

Títol: Generació de codi per a l'arquitectura CELL BE

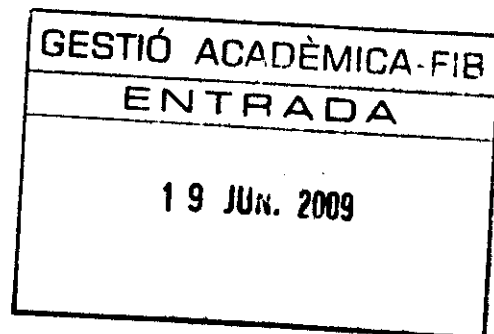
Volum: 1

Alumne: Sandra Franch Monràs

Director/Ponent: Marc González Tallada

Departament: Arquitectura de Computadors

Data: 16/06/2009



10/10/10

10/10/10

10/10/10

ÍNDEX

| | |
|---|----|
| ÍNDEX..... | 1 |
| 1 INTRODUCCIÓ..... | 5 |
| 1.1 Definició d'Arquitectures Multi nucli | 5 |
| 1.2 Principals Processadors Multi nucli de l'Actualitat | 6 |
| 1.2.1 INTEL..... | 6 |
| 1.2.2 AMD..... | 9 |
| 1.2.3 APPLE – IBM - MOTOROLA | 12 |
| 1.2.4 SUN..... | 12 |
| 1.2.5 SONY - TOSHIBA – IBM | 13 |
| 1.3 Tècniques de la Programació Multi nucli | 13 |
| 1.3.1 OPENMP | 15 |
| 1.3.2 MPI | 17 |
| 1.3.3 INTEL THREADING BUILDING | 20 |
| 1.3.4 CELLSs (SuperScalar)..... | 21 |
| 1.3.5 PTHREADS..... | 23 |
| 2 MOTIVACIÓ DEL PROJECTE I OBJECTIUS | 27 |
| 3 TREBALL RELACIONAT | 29 |
| 3.1 Plataformes de Compilació | 29 |
| 3.2 Programabilitat i Memòries Locals | 30 |
| 3.3 Models de programació | 30 |
| 4 PROCESSADOR CELL BE | 31 |
| 4.1 Estructura..... | 31 |
| 4.2 Suport de Compilació i Execució del Processador CELL BE | 34 |

| | | |
|-------|--|----|
| 4.2.1 | TILING | 34 |
| 4.2.2 | SOFTWARE CACHE..... | 35 |
| 5 | SOFTWARE CACHE HÍBRIDA | 41 |
| 5.1 | Disseny de la Software Cache Híbrida..... | 41 |
| 5.1.1 | Cache d'Alta Localitat..... | 41 |
| 5.1.2 | Cache Transaccional..... | 43 |
| 5.1.3 | Consistència de Memòria..... | 45 |
| 5.2 | Transformacions de Codi per Aquesta Software Cache..... | 46 |
| 5.2.1 | Classificació d'accessos a memòria..... | 46 |
| 5.2.2 | Transformacions dels accessos d'alta localitat | 46 |
| 5.2.3 | Transformacions per accessos irregulars..... | 47 |
| 6 | COMPILADOR NANOS MERCURIUM C/C++..... | 49 |
| 6.1 | Introducció al Compilador Nanos Mercurium C/C++ | 49 |
| 6.2 | Fitxers de l'entorn del compilador..... | 53 |
| 7 | DISSENY DE LA NOVA FASE..... | 55 |
| 7.1 | Desenvolupant noves fases del compilador | 55 |
| 7.2 | Desenvolupant la nostra fase del compilador | 57 |
| 7.3 | Ubicació nova fase dins de l'estructura del compilador | 58 |
| 7.4 | Disseny Generació de Codi..... | 59 |
| 7.4.1 | Transformació de codi..... | 59 |
| 7.4.2 | Transformacions del cos del bucle..... | 62 |
| 8 | IMPLEMENTACIÓ..... | 69 |
| 8.1 | Entorn de Desenvolupament | 69 |
| 8.2 | Introducció a TL..... | 69 |
| 8.2.1 | Compilador de fases..... | 69 |

