquitectures vectorials. Així doncs, hi ha arquitectures que faciliten dos fils d'execució, com les *shared multithreading*, i n'hi ha que donen accés a desenes de fils d'execució, com les arquitectures multinucli.

En general, una arquitectura és més complexa com més fils facilita. Les arquitectures multinucli són força escalables i poden donar accés a un nombre elevat de fils. Ara bé, com ja s'ha vist, per tal de dur-ho a terme cal dissenyar sistemes que són més complexos. D'altra banda, la complexitat d'aquesta també es veu incrementada si la formen components MIMD i SIMD. En aquest cas, el model de programació ha de tenir mecanismes per a poder explotar, per exemple, el joc d'instruccions vectorials.

Com més complexa és l'arquitectura, més factors cal tenir en compte a l'hora de programar-la. Si tenim una aplicació que té una zona paral·lela que en un processador seqüencial es pot executar en temps x, és d'esperar que en una màquina multifil amb m fils disponibles, aquesta aplicació potencialment ho faci en un temps de x/m. No obstant això, hi ha certs factors inherents al tipus d'arquitectura sobre la qual s'executa i al model de programació emprat que poden limitar aquest increment de rendiment; per exemple, l'accés a les dades compartides per fils que s'estan executant en diferents nuclis.

Per tal d'obtenir el màxim rendiment de les arquitectures sobre les quals s'executen les aplicacions paral·leles, s'han de considerar les característiques d'aquestes arquitectures. Per tant, cal considerar la jerarquia de memòria del sistema, el tipus d'interconnexió sobre el qual s'envien dades, l'amplada de banda de memòria, etc. És a dir, si es vol extreure el màxim rendiment cal redissenyar o adaptar els algorismes a les característiques del maquinari que hi ha per sota.

Si bé és cert que cal adaptar les aplicacions segons les arquitectures en què es volen executar, també és cert que hi ha utilitats que permeten no haver de tenir en compte algunes de les complexitats d'aquestes arquitectures. La majoria apareixen en forma de models de programació i biblioteques que poden ser emprades per les aplicacions.

Aquest apartat presenta els factors més importants que poden limitar l'accés al paral·lelisme que dóna una arquitectura lligats al model de programació.

5.1. Factors importants per a la llei d'Amdahal en arquitectures multifil

La llei d'Amdahal estableix que una aplicació dividida en una part inherentment seqüencial (és a dir, només pot ser executada per un fil) i una part paral·lela P, potencialment pot tenir una millora de rendiment de S (en anglès *speedup*) augmentant el paral·lelisme de l'aplicació a *P*.

$$T' = T * \frac{1}{\left(1 - P + \frac{P}{S}\right)}$$

Ara bé, per a arribar al màxim teòric cal considerar les restriccions inherents al model de programació i restriccions inherents a l'arquitectura sobre la qual s'està executant l'aplicació.

El primer conjunt de restriccions fa referència als límits vinculats a l'algorisme paral·lel considerat, com també a les tècniques de programació emprades. Un cas en què apareixen aquestes restriccions és quan s'ordena un vector, ja que hi ha limitacions causades per l'eficiència de l'algorisme (per exemple, Radix Sort) i per a implementar-lo (per exemple, la manera d'accedir a les variables compartides, etc.).

El segon conjunt fa referència a límits lligats a les característiques del processador sobre el qual s'està executant l'aplicació. Factors com ara la jerarquia de memòria cau, el tipus de memòria cau o el tipus de xarxa d'interconnexió dels nuclis poden limitar aquest rendiment.

Exemple

Pot causar moltes ineficiències que una variable sigui compartida entre dos fils que no són dins el mateix nucli.

Els propers dos subapartats presenten alguns dels factors més importants d'aquests dos blocs que acabem d'esmentar. Del primer conjunt no s'estudien algorismes paral·lels (Gibbons i Rytte, 1988), ja que no és l'objectiu de la unitat, sinó els mecanismes que fan servir aquests algorismes per tal d'implementar tasques paral·leles i tots els factors que cal considerar. Del segon bloc es discuteixen les característiques més rellevants de l'arquitectura que cal considerar en el desenvolupament d'aquest tipus d'aplicacions.

És important remarcar que els factors que s'estudien a continuació són una part dels molts que s'han identificat durant les darreres dècades. A causa de la importància d'aquest àmbit, s'ha fet molta recerca centrada a millorar el rendiment d'aquestes arquitectures i el disseny de les aplicacions que s'hi executen (com les aplicacions de càlcul numèric o les de càlcul del genoma humà).

Lectures recomanades

Per tal d'aprofundir més en les problemàtiques i estudis fets tant en l'àmbit acadèmic com empresarial, és molt recomanable estendre la lectura d'aquesta unitat didàctica amb les referències bibliogràfiques facilitades.

5.2. Factors vinculats al model de programació

5.2.1. Definició i creació de les tasques paral·leles

En el disseny d'algorismes paral·lels es consideren dos tipus de paral·lelisme: a nivell de dades i a nivell de funció. El primer defineix quines parts de l'algorisme s'executen de manera concurrent, i el segon, com es processen les dades de manera paral·lela.

1) En la creació del **paral·lelisme a nivell de funció** és important considerar que les tasques que treballin amb les mateixes funcions i dades tinguin localitat al nucli en què s'executaran. D'aquesta manera els fluxos del mateix nucli comparteixen les entrades corresponents a la memòria cau de dades, com també les seves instruccions.

2) El **paral·lelisme a nivell de dades** també ha de considerar la localitat esmentada, però a més a més ha de tenir en compte la mida de les memòries cau de què disposa. És a dir, el flux d'instruccions ha de treballar amb les dades de manera local a la L1 tant com sigui possible i la resta intentar mantenir-la a nivells de memòria cau tan propers com sigui possible (L2 o L3). D'altra banda, també cal evitar efectes ping-pong entre els diferents nuclis del processador. Això pot succeir quan fils de diferents nuclis accedeixin als mateixos blocs d'adreces de manera paral·lela.

La figura següent mostra un exemple d'escenari que s'ha d'evitar en la creació de fils, tant en l'assignació de tasques com en les dades. En aquest escenari, els dos fils número 0 de tots dos nuclis estan executant la mateixa funció $F(x)$. Aquesta funció conté un bucle intern que fa alguns càlculs sobre el vector k i el resultat l'afegeix a la variable global k. Durant l'execució d'aquests fils s'observa el següent:

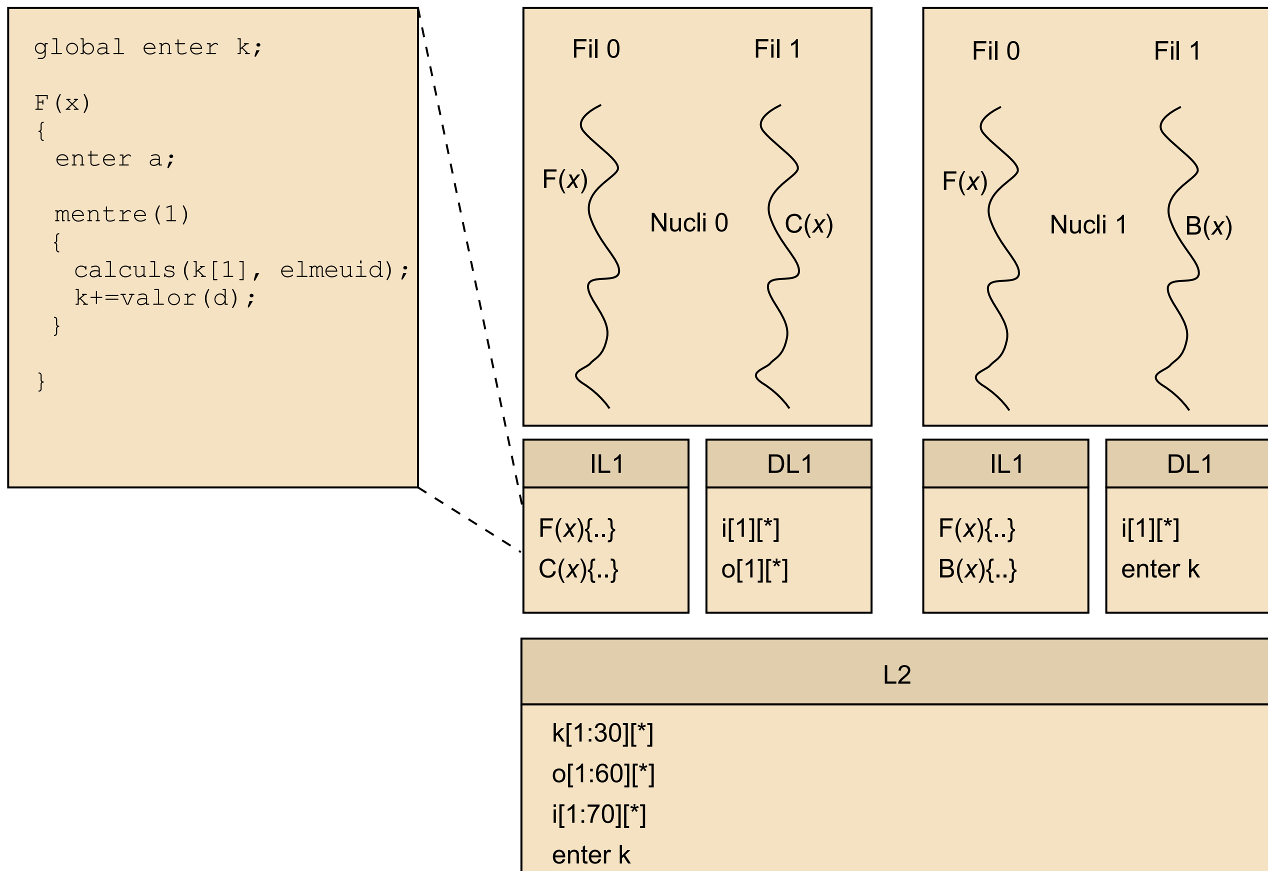
- Els dos nuclis cada vegada que vulguin accedir a la variable global k han d'invalidar la L1 de dades de l'altre nucli. Com ja s'ha vist, depenent del protocol de coherència, pot ser extremadament costós. Per tant, aquest aspecte pot fer baixar el rendiment de l'aplicació.