| | | |
|-------|--|----|
| 8.2.2 | Classes <i>WRAPPER</i> | 72 |
| 8.2.3 | Creant/Omplint el <i>SOURCE</i> | 72 |
| 8.2.4 | Recorrent al <i>SOURCE</i> creat..... | 73 |
| 8.2.5 | Tractar amb els constructors del llenguatge comú..... | 73 |
| 8.2.6 | Functors i signals..... | 74 |
| 8.2.7 | Travessant l'arbre..... | 75 |
| 8.3 | Pseudocodi de l'Algorisme..... | 75 |
| 9 | AVÀLUACIÓ..... | 79 |
| 10 | COST I PLANIFICACIÓ..... | 85 |
| 10.1 | Cost del projecte..... | 85 |
| 10.2 | Planificació Inicial..... | 86 |
| 10.3 | Planificació Final..... | 88 |
| 10.4 | Desviacions de la Planificació..... | 90 |
| 11 | CONCLUSIONS I TREBALL FUTUR..... | 91 |
| 11.1 | Conclusions..... | 91 |
| 11.2 | Treball Futur..... | 91 |
| | ANNEX A : INSTAL·LACIÓ I CONFIGURACIÓ NANOS MERCURIUM C/C++..... | 95 |
| | BIBLIOGRAFIA..... | 99 |

1 INTRODUCCIÓ

En aquest primer capítol es fa una introducció als processadors d'arquitectura multi nucli, s'analitzen els processadors multi nucli més importants actualment, i s'introdueixen les diferents tècniques de programació per processadors multi nucli.

Cada vegada la miniaturització dels components del processador es fa més difícil; el problema la dissipació de calor i el consum d'energia, produint que sigui més difícil augmentar la freqüència principal del processador. Tots aquests problemes dificulten l'augment de rendiment dels processadors seguint el paradigma mono processador.

Aquests processadors necessiten grans dissipadors i ventiladors perquè generen molta calor i no es podia continuar fabricant processadors de la mateixa manera, s'estava arribant a un "estancament"; calia prendre un altre camí, utilitzar una altra variable que fes que el rendiment del processador augmentés. Aleshores, basant-se en el processament en paral·lel, es van començar a construir els processadors multi nucli.

A continuació veurem la definició d'arquitectures multi nucli, els principals processadors multi nucli en l'actualitat i les tècniques de programació per aquests processadors.

1.1 DEFINICIÓ D'ARQUITECTURES MULTI NUCLI

Un processador multi nucli és un processador que conté en el seu interior varis nuclis. Mentre que els processadors que són mono nucli, o sigui que tenen un sol nucli per executar processos, els processadors multi nucli poden repartir els processos entre els seus nuclis. D'aquesta manera si els processos són divisibles poden acabar més ràpid. Només quan s'executa una sola aplicació que no sigui paral·lelitzable (no es pugui descompondre en *threads*) és quan no s'aprofita el potencial de processament que tenen aquests processadors.

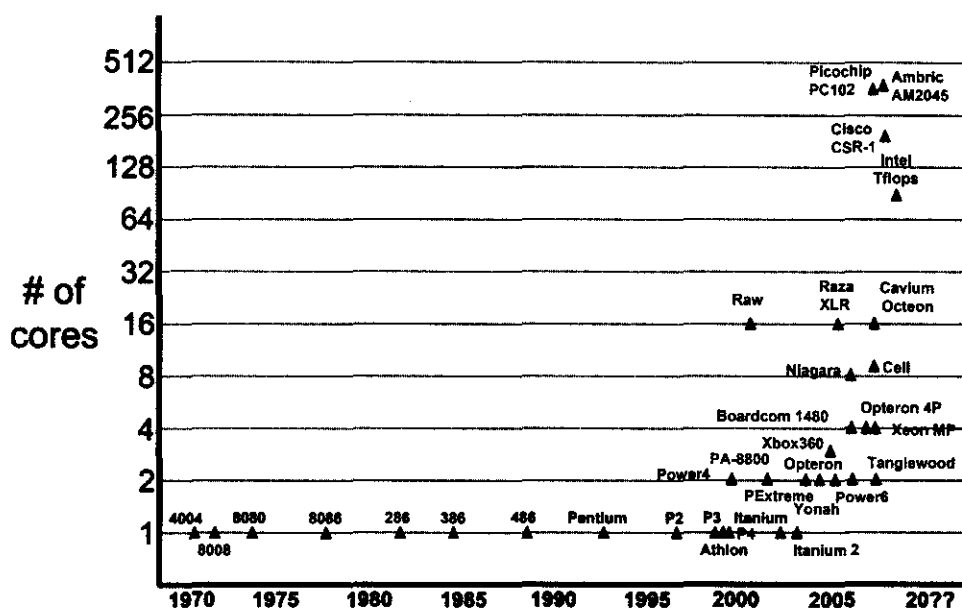
Per fabricar els processadors multi nucli es van basar en els sistemes distribuïts, la computació paral·lela, i les tecnologies com el *Hyperthreading*¹; que mostraven com dividir el treball entre diverses unitats d'execució. El processament en paral·lel és la divisió d'una aplicació en diverses parts per tal que siguin executades a la vegada per diferents unitats d'execució.

Les aplicacions que li treuen més profit a aquests processadors, són aquelles que poden generar molts *threads* d'execució com aplicacions d'àudio, vídeo, càlculs científics, jocs, tractament de gràfics 3D, etc.

¹ *HyperThreading* és una marca registrada de l'empresa Intel per nombrar la seva implementació de la tecnologia SMP (Multiprocés simètric). Permet als programes preparats per executar múltiples *threads* processar-los en paral·lel dins d'un únic processador.

Actualment molts programes són poc paral·lelitzables (excepte en els sectors on s'usen superordinadors, sistemes distribuïts i paral·lels, etc.), però es poden executar molts d'ells a la vegada. Els processadors multi nucli també s'usen en el disseny de sistemes multiprocessadors, que consisteixen en una placa mare que pot suportar des de 2 a més processadors.

Com història es pot dir que la primera CPU multi nucli en el mercat va ser el IBM Power4 l'any 2000. En la següent imatge veiem una gràfica dels processadors per any i per número de nuclis.



1.2 PRINCIPALS PROCESSADORS MULTI NUCLI DE L'ACTUALITAT

Els principals fabricants de processadors clarament han apostat per les arquitectures multi nuclis. A continuació descrivim diferents propostes.

1.2.1 INTEL

1.2.1.1 PENTIUM D

Els processadors Pentium D [1] va ser introduïts per Intel el 2005. Un chip Pentium D consisteix bàsicament en 2 processadors Pentium 4 integrats en un chip i comunicats a través del FSB²(Front Side Bus). El seu procés de fabricació va ser inicialment de 90 nm (anomenats "Smithfield") y en la seva segona generació de 65 nm (anomenats Presler). Els processadors Pentium D no són monolítics, és a dir, els nuclis no comparteixen una única cache i la comunicació entre ells no es directa, es realitza a través del FSB.

² FSB (Front Side Bus - "Bus de la part frontal"), és el tipus de bus usat com a bus principal en alguns processadors de la marca Intel per comunicar-se amb el chipset.



Fabricació: 2005 al 2007

Fabricant: Intel

Velocitat de CPU: 2,66 GHz a 3,73 GHz

Velocitat de FSB: 533 MHz a 1066 MHz

Mida processos: 90 nm a 65 nm

Conjunt d'Instruccions: MMX, SSE, SSE2 i EM64T

Microarquitectura: NetBurst

Nuclis: 2 (2x1)

Socket: LGA 775

Nom dels nuclis: Smithfield i Presler

1.2.1.2 CORE DUO

El Core Duo [2] és un microprocessador de sexta generació llançat el gener de 2006 per Intel amb 2 nuclis d'execució, optimitzat per les aplicacions de subprocessos múltiples i per la multi tasca. Pot executar varies aplicacions exigents simultàniament. Es un model anterior al Core 2 Duo i posterior als Pentium D. Intel Core Duo és el primer microprocessador de Intel utilitzat en les computadors de Apple Macintosh.