- Els dos fils accedeixen potencialment a les dades del vector l. Així doncs, les L1 de dades de tots dos nuclis tenen emmagatzemades les mateixes dades (assumint que la línia es troba en estat compartit). En aquest cas seria més eficient que els dos fils s'executessin en el mateix nucli per tal de compartir les dades de la L1 de dades (és a dir, més localitat). Això permetria usar més eficientment l'espai total del processador.

- De manera similar, la L1 d'instruccions de tots dos casos nuclis tenen una còpia de les instruccions del mateix codi que executen els dos fils (*F*). Semblantment al punt anterior, aquest aspecte provoca una utilització inefici-

ent dels recursos, ja que les mateixes dades es troben replicades en totes dues memòries cau. Cal remarcar que, en aquest cas, els dos nuclis no invaliden les línies compartides, ja que en generar les entrades de memòria d'instruccions, es troben en estat compartit i aquestes no s'acostumen a modificar.

Figura 18. Definició i creació de fils



L'exemple anterior mostra una situació força senzilla de solucionar. Ara bé, la definició, la creació de tasques i l'assignació de dades no és una tasca fàcil. Com ja s'ha esmentat, cal considerar la jerarquia de memòria, la manera en què els processos s'assignen als nuclis, el tipus de coherència que el processador facilita, etc.

Tot i la complexitat d'aquesta tasca hi ha molts recursos que ajuden a definir aquest paral·lelisme, com els següents:

- Aplicacions que permeten la paral·lelització automàtica d'aplicacions seqüencials. Molts compiladors inclouen opcions per tal de generar codi paral·lel automàticament. Ara bé, és fàcil trobar-se en situacions en què el codi generat no és òptim o no té en compte alguns dels aspectes introduïts.

- Aplicacions que donen assistència en la paral·lelització dels codis seqüencials. Per exemple, ParaWise (ParaWise, 2011) és un entorn que guia

Lectures recomanades

Per tal d'aprofundir en aquest àmbit és recomanable llegir:

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

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

l'usuari en la paral·lelització de codi Fortran. En qualsevol cas el resultat pot ser similar a la paral·lelització automàtica.

- Finalment, també hi ha biblioteques que proporcionen interfícies per a implementar algorismes paral·lels. Aquestes interfícies acostumen a ser la solució més eficaç per a treure el màxim rendiment de les aplicacions: faciliten l'accés a funcionalitats que permeten definir el paral·lelisme a nivell de dades i funcions, afinitats de fils a nuclis, afinitats de dades a la jerarquia de memòria, etc. Algunes d'aquestes biblioteques són OpenMP, Cilck (Intel, Intel Cilk Plus, 2011), TBB (Reinders, 2007), etc.

5.2.2. Mapatge de tasques a fils

Una tasca és una unitat de concurrència d'una aplicació, és a dir, inclou un conjunt d'instruccions que poden ser executades de manera concurrent (paral·lela o no) a les instruccions d'altres tasques. Un fil és una abstracció del sistema operatiu que permet executar paral·lelament diferents fluxos d'execució. Una aplicació pot estar formada des d'un nombre relativament petit de tasques fins a milers.

Tasques

Exemples clars són els servidors de videojocs o els servidors de pàgines web: poden tenir milers de tasques atenent les peticions dels usuaris.

No obstant això, el sistema operatiu no pot donar accés a un nombre tan elevat de fils d'execució per la limitació dels recursos. D'una banda, la gestió d'un nombre tan elevat és altament costós (molts canvis de contextos, gestió de moltes interrupcions, etc.). D'altra banda, tot i que potencialment el sistema operatiu pugui donar accés a un miler de fils, el processador sobre el qual s'executen les aplicacions dóna accés a pocs fils físics[16]. Per tant, cal anar fent canvis de context entre tots els fils disponibles a nivell de sistema i els fils que el maquinari faciliti. Aquest és el motiu pel qual el nombre de fils del sistema ha de ser configurat coherentment amb el nombre de fils del processador.

(16)En anglès anomenats *hardware threads*.

L'aplicació, per tal de treure el màxim rendiment del processador, ha de definir un mapatge eficient i adequat de les tasques que vol executar als diferents fils de què disposa. Alguns dels algorismes de mapatge de tasques a fils més emprats són els següents:

- Patró *master/slave*: un fil s'encarrega de distribuir les tasques que resten per executar als fils esclaus que no estiguin executant res. El nombre de fils esclaus és variable.

- Patró *pipeline* o cadena: en què cadascun dels fils fa una operació específica en les dades que s'estan processant, alhora que facilita el resultat al fil següent de la cadena.

- Patró *task pool*: en què hi ha una cua de tasques pendents a ser executades i cadascun dels fils disponibles agafa una d'aquestes tasques pendents quan acaba de processar l'actual.

Aquest mapatge es pot fer basant-se en molts criteris diferents. No obstant això, els aspectes introduïts al llarg d'aquesta unitat didàctica s'haurien de considerar en arquitectures multifil heterogènies. Depenent de com s'assignin les tasques als fils i com estiguin assignats els fils als nuclis, el rendiment de les aplicacions varia molt.

Lectura recomanada

Un treball de recerca molt interessant sobre les tècniques de mapatge és el presentat per Philbin i altres. Els autors presenten tècniques de mapatge i gestió de fils per tal de mantenir la localitat a les diferents memòries cau:

J. Philbin; J. Edler; O. J. Anshus; C. Douglas; K. Li (1996). "Thread scheduling for cache locality". *Seventh international conference on Architectural support for programming languages and operating systems*.

Exemple de mapatge de tasques

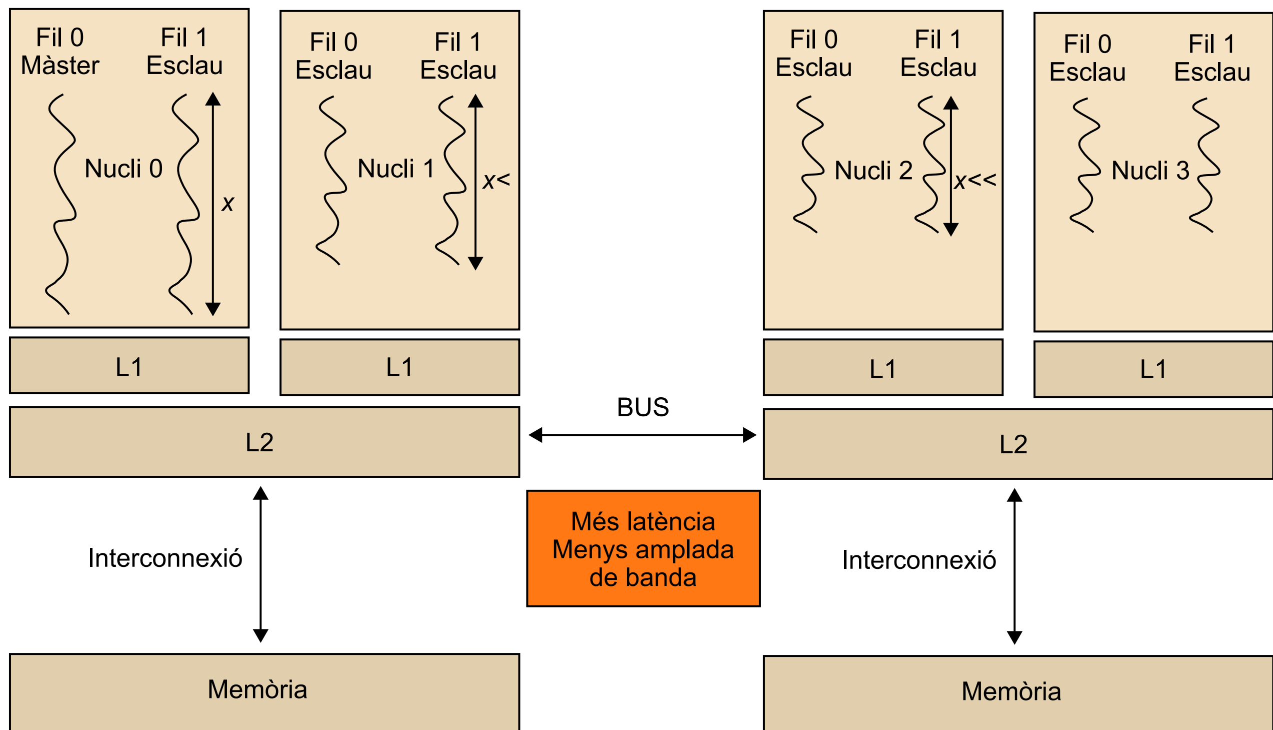
La figura 19 presenta un exemple d'aquesta situació. Una aplicació multifil s'executa sobre una arquitectura composta per dos processadors. Cadascun amb accés a memòria i tots dos connectats per un bus. Els fils que s'estan executant al nucli 2 i al nucli 3 accedeixen a les dades i a la informació que el màster els facilita.

Respecte d'una xarxa d'interconnexió local, el bus acostuma a tenir una latència més elevada i menys amplada de banda. Aquest és el motiu pel qual els fils que s'executen al nucli 2 i al nucli 3 tenen cert desbalanceig respecte dels que ho fan al nucli 0 i al nucli 1. Aquests factors cal considerar-los a l'hora de decidir com assignar les tasques i la feina a cadascun dels fils.

En l'exemple considerat el rendiment del processador i de l'aplicació és força menor del que potencialment es podria assolir. En l'instant de temps *T*, el fil esclau del nucli 1 ha acabat de fer la feina *X*. Els fils 0 i 1 del nucli 1 tarden un cert temps (*T'*) més a finalitzar la seva tasca. I els fils dels nuclis 2 i 3 tarden un temps (*T''*)força major a *T* fins a acabar. Per tant, els nuclis 0 i 1 no es fan servir durant *T* – *T''* i *T'* – *T''* respectivament.

No només la utilització del processador és més baixa, sinó que aquest desbalanceig fa finalitzar l'aplicació més tard. En aquest cas, a l'hora de dissenyar el repartiment de tasques cal tenir en compte en quina arquitectura s'executa l'aplicació, com també quins són els elements que potencialment poden causar desbalancejos i quins fils poden fer més feina per unitat de temps.

Figura 19. Patró *master/slave* en un multinucli amb dos processadors



5.2.3. Definició i implementació de mecanismes de sincronització

Sovint les aplicacions multifil usen mecanismes per a sincronitzar les tasques que els diferents fils estan duent a terme. Un exemple el trobem en la figura de l'exemple anterior (figura 19), en què el fil màster s'espera que els esclaus acabin mitjançant funcions d'espera.

En general s'acostumen a fer servir tres tipus de mecanismes diferents de sincronització:

- L'ús de variables per a controlar l'accés a determinades parts de l'aplicació o a determinades dades de l'aplicació (com un comptador global). Exemples d'aquest tipus de variables ho són les dels semàfors o les d'exclusió mútua.

- L'ús de barreres per tal de controlar la sincronització entre els diferents fils d'execució. Aquestes ens permeten assegurar que tots els fils no passen d'un determinat punt fins que tots hi han arribat.

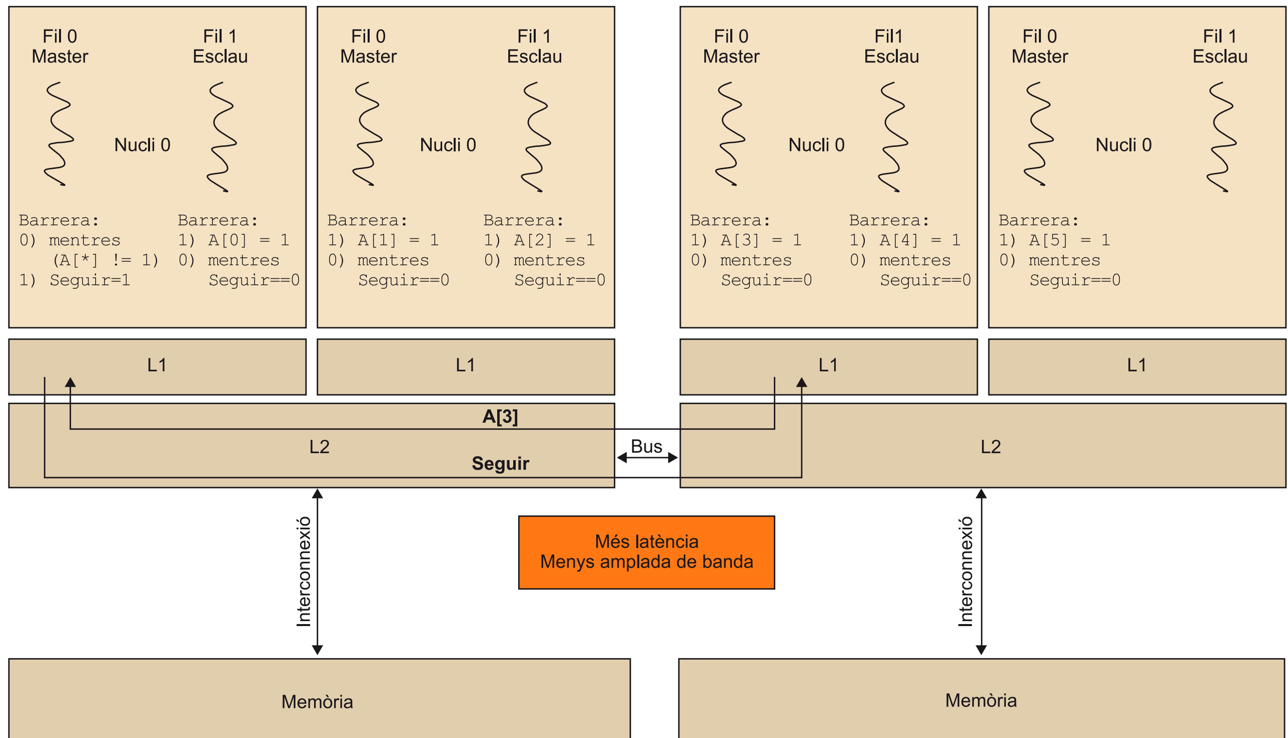
- L'ús de mecanismes de creació i espera de fils. Com a l'exemple anterior, el fil màster espera la finalització dels diferents fils esclaus mitjançant crides a funcions d'espera.

Des del punt de vista d'arquitectures multifil/nucli, el segon i el tercer punt són menys intrusius al rendiment de l'aplicació (Villa, Palermo i Silvano, 2008). Com es veurà a continuació, les barreres són mecanismes que s'empren en només certes parts de l'aplicació i que es poden implementar de manera eficient dins d'un multifil/nucli. En canvi, un ús excessiu de variables de control pot provocar un descens significatiu en el rendiment de les aplicacions.

Barreres en arquitectures multinucli

La figura 20 presenta un exemple de possible implementació de barrera i com es comportaria en un multinucli. En aquest cas, un fil s'encarrega de controlar el nombre de nuclis que han arribat a la barrera. Cal fer notar que el nucli 0 per tal d'accedir al vector A té totes aquestes dades a la L1, en estat compartit o en exclusiu.

Figura 20. Funcionament d'una barrera en un multinucli



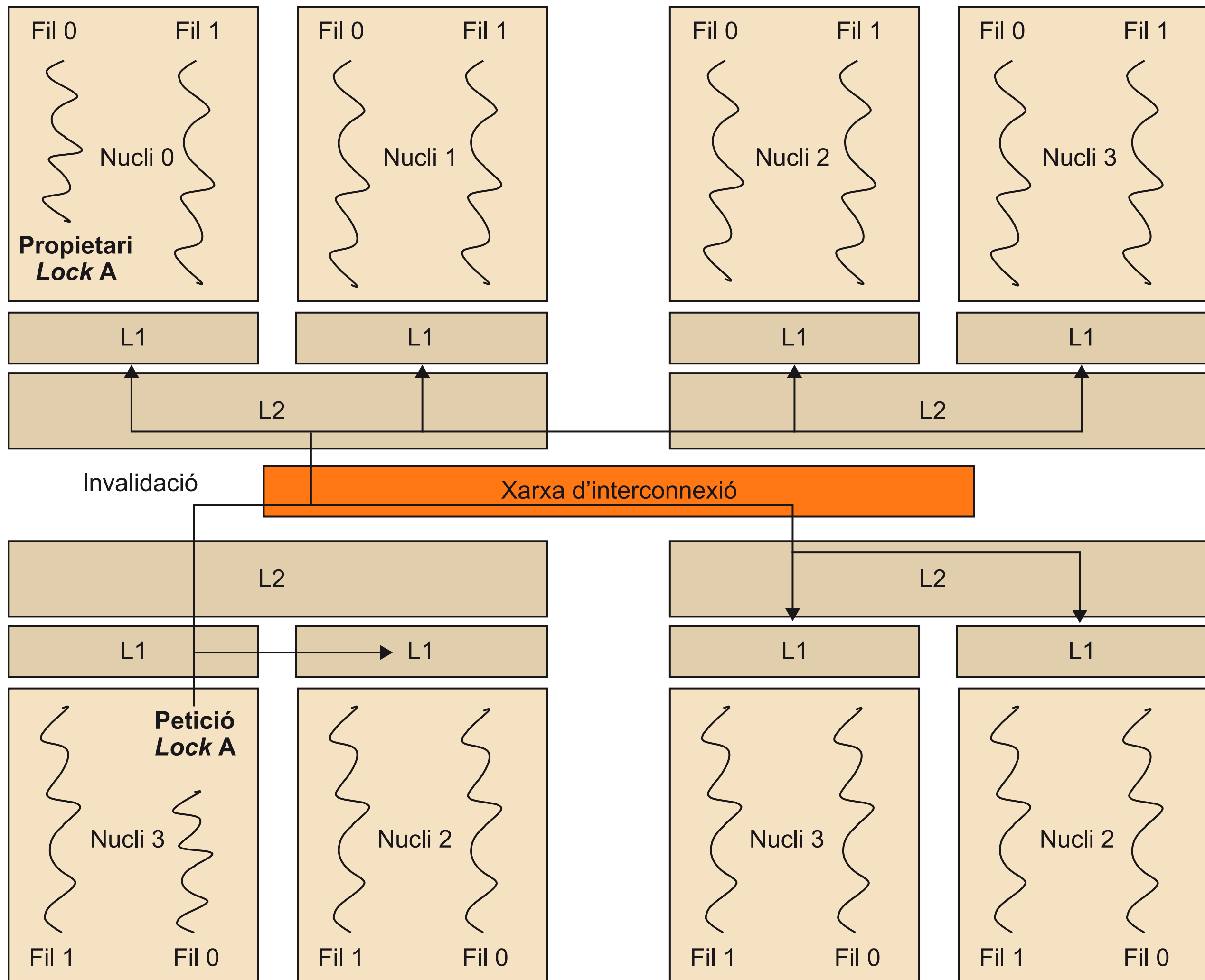
Cada vegada que un fil arriba a la barrera, vol modificar la posició corresponent del vector. Per tant, invalida la línia corresponent (la que conté A[i]) de tots els nuclis i la modifica. Quan el nucli 0 torna a llegir l'adreça d'A[i], ha de demanar el valor al nucli i. Segons el tipus de protocol de coherència que implementi el processador, el nucli 0 invalida la línia del nucli i o bé la mou a estat compartit.

Com es pot deduir d'aquest exemple, el rendiment d'una barrera pot ser més o menys eficient segons la seva implementació i l'arquitectura sobre la qual s'està executant. Així, una implementació en què, en lloc d'un vector, hi ha una variable global que compta els que han acabat seria més ineficient, ja que els diferents nuclis haurien de competir per agafar la línia, invalidar els altres nuclis i escriure'n el valor nou.

Mecanismes d'exclusió mútua en arquitectures multinucli

Aquests mecanismes s'empren per tal de poder accedir de manera exclusiva a certes parts del codi o a certes variables de l'aplicació. Són necessaris per a mantenir l'accés coordinat als recursos compartits i evitar condicions de carreres, *deadlocks* o situacions similars. Alguns d'aquest tipus de recursos són semàfors, *mutex-locks* o *read-writerlocks*.