El Core Duo té 151 milions de transistors, incloent la memòria cache de 2Mb. Els 2 nuclis es comuniquen a través del FSB de 667 ó 533 MHz.



Fabricació: 2006 fins 2008

Fabricant: Intel

Velocitat de CPU: 1.06 GHz to 2.33GHz

Velocitat de FSB: 533 MT/s to 667 MT/s

Mida processos: 65nm

Conjunt d'Instruccions: x86-686

Microarquitectura: P6 (Pentium M) derivat

Nuclis: 1 or 2

Socket: Socket M

Nom del nucli: Yonah

1.2.1.3 CORE 2 DUO

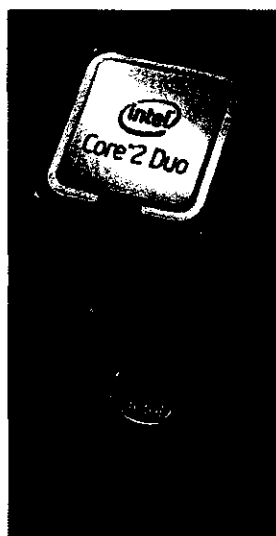
La marca Core 2 [3] es refereix a una gamma de CPU comercials d'Intel de 64 bits de doble nucli i CPU 2x2 MCM³ de quatre nuclis amb el conjunt d'instruccions x86-64, basat en la microarquitectura Core d'Intel, derivat del processador portàtil de doble nucli de 32 bits Yonah. El Core 2 va fer que els Pentiums anessin a un mercat de gamma mig-baix, i va reunifica

³ MCM (Mòdul Mutli-Chip)

les línies de sobretaula i portàtils, les quals prèviament havien estat dividides en les marques Pentium 4, D, i M.

La microarquitectura Core tornava a velocitats de CPU baixes i millorava l'ús del processador de dos cicles de velocitat i energia, comparats amb els anteriors NetBurst⁴ de la CPU Pentium 4 / D. La microarquitectura Core proveeix etapes de descodificació, unitats d'execució, cache i busos més eficients, reduint el consum d'energia de les CPUs Core 2, mentre s'incrementa la capacitat de processament. Les CPUs d'Intel han variat molt bruscament en consum d'energia d'acord a velocitat de processador, arquitectura i processos de semiconductor.

La marca Core 2 va ser introduïda el 27 de juliol de 2006, abastant les línies **Solo** (un nucli), **Duo** (doble nucli), **Quad** (quad-core), i **Extreme** (CPUs de dos o quatre nuclis), durant el 2007. Els processadors Intel Core 2 amb tecnologia VPRO⁵ (dissenyats per negocis) inclouen les branques de doble nucli i quatre nuclis.



Fabricació: Des de 2006 fins actualment

Fabricant: Intel

Velocitat de CPU: 1.06 GHz a 3.33 GHz

Velocitat de FSB: 533 MT/s a 1600 MT/s

Mida processos: 65 nm a 40 nm

Conjunt d'Instruccions: x86, MMX, SSE, SSE2, SSE3, SSSE3, x86-64, SSE4.1

Microarquitectura: Intel Core microarchitecture

Nuclis: 1, 2, or 4 (2x2)

Socket:Socket T (LGA 775)

Socket M (μ PGA 478)

Socket P (μ PGA 478)

Micro-FCBGA (μ BGA 479)

Nom del nucli: Allendale, Conroe, Merom-2M, Merom, Kentsfield, Wolfdale, Yorkfield, Penryn

⁴ Microarquitectura Intel NetBurst, anomenat P68 dins d'Intel, va ser el successor de la microarquitectura P6 a la família de CPU x86 realitzats per Intel. L'arquitectura NetBurst d'Intel inclou característiques com ara "Hyper Pipelined Technology" i "Rapid Execution Engine", que són els primers en aquest microarquitectura.