Figura 21. Adquisició d'una variable d'exclusió mútua



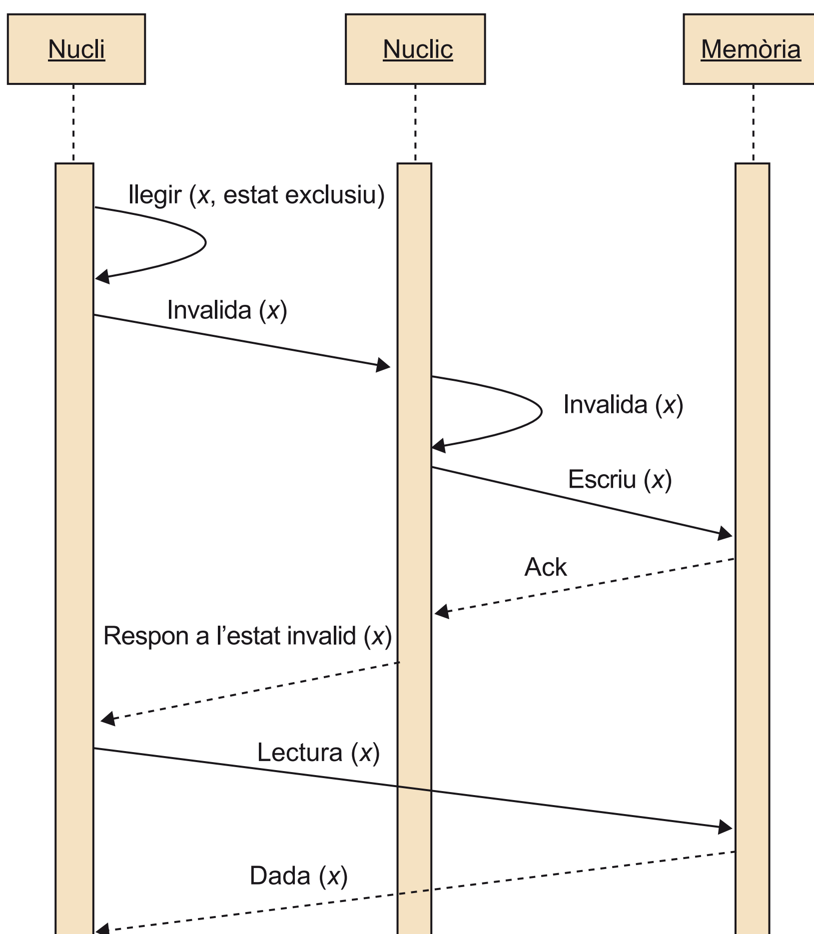
En arquitectures d'un sol nucli, l'accés a aquests tipus d'estructures pot tenir un impacte relativament menor. Amb alta probabilitat tots els fils estan compartint els accessos a les mateixes línies de la L1 que les desen. No obstant això, en un sistema multinucli l'accés concurrent de diferents fils a aquestes estructures pot dur a problemes d'escalabilitat i de rendiments greus. De manera similar al que s'ha mostrat amb les barreres, l'accés a les línies de memòria que contenen les variables emprades per a gestionar l'exclusió mútua impliquen invalidacions i moviments de línies entre els nuclis del processador.

La figura 21 mostra un escenari en què l'ús freqüent d'accés a zones d'exclusió mútua entre els diferents fils pot reduir substancialment el rendiment de l'aplicació. En aquest exemple s'assumeix un protocol de coherència entre els diferents nuclis de tipus *snoop*; per tant, cada vegada que un dels fils d'un nucli vol agafar la propietat de la variable d'exclusió mútua (*lock*) ha d'invalidar tota la resta de nuclis. Cal fer notar que la línia de la memòria cau que conté la variable en qüestió es troba en estat modificat cada cop que el nucli l'actualitzi.

Cada vegada que un fil agafa el *lock*, modifica la variable per tal de marcar-la com a pròpia. En el cas que el fil només l'estigui consultant, no caldria invalidar els altres nuclis.

Es pot assumir que la variable es reenvia del nucli que la té (el nucli 0) al nucli que la demana (el nucli 3) en estat modificat. Ara bé, depenent del tipus de protocol de coherència que implementés el processador, la línia s'escriuria primer a memòria i, llavors, el nucli 3 la podria llegir. En aquest cas el rendiment seria extremadament baix: per cada lectura del *lock x*, s'invalidarien tots els nuclis; així, el que la tingués en estat modificat l'escriuria a memòria i finalment el nucli que l'estigués demanant la llegiria de memòria (figura 22).

Figura 22. Lectura del *lock x* en estat exclusiu



Com s'ha pogut veure, és recomanable un ús moderat d'aquest tipus de mecanismes en arquitectures amb molts nuclis i també en arquitectures heterogènies. Així doncs, a l'hora de decidir quin tipus de mecanismes de sincronització es fan servir, cal considerar l'arquitectura sobre la qual s'executa l'aplicació (jerarquia de memòria, protocols de coherència i interconnexions entre nuclis) i com s'implementen tots aquests mecanismes. D'altra banda, és també recomanable reduir al màxim la quantitat de dades que comparteixen els di-

Lectura recomanada

Per tal d'aprofundir en la creació de mecanismes d'exclusió mútua escalables en arquitectures multinucli es recomana llegir l'article:

M. Chynoweth i M. R. Lee (2009). "Implementing Scalable Atomic Locks for Multi-Core".

ferents nuclis. D'aquesta manera es disminueix la quantitat de missatges de protocol de coherència que s'envien entre nuclis i el nombre d'invalidacions i *snoops* que han de processar els nuclis.

Un dels punts més importants quan es fa el disseny d'arquitectures multinucli eficients, per a aplicacions de supercomputació o HPC, és precisament que facilitin mecanismes per a poder implementar de manera eficient barreres entre tots els fils de les aplicacions. Depenent del nombre de nuclis i dels mecanismes que el processador faciliti una barrera pot tardar des d'un centenar de cicles de processador fins a milers de cicles.

Lectura recomanada

Els autors de l'article següent tracten d'implementacions escalables de mecanismes de sincronització entre fils:

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

5.2.4. Gestió d'accés concurrent a dades

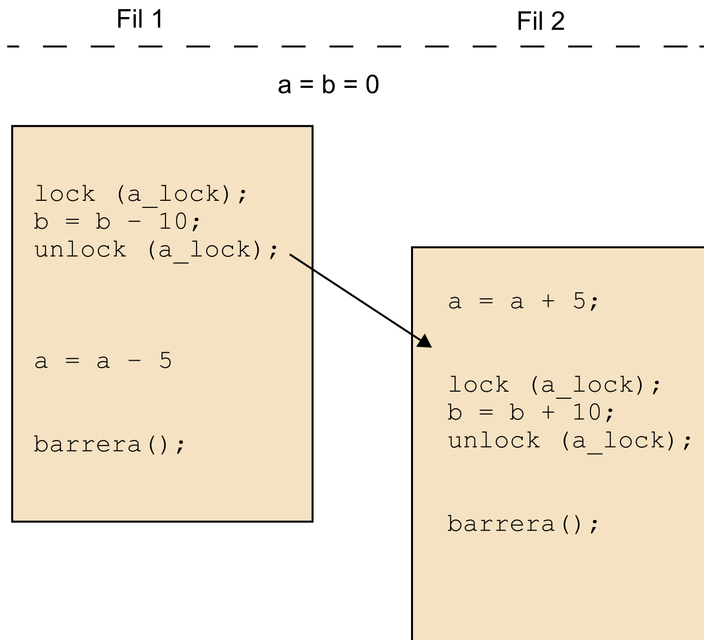
Acabem de veure les implicacions que té usar diferents tècniques d'exclusió mútua en arquitectures multiprocessador. Aquest tipus de tècniques són emprades per tal d'assegurar que l'accés a les dades entre diferents fils és controlat. Si no s'usen aquestes tècniques de manera adequada les aplicacions poden acabar tenint carreres d'accés[17].

(17)En anglès, *race accesses*.

En general es poden distingir dos tipus de carreres d'accés que cal evitar quan es desenvolupa un codi paral·lel:

1) Carreres d'accés a dades: succeeixen quan diferents fils estan accedint de manera paral·lela a les mateixes variables sense cap tipus de protecció. Per tal que hi hagi una carrera d'aquest tipus, un dels accessos ha de ser en forma d'escriptura. La figura 23 mostra un codi que potencialment pot tenir un accés a carrera de dades. Suposant que els dos fils s'estan executant al mateix nucli, quan tots dos fils arriben a la barrera, quin valor té *a*? Com que l'accés a aquesta variable no es troba protegit i s'hi accedeix tant en mode lectura com en escriptura (afegint-hi 5 i –5) aquesta variable pot tenir els valors següents: 0, 5 i –5.

Figura 23. Carrera d'accés a dades



2) Carreres d'accés general: aquest tipus de carreres succeeixen quan dos fils diferents han de seguir un ordre específic en l'execució de diferents parts del seu codi, però no tenim estructures que forcin aquest ordre. L'exemple següent mostra un d'aquest tipus de carreres. Tots dos fils usen una estructura guardada a memòria compartida en què el primer fil posa un treball i el segon el processa. Com es pot observar, si no s'hi afegeix cap tipus de control o variable de control, és possible que el fil 2 s'acabi d'executar sense processar el treball *a*.

Exemple carrera d'accés general

```
Fil 1:
Treball a = nou_treball();
Configurar_Treball(a);
EncuaTreball(a,Cua);
PostProces();
Fil 2:
Treball b  = AgafaTreball(Cua);
si(b != INVALID)
ProcessaTreball(b);
Acaba();
```

Carrera d'accés a dades

Cal esmentar que una carrera d'accés a dades és un tipus específic de carrera d'accés general. Tots dos tipus poden ser evitats emprant els mecanismes d'accés introduïts en l'anterior subapartat (barreres, variables d'exclusió mútua, etc.). Sempre que es dissenyi una aplicació multifil cal considerar que els accessos en mode escriptura i lectura a parts de memòries compartides han d'estar protegits; si els diferents fils assumeixen un ordre específic en l'execució de diferents parts del codi cal forçar-ho via mecanismes de sincronització i espera.

El primer dels dos exemples presentats (figura 23) es pot evitar afegint una variable d'exclusió mútua que controli l'accés a la variable *a*. D'aquesta manera, independentment de qui accedeixi primer a modificar el valor de la variable, el valor d'aquesta un cop arribat a la barrera és 0. El segon dels exemples de

la figura 23 podria ser evitat amb mecanismes d'espera entre fils, és a dir, com mostra l'exemple següent, el segon fil hauria d'esperar que el primer fil notifiqui que li ha facilitat el treball.

Evitar la carrera d'accés mitjançant sincronització

```
Fil 1:
Treball a = nou_treball();
Configurar_Treball(a);
EncuaTreball(a,Cua);
NotificaTreballDisponible();
PostProces();
Fil 2:
EsperaTreball();
Treball b  = AgafaTreball(Cua);
si(b != INVALID)
ProcessaTreball(b);
Acaba();
```

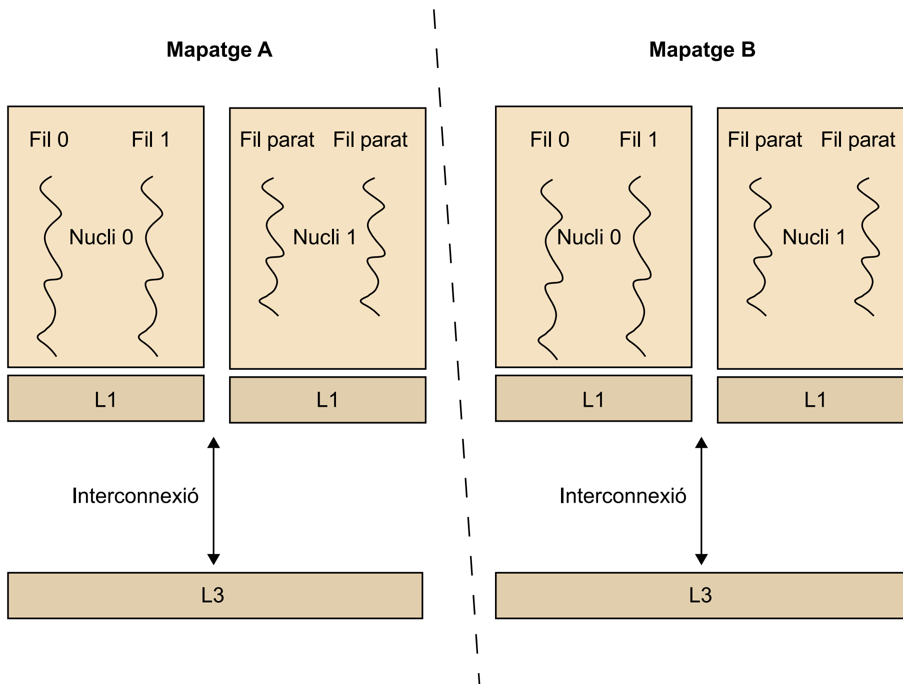
Condicions de carrera en arquitectures multinuclis

En el subapartat anterior s'han introduït diverses situacions en què l'ús de memòria compartida entre diferents fils és inadequat. El primer, les carreres d'accés a dades, esdevé quan dos fils diferents estant llegint i escrivint de manera descontrolada a una zona de memòria. S'ha mostrat com el valor d'una variable pot ser funcionalment incorrecte quan tots dos fils finalitzen els seus fluxos d'instruccions. Ara bé, aquesta cadena d'esdeveniments succeeix independentment de l'arquitectura i del mapatge de fils sobre el qual s'executa l'aplicació?

En aquest subapartat es vol mostrar com el mateix codi es pot comportar de manera diferent dins d'un mateix processador segons com s'assignin els fils als nuclis, ja que, depenent d'aquest aspecte, la condició de carrera analitzada en el subapartat anterior succeeix o no.

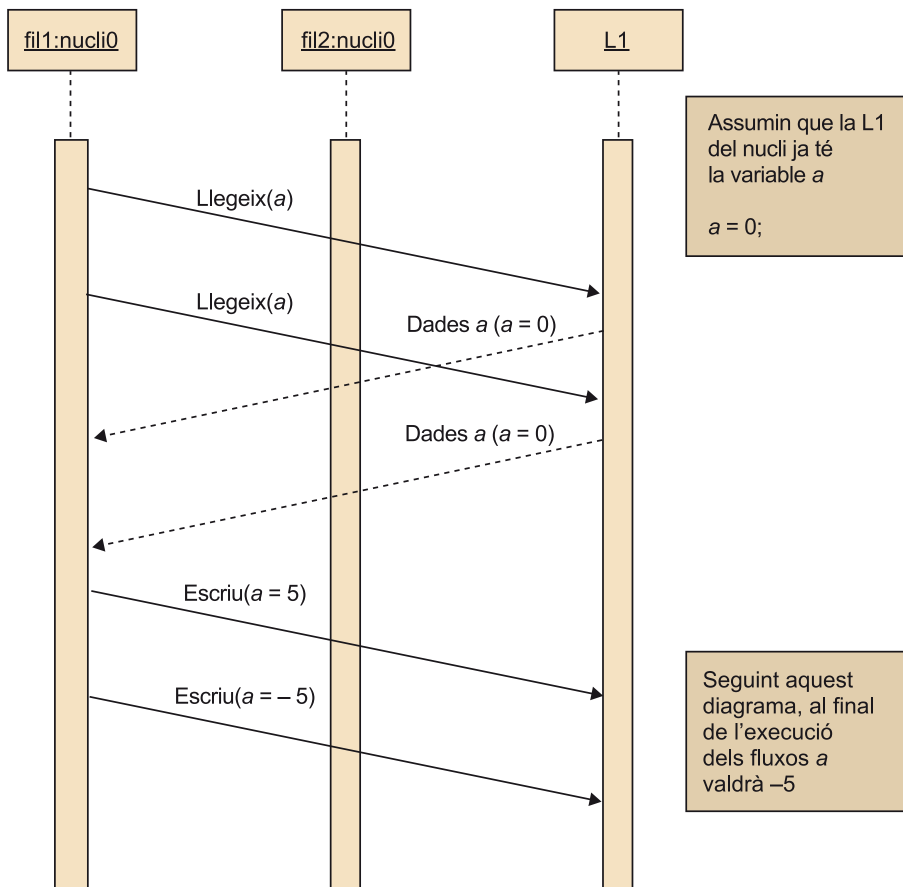
Suposem els dos escenaris que mostra la figura següent:

Figura 24. Dos mapatges de fils diferents executant l'exemple de carrera d'accés general



En el primer dels dos escenaris, els dos fils s'estan executant en el mateix nucli. Tal com mostra la figura 25, els dos fils accedeixen al contingut de la variable *a* de la mateixa memòria cau de nivell dos. Primer el fil 1 en llegeix el valor.

Figura 25. Accés compartit a *a* en un mateix nucli

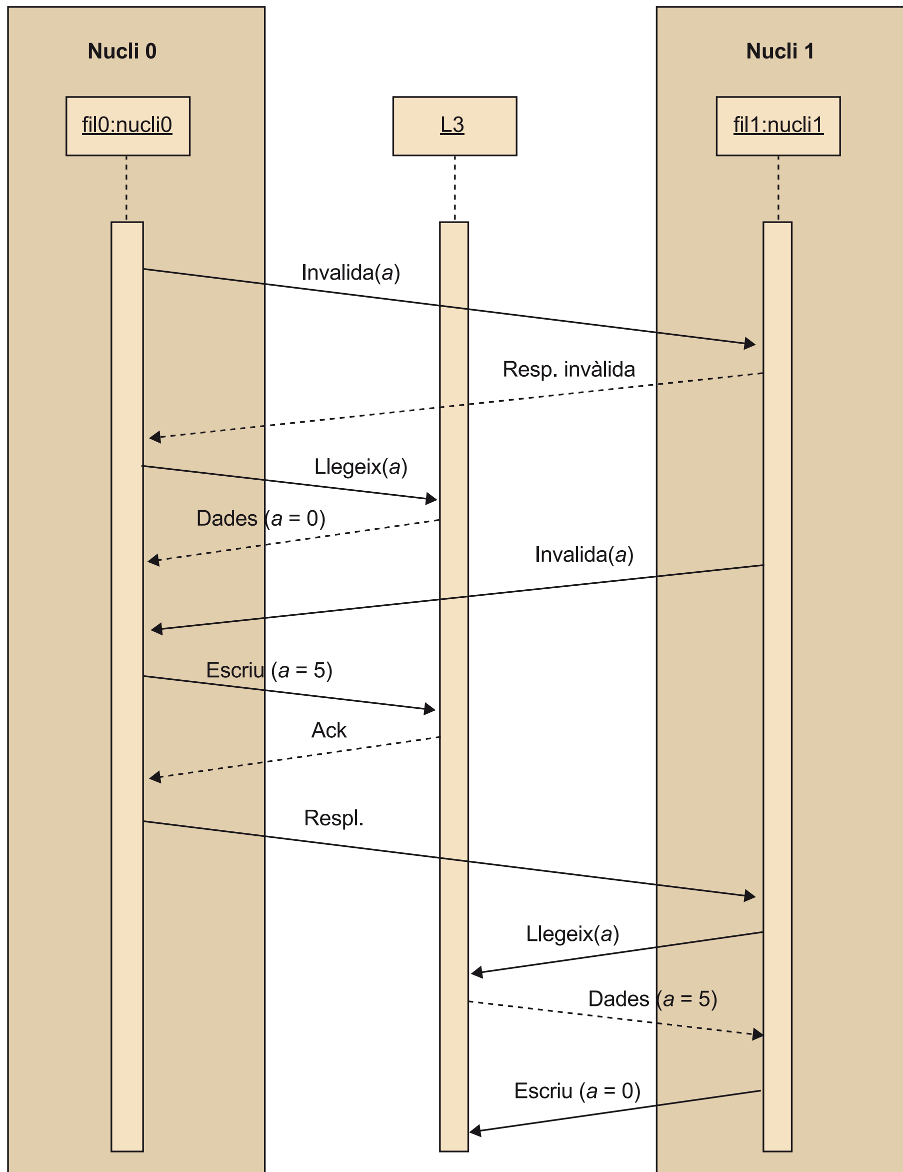


A continuació, tot i que el fil 1 ja està modificant la variable, el segon fil en llegeix el valor original. Finalment, tots dos fils escriuen els valors a memòria. No obstant això, el segon fil sobreescriu el valor actual (5) pel valor resultant de l'operació aritmètica que el fil ha aplicat (–5). Aquest flux d'esdeveniments, com ja s'ha vist, és funcionalment incorrecte, és a dir, el resultat de l'execució dels diferents fils no és l'esperat per l'aplicació en qüestió.