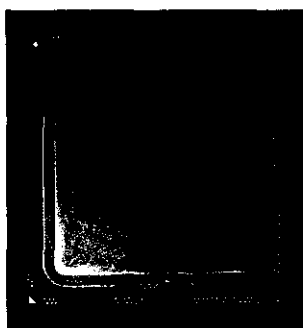
⁵ La tecnologia Intel VPRO és un conjunt de característiques en una placa base del PC i altre maquinari. Intel VPRO no és el propi PC, ni tampoc és un conjunt únic de característiques de gestió (com la tecnologia d'administració activa d'Intel (Intel AMT) per al sistema d'administradors. Intel VPRO és una combinació de tecnologies de processament, l'equipament de maquinari, funcions de gestió i tecnologies de seguretat que permeten l'accés remot al PC - inclosa la vigilància, el manteniment i la gestió - amb independència de l'estat del sistema operatiu (SO) o estat d'energia del PC.

1.2.1.4 CORE I7

Intel Core i7 [4] és una família de tres processadors de l'arquitectura Intel x86-64. Intel Core i7 són els primers processadors que usen micrbarquitectura Nehalem⁶ d'Intel i és el successor de la família Core 2. Els tres models són processadors de quatre nuclis.

Intel va llançar aquesta família de processadors el 17 de Novembre del 2008 a Costa Rica. Un altre punt interessant és que Core i7 va ser creat per Intel Costa Rica.

S'ha re-implementat el *HyperThreading*. Cadascun dels quatre Cores pot processar dues tasques simultàniament, per tant el processador ofereix 8 fluxes d'execució.



Fabricació: Des de 2008

Fabricant: Intel

Velocitat de CPU: 2,66 GHz a 3,33 GHz

Velocitat de FSB: 4.8 GT/s a 6.4 GT/s

Mida Processos: 45 nm

Conjunt d'instruccions: x86, x86-64, MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2

Microarquitectura: Nehalem

Nuclis: 4 (físics), 8 (lògics)

Socket: Socket B (LGA 1366)

Nom del nucli: Bloomfield

1.2.2 AMD

Abans de treure els seus processadors multi nucli al mercat, AMD ja havia aconseguit un gran èxit amb el seu processador Athlon 64, el primer processador de 64 bits. Incorporava la tecnologia HyperTransport que era un nou bus bastant ràpid que eliminava els anteriors colls d'ampolla, i altres tecnologies; aquest processador va ser pres com a base per a la construcció del seu processador de doble nucli Athlon 64 X2, que va sortir al mercat a partir del 2005.

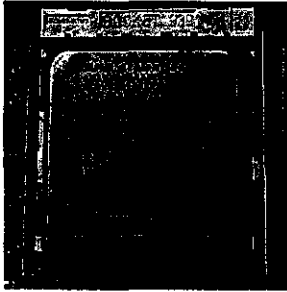
1.2.2.1 DUAL CORE ATHLON 64 X2

L'AMD Athlon 64 X2 [5] és un microprocessador de 64 bits multi nucli produït per AMD. Aquest microprocessador va ser introduït per al socket 939 (en 90 nm) i per al socket AM2 (en 90 nm i 65 nm) amb un bus HyperTransport de 2000 Mhz i suport de memòria DDR2 a partir dels models AM2 i conjunt d'instruccions SSE3⁷. Cada nucli compta amb una unitat de memòria cache independent, i tenen entre 154 a 233,2 milions de transistors depenent de la mida de la cache. Els nous processadors que van aparèixer al mes de juliol de 2006 per al socket AM2

⁶ Nehalem és el nom clau per a un processador amb la nova micrarquitectura d'Intel successora de la micrarquitectura Core.

⁷ SSE3, Streaming SIMD Extensions 3, és la tercera generació del conjunt d'instruccions SSE de l'IA-32 (Intel Architecture, 32-bit). SSE es un conjunt d'instruccions de SIMD per a l'arquitectura x86. SIMD (Single Instruction, Multiple Data col·loquialment "vector d'instruccions") és una tècnica emprada per aconseguir el paral·lelisme a nivell de dades.

compten amb el suport per memòria DDR2, van ser fabricats en 90 nm i 65nm i van incloure