Assumim ara el segon dels escenaris, en què cadascun dels fils s'executa en un nucli diferent. En les arquitectures de processadors amb memòria coherent, si dos nuclis diferents estan accedint en un mateix moment a una mateixa línia de memòria, el protocol de coherència assegura que només un nucli pot modificar el valor d'aquesta línia. Per tant, en un instant de temps tan sols un nucli pot tenir la línia en estat exclusiu per tal de modificar-la.

En aquest escenari nou, i gràcies al protocol de coherència, la carrera d'accés mostrada en el cas anterior no succeeix. Com mostra la figura 26, el protocol de coherència protegeix l'accés en mode exclusiu a la variable *a*. Quan el primer fil vol llegir el valor de la variable en mode exclusiu per a ser modificat, n'invalida totes les còpies dels nuclis del sistema (en aquest cas només un). Un cop el nucli 0 notifica que té la línia en estat invàlid, en demana el valor a la memòria cau de tercer nivell. Per tal de simplificar l'exemple, suposem que modifica el valor de *a* tan aviat com la rep.

Figura 26. Accés compartit a *a* en nuclis diferents



A continuació, el fil 0 que s'està executant en el nucli 1, per tal d'agafar la línia en mode exclusiu, invalida la línia dels altres nuclis del sistema (nucli 0). Quan el nucli 0 en rep la petició d'invalidació, aquest en valida l'estat. Com que es troba en estat modificat, n'escriu el valor a la memòria cau de tercer nivell. Un cop en rep l'*acknowlegement*[18], respon al nucli 1 que la línia es troba en estat invàlid. Arribat a aquest punt, el nucli 1 en demana el contingut a la L3, rep la línia, la modifica i l'escriu de tornada amb el valor correcte.

(18)D'ara en endavant ack.

D'una banda, és important remarcar que si bé en aquest cas no es dóna la carrera d'accés, la implementació paral·lela segueix tenint un problema de sincronització. Depenent de quin tipus de mapatge s'apliqui als fils de l'aplicació es troba l'error ja estudiat.

D'altra banda, cal tenir present que, depenent de quin tipus de protocol de coherència implementi el processador, el comportament de l'aplicació podria variar. Per aquest motiu, emprar les tècniques de sincronització per tal de fer que l'execució sigui determinista i no lligada a factors arquitectònics o de mapatge és realment rellevant.

5.2.5. Altres factors que cal considerar

Aquest subapartat presenta diferents factors que cal considerar a l'hora de dissenyar, desenvolupar i executar aplicacions paral·leles en arquitectures multifil, com també les característiques principals i les implicacions que aquestes aplicacions tenen sobre les arquitectures multifil.

Com ja s'ha esmentat, el disseny d'aplicacions paral·leles és un camp en què s'ha fet molta recerca (tant acadèmica com industrial). És, per tant, aconsellable aprofundir en algunes de les referències facilitades. Altres factors que no s'han esmentat, però que també són importants, són els següents:

- Els *deadlocks*. Succeeixen quan dos fils es bloquegen esperant un recurs que té l'altre. Tots dos fils queden bloquejats per sempre, per tant bloquegen l'aplicació (Kim i Jun, 2009).

- Composició dels fils paral·lels. És a dir, la manera en què s'organitzen els fils d'execució. Aquesta organització depèn del model de programació (com OpenMP o MPI) i de com es programa l'aplicació (per exemple, depèn de si els fils estan gestionats per l'aplicació amb *PosixThreads*).

- Escalabilitat del disseny. Factors com ara el nombre de fils, baixa concurrència en el disseny o bé massa contenció en accessos als mecanismes de sincronització poden reduir substancialment el rendiment de l'aplicació.

Lectura recomanada

Per tal d'aprofundir en la escalabilitat del disseny, es recomana la lectura següent:

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

5.3. Factors lligats a l'arquitectura

Aquest subapartat presenta alguns factors que poden impactar en el rendiment de les aplicacions paral·leles que no estan directament lligades al model de programació. Com es veurà a continuació, alguns d'aquests factors apareixen depenent de les característiques del processador multifil sobre el qual s'executa l'aplicació. Així doncs, tractarem alguns dels factors més importants que cal que tenir en compte quan es desenvolupen aplicacions paral·leles per a arquitectures multifil.

D'una banda, la compartició de recursos entre diferents fils pot portar a situacions de conflictes que degraden molt el rendiment de les aplicacions. Un cas de compartició és el de les mateixes entrades d'una memòria cau.

D'altra banda, el disseny de les arquitectures multifil implica certes restriccions que cal considerar en implementar les aplicacions. Per exemple, un processador multinucli té un sistema de jerarquia de memòria en què els nivells inferiors (per exemple, la L3) es comparteixen entre diferents fils i els superiors es troben separats pel nucli (per exemple, la L1). Les fallades als nivells superiors són força menys costoses que als nivells inferiors. No obstant això, la mida d'aquestes memòries és força menor, per tant, cal adaptar les aplicacions perquè tinguin el màxim de local possible als nivells superiors.

Aquest subapartat està centrat en aspectes lligats als protocols de coherència i gestió de memòria, tot i que hi ha molts altres factors que cal considerar si es vol treure el màxim rendiment de l'algorisme que s'està dissenyant.

Exemple

Optimitzacions del compilador (Lo, Eggers, Levy, Parekh i Tullsen, 1997), característiques de les memòries cau (Hily i Seznec, 1998) o el joc d'instruccions del processador (Kumar, Farkas, Jouppi, Ranganathan i Tullsen, 2003).

5.3.1. Compartició falsa

En moltes situacions es vol que diferents fils de l'aplicació comparteixin zones de memòria concretes. Com hem vist, sovint es donen situacions en què diferents fils comparteixen comptadors o estructures en els quals es desen dades resultants de càlculs fets per cadascun, en què s'usen variables d'exclusió mútua per tal de protegir aquestes zones.

Les dades que els diferents fils estan usant en un instant de temps concret es desen en les diferents memòries cau de la jerarquia (des de la memòria cau de darrer nivell, fins a la memòria cau de primer nivell del nucli que l'està fent servir). Per tant, quan dos fils estan compartint una dada, també comparteixen les mateixes entrades de les diferents memòries cau que desen aquesta dada.

Compartició d'entrades

En la figura 25, tots dos fils accedeixen a la mateixa entrada de la L1 que desa la variable *a*.

Des del punt de vista de rendiment el que es busca és que els accessos dels diferents fils encertin alguna de les memòries cau (com més propera al nucli millor), ja que l'accés a memòria és molt costós. D'això se'n diu mantenir la localitat en els accessos (Grunwald, Zorn i Henderson, 1993).

Ara bé, hi ha situacions en què les mateixes entrades de la memòria cau les comparteixen dades diferents. El càlcul de quina entrada d'una memòria cau es fa servir es defineix segons l'associativitat i el tipus de mapatge de la memòria (Handy, 1998).

Memòria 2-associativa

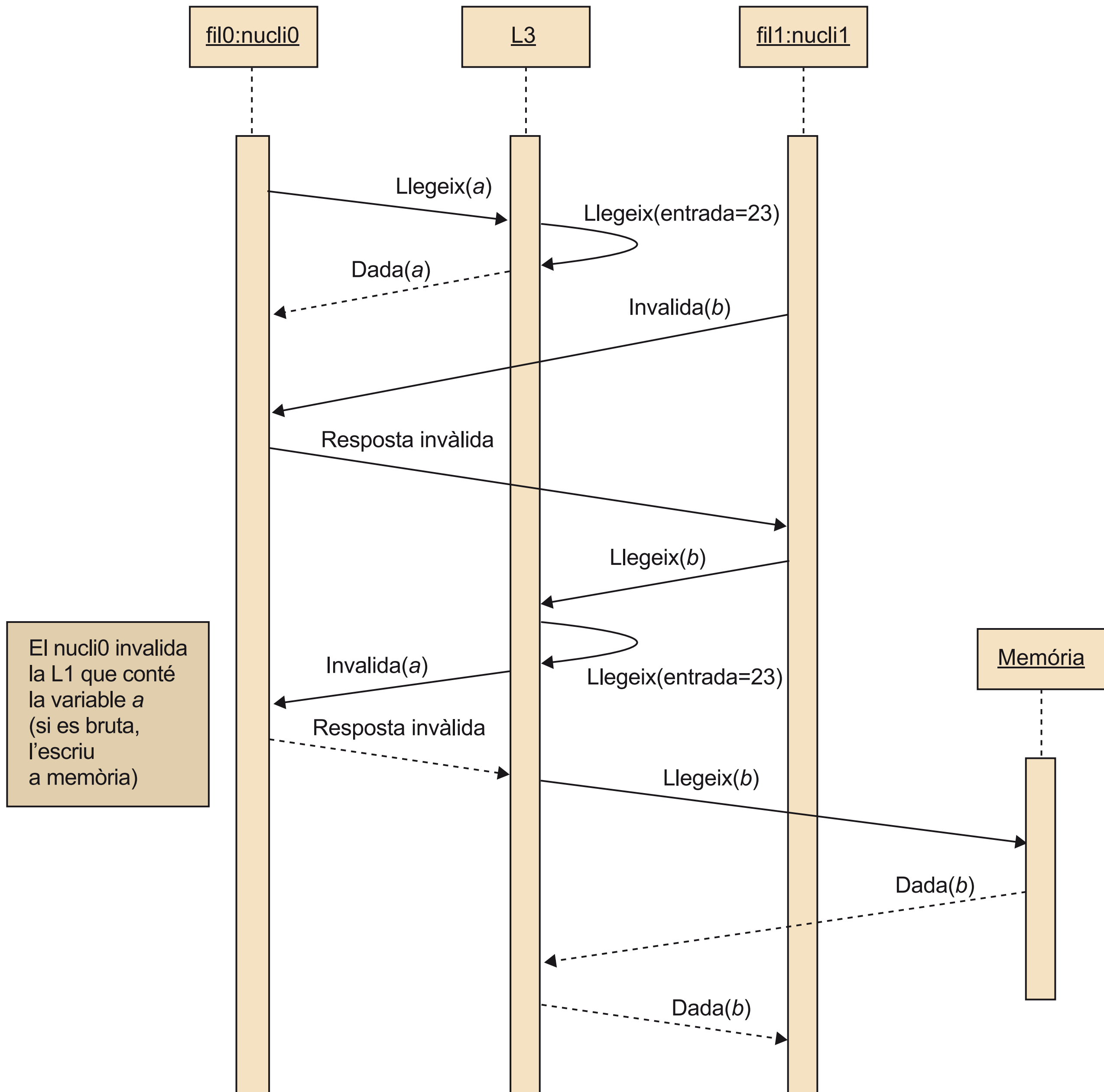
Una memòria 2-associativa (Seznec, 1993) divideix la memòria cau en conjunts de dues entrades. Primer es calcula en quin dels conjunts pertany l'adreça i després s'escull quina de les dues entrades conjunt es fa servir.

Pot passar que dos fils accedeixin a dos blocs de memòria diferents, que es mapegin al mateix conjunt d'una memòria cau. En aquest cas, els accessos d'un fil al seu bloc de memòria invaliden les dades de l'altre fil desades a les mateixes entrades de la memòria. Tot i que les adreces coincideixen en les mateixes entrades de la memòria, les dades i les adreces són diferents. Per tant, per a cada accés s'invaliden les dades de l'altre fil i es demana la dada al següent nivell de la jerarquia de memòria (per exemple, la L3 o la memòria principal). Com es pot deduir, aquest aspecte implica una reducció important en el rendiment de l'aplicació. Això s'anomena *compartició falsa*[19].

(19) En anglès, *false sharing*.

La figura següent mostra un exemple del que podria passar en una arquitectura multinucli en què dos fils de nuclis diferents estan experimentant *falsesharing* al mateix set de la L3. Per a cada accés d'un dels fils s'invalida una línia de l'altre fil, que és desada al mateix conjunt. Com s'observa per a cada accés es generen un nombre important d'accions: invalidació d'adreça a l'altre nucli, lectura a la L3, es victimitza la dada de l'altre fil de l'altre nucli i s'escriu a memòria si és necessari, es llegeix la dada a memòria i s'envia al nucli que l'ha demanada.

Figura 27. Compartició falsa a la L3



En el cas que tots dos fils no tinguin conflicte en els mateixos conjunts de la L3, amb una probabilitat força alta encerten a L3 i s'estalvien la resta de transaccions que s'esdevenen per culpa de la compartició falsa.

La compartició falsa es pot detectar quan una aplicació mostra un rendiment molt baix i el nombre d'invalidacions d'una memòria cau concreta i misos és molt més elevat del que s'esperava. En aquest cas, és molt probable que dos fils competeixin pels mateixos recursos de la memòria cau. Hi ha eines com el Vtune d'Intel (Intel, *Boost Performance Optimization and Multicore Scalability*, 2011) que permeten detectar aquest tipus de problemes.

Evitar la compartició falsa és relativament més senzill del que pot semblar. Un cop s'han detectat quines són les estructures que probablement causen aquest efecte, cal afegir un desplaçament per tal que caiguin en posicions de memòria diferent per a cadascun dels fils. D'aquesta manera les dades que abans compartien un mateix set de la memòria cau, aquest cop es desen en sets diferents.

5.3.2. Penalitzacions per a fallades a la L1 i tècniques de *prefetch*

Moltes aplicacions paral·leles tenen una alta localitat a les memòries cau del nucli sobre les quals s'executen. És a dir, la majoria d'accessos que es fan a memòria encerten la memòria cau de primer nivell.

Ara bé, els casos en què les peticions a memòria no encerten la memòria cau del nucli han de fer un procés molt més llarg fins que la dada és disponible per al fil: petició a la L3, fallada a la L3, petició a memòria, etc. Evidentment, com més alt és el percentatge de fallades, més penalitzada es troba l'aplicació. La causa és que un accés a memòria que falla a la L1 té una latència molt més elevada que una que l'encerta (un o dos ordres de magnituds més elevats, depenent de l'arquitectura i protocol de coherència).

Una de les tècniques que s'usa per a mirar d'amagar la latència més llarga de les peticions que fallaran a la memòria cau s'anomena *prefetching*. Aquesta tècnica consisteix a demanar la dada a memòria molt més aviat del que l'aplicació la necessita. D'aquesta manera, quan realment la necessita, aquesta ja es troba a la memòria local del nucli (L1). Per tant, l'encerta, la latència de la petició a memòria és molt més baixa i l'aplicació no es bloqueja.

Hi ha dos tipus de tècniques de *prefetching* usades en les arquitectures multinucli:

1) ***Hardware prefetching***. És una tècnica que s'implementa en les peces dels nuclis que s'anomenen *hardware prefetchers*. Aquests intenten predir quines són les properes adreces que l'aplicació demanarà i les demanen de manera proactiva abans que l'aplicació ho faci.

Aquest component es basa en tècniques de predicció que no requereixen una lògica gaire complexa, com les cadenes de Markov (Joseph i Grunwald, 1997). El problema principal d'aquest tipus de tècniques és que són agnòstiques res-

pecte del que l'aplicació està executant. Per tant, tot i que per algunes aplicacions pot funcionar força bé, per a altres les prediccions poden ser errònies i fer que el rendiment de l'aplicació empitjori.

2) ***Software prefetching***. Alguns dels sistemes multinucli faciliten instruccions per tal que les aplicacions puguin demanar de manera explícita les dades a memòria, usant la instrucció de *prefetch*.

La instrucció *vprefetch*

En l'arquitectura Intel es fa servir la instrucció *vprefetch* (Intel, *Use Software Data Prefetch on 32-Bit Intel / IA-32 Intel® Architecture Optimization Reference Manual*, 2011). L'aplicació és responsable de demanar les dades a memòria de manera anticipada usant aquesta instrucció.

L'avantatge d'aquest mecanisme és que l'aplicació sap exactament quan necessitarà les dades. Per tant, basant-se en la latència d'un accés a memòria, ha de demanar les dades amb el nombre de cicles suficient per tal que quan la necessiti la tingui. L'ús d'aquest tipus de *prefetch* és el que acostuma a permetre obtenir el rendiment més elevat de l'aplicació.

El desavantatge principal és que el codi es troba lligat a una arquitectura concreta. És a dir, la majoria d'arquitectures multinucli tenen latències diferents. Per tant, cada vegada que es vol executar aquesta aplicació en una plataforma nova cal calcular les distàncies de *prefetch* adequades al sistema nou.

La figura següent presenta un exemple de com funcionaria un *prefetch* en un sistema format per un multinucli amb una memòria cau de tercer nivell. En aquest cas l'aplicació necessita la dada *a* al cicle *X*; assumint que la latència d'un accés a memòria que falla a la L3 és de 400 cicles, ha de fer el *prefetchX*-400 cicles abans.

Figura 28. Exemple de *prefetch*

Cal tenir, però, les consideracions següents:

- Els *prefetchs* acostumen a poder ser eliminats del *pipeline* del processador si no hi ha prou recursos per a dur-lo a terme (per exemple, si la cua que desa les dades que tornen de la L3 és plena). Per tant, si el nombre de *prefetchs* que fa una aplicació és massa elevat, poden ser eliminats del sistema.

- La latència de les peticions a memòria poden variar depenent del camí que segueixin. Per exemple, un encert a la L3 fa que una dada estigui disponible a la mateixa memòria L3 molt més aviat del previst, i en el cas que hi hagi una víctima interna serà molt més tardana. Quan es fan servir aquest tipus de peticions cal tenir en compte l'ús de tota la jerarquia de memòria, com també les característiques de l'aplicació que s'usa.

- Els *prefetchs* poden tenir impactes de rendiment tant positius com negatius en l'aplicació desenvolupada. Cal estudiar quins requisits té l'aplicació desenvolupada i com es comporten en l'arquitectura que executa l'aplicació.

Lectura recomanada

L'ús de tècniques de *prefetch* en arquitectures multifil és un recurs freqüent per tal de treure rendiment en aplicacions paral·leles. Per aquest motiu, es recomana llegir l'article:

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

5.3.3. Impacte del tipus de memòria cau

Cada vegada que un nucli accedeix a una línia de memòria per primera vegada, l'ha de demanar al nivell de memòria següent. En els exemples estudiats, les fallades a la L2 es demanen a la L3, i les fallades a la L3 es demanen a memòria. Cadascuna d'aquestes fallades genera un conjunt de víctimes en les diferents memòries cau: cal alliberar una entrada per la línia que s'està demanant.

Per tal de treure rendiment a les aplicacions paral·leles que es desenvolupen és important mantenir al màxim la localitat en els accessos a les memòries cau. És a dir, maximitzar el reús (percentatge d'encert) a les diferents memòries cau (L1,L2, L3, etc.). Per aquest motiu és important considerar les característiques de la jerarquia de memòria cau: inclusiva, no inclusiva/exclusiva i mides.

A continuació es discuteixen els diferents punts esmentats des del punt de vista de l'aplicació.

Inclusivitat

Si dues memòries (Lx i Lx – 1) són inclusives vol dir que qualsevol adreça @X que sigui a Lx – 1 es troba sempre a la Lx. Per exemple, si la L1 és inclusiva amb la memòria L2, aquesta inclou la L1 i altres línies. En aquest cas cal considerar el següent:

1) Quan una línia de la Lx – 1 es victimitza, al final de la transacció aquesta està disponible al nivell següent de memòria Lx.

2) Quan una línia de la Lx es victimitza, al final de la transacció aquesta línia ja no està disponible al nivell superior Lx – 1. Per exemple, en el cas de victimitzar la línia @X a L3, aquesta s'invalida també a la memòria cau L2. I si la L1 és inclusiva amb la L2, la primera també invalida la línia en qüestió.

3) Quan un nucli llegeix una adreça @X que no es troba a la memòria Lx – 1, al final de la transacció aquesta també es troba inclosa a la Lx. Per exemple, en el cas que tinguem una L1, L2 i L3 inclusives, @X s'escriu a totes les memòries. Cal remarcar que cada una d'aquestes entrades usades potencialment ha generat una víctima. És a dir, una adreça @Y que usa el *way* i el *set* en el qual s'ha desat @X. La selecció d'aquesta posició depèn de la política de gestió de cada memòria cau, com també de la mida.

Els punts 2 i 3 poden causar un impacte força important en el rendiment de l'aplicació. Per tant, és important dissenyar les aplicacions per tal que el nombre de víctimes generades en nivells superiors sigui el més baix possible i maximitzar el percentatge d'encerts de les diferents jerarquies de memòria cau més properes al nucli (per exemple, L1).

Exclusivitat i no inclusivitat

Que dues memòries cau siguin exclusives implica que si la memòria cau Lx té una línia @X, la memòria cau Lx – 1 no la té. No s'acostumen a tenir arquitectures en què tots els nivells són exclusius. Habitualment les jerarquies que tenen memòries cau exclusives acostumen a ser híbrides.

Processadors amb memòria exclusiva

Un exemple de processador amb memòria exclusiva és l'AMD Athlon (AMD, 2011) i l'Intel Nehalem (Intel, *Nehalem Processor*, 2011). El primer té una L1 exclusiva amb la L2. El segon té una L2 i L1 no inclusives i la L3 és inclusiva de la L2 i la L1.

En els casos en què una memòria cau Lx és exclusiva amb Lx – 1 cal considerar el següent:

- Quan s'accedeix a una línia @Y a la memòria Lx – 1, aquesta es mou de la memòria Lx.

- Quan una línia es victimitza de la memòria Lx – 1, aquesta es mou a la memòria cau Lx. En aquests casos, com que la memòria és exclusiva, cal trobar una entrada en la memòria Lx. Per tant, cal victimitzar la línia que hi hagi desada al *way* i al *set* seleccionat.

- Quan una línia es victimitza de la memòria Lx no cal victimitzar la memòria cau Lx – 1.

Hi ha certes situacions en què cal saber detalladament quin tipus de jerarquia de memòria i protocol de coherència implementa el processador. Per exemple, si es considera una arquitectura multinucli en què la L1 és exclusiva amb la L2, en els casos en què els diferents fils estiguin compartint l'accés a un conjunt

elevat d'adreces el rendiment de l'aplicació es pot veure deduït. En aquesta situació, cada accés a una dada @X en un nucli podria implicar la victimització d'aquesta mateixa adreça en un altre nucli i demanar l'adreça a memòria.

Algunes memòries cau exclusives permeten que en certes situacions algunes dades es trobin en dues memòries que són exclusives entre si per defecte. Per exemple, en les línies de memòria compartides per diferents nuclis.

Mida de la memòria

El rendiment de l'aplicació depèn en gran mesura de la localitat dels accessos dels diferents fils en les memòries cau. En situacions en què els fils demanen diverses vegades les mateixes línies a memòria per mala praxi de programació, el rendiment de l'aplicació cau substancialment. Això passa quan un nucli demana una adreça @X, aquesta línia es victimitza i es torna a demanar més tard.

Un exemple senzill és a accedir a una matriu d'enters de 8 bytes per files quan aquesta s'ha emmagatzemat per columnes. En aquest cas, cada 8 enters consecutius d'una mateixa columna es trobarien mapats a la mateixa línia. Ara bé els elements i i $i + 1$ d'una fila es trobarien desats en línies diferents. Per tant, en el cas de recórrer a la matriu per columnes el nombre de fallades a la L1 serà molt més elevat.

Per aquest motiu, és important usar tècniques d'accés a les dades que intentin mantenir localitat a les diferents memòries cau.

No obstant això, en molts casos les aplicacions paral·leles dissenyades no segueixen cap dels patrons que hem analitzat en altres estudis acadèmics (per exemple, en tècniques de partició de matrius). Per a aquests problemes hi ha aplicacions disponibles que permeten analitzar com es comporten les aplicacions paral·leles i veure de quina manera es poden millorar.

Lectures recomanades

De manera semblant a d'altres factors ja introduïts anteriorment, s'ha fet molta recerca en aquest àmbit. Una referència en tècniques de partició en blocs per la multiplicació de matrius és la següent:

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

I en tècniques de compressió en el procés de grafs:

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

Exemple

Alguns exemples són Vtune (Kishan Malladi, 2011) o Cachegrind (Valgrind, 2011).

5.3.4. Arquitectures multinucli i multiprocessador

En algunes situacions, les arquitectures en què s'executen les aplicacions paral·leles no només donen accés a múltiples fils i múltiples nuclis, sinó a múltiples processadors. En la majoria de casos, aquestes arquitectures contenen una placa mare o més en què cada una conté un conjunt de processadors que estan connectats per mitjà d'una connexió d'alta velocitat. Si l'arquitectura de computació està formada per més d'una placa, acostumen a estar connectades per xarxes de connexió més lentes.

Tipus de connexions

Un exemple de connexió d'alta velocitat és l'Intel Quickpath Interconnect (Intel, *Intel® Quickpath Interconnect Maximizes Multi-Core Performance*, 2012), també conegut com a *QPI*. I com a exemple de connexió més lenta, tenim la Myrinet (Flich, 2000).

En aquestes arquitectures es poden assignar els diferents fils de l'aplicació a cadascun dels fils disponibles en cadascun dels nuclis dels diferents processadors connectats a una mateixa placa. Tots els diferents fils acostumen a compartir un espai de memòria coherent. Des del punt de vista de l'aplicació, aquesta té accés a N nuclis diferents i a un espai de memòria comú.

D'altra banda, els fils que s'han ubicat en plaques diferents no acostumen a compartir l'espai d'adreces de memòria. Per tant, si es vol que comparteixin informació, cal emprar un entorn que permeti fer-ho. Hi ha models de programació que ho permeten. El model de pas de missatges[20] n'és l'exemple més conegut. Aquest model de programació permet la comunicació de processos ubicats en plaques diferents (que, per tant, no comparteixen l'espai d'adreces)

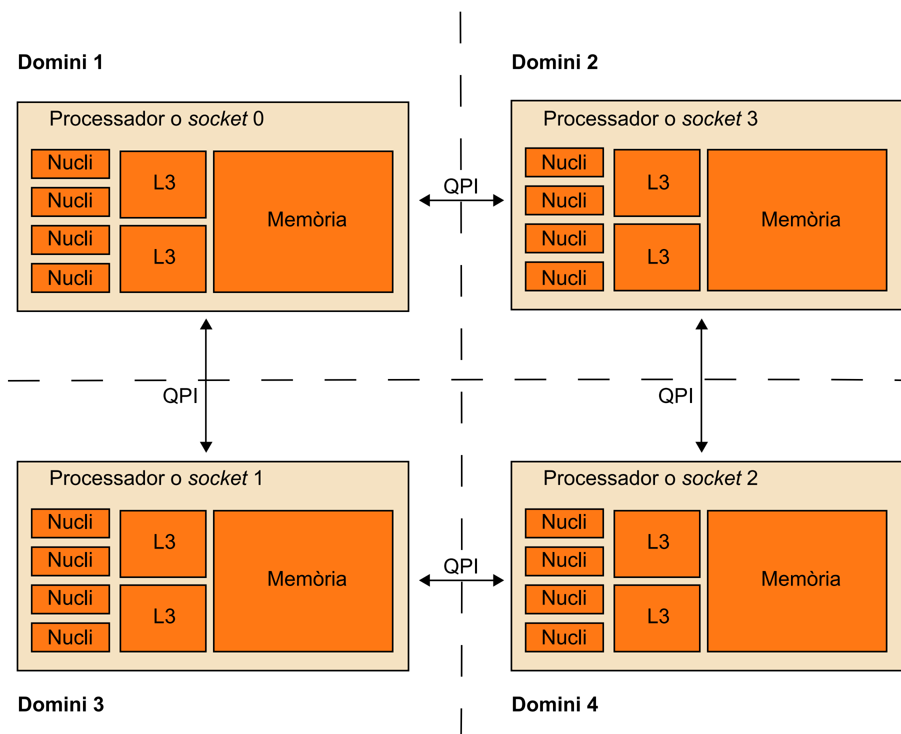
(20) En anglès, *message passing interface* (MPI).

per mitjà d'enviament de missatges. En molts casos, aquest model es combina amb OpenMP. Aquest darrer permet definir paral·lelisme dins d'una mateixa placa assumint que els diferents fils comparteixen el mateix espai d'adreces.

Tot i que aquest espai de memòria pot ser compartit per tots fils d'una mateixa placa, l'accés d'un fil a determinades zones de memòria pot tenir latències diferents. Cada processador disposa d'una jerarquia de memòria pròpia (des de la L1 fins a la memòria principal) i aquest gestiona un rang d'adreces de memòria concret. Així, si un processador disposa d'una memòria principal de 2 GB, aquest processador gestiona l'accés de l'espai d'adreces assignat a aquests 2 GB (per exemple, de 0x a FFFFFFFF).

Cada vegada que un fil accedeixi a una adreça que es troba assignada a un espai que gestiona un altre processador, ha d'enviar la petició de lectura d'aquesta adreça al processador que la gestiona a través de la xarxa d'interconnexió (com en una QPI). Aquests accessos són molt més costosos ja que han d'enviar la petició a través de la xarxa, arribar a l'altre processador i fer l'accés (seguint el protocol de coherència que segueixi l'arquitectura).

Figura 29. Arquitectura multifil



Aquest model de programació segueix el paradigma del que s'anomena *non-uniform memory access* (NUMA), en què un accés de memòria pot tenir diferents tipus de latència depenent d'on es trobi assignat.

Quan es programi per aquest tipus d'arquitectura, cal considerar tots els factors que s'han explicat en aquest apartat, però en una escala superior. Així doncs, en l'accés a variables d'exclusió mútua s'ha de considerar que, per cada vegada que s'agafi el *lock*, cal invalidar tota la resta de nuclis del sistema. Els que siguin a fora del processador van per la xarxa d'interconnexió i tarden més a retornar la resposta. Val a dir que el comportament depèn del protocol de coherència i de la jerarquia de memòria del sistema.

En aquest mòdul s'han introduït alguns dels aspectes més importants que cal tenir en compte a l'hora de dissenyar i desenvolupar aplicacions per a arquitectures de computació d'altes prestacions. Com s'ha pogut veure amb la gran quantitat de referències facilitades, aquest àmbit és molt complex i, per tal d'aprofundir-hi, cal dedicar-hi molts esforços i dedicació.

Lectures recomanades

Dins l'àmbit de programació multifil, per aquest tipus d'arquitectures s'hi poden trobar moltes referències interessants. Algunes són les següents:

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

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

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

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

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.

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/.

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.

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".

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

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.

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.

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*.

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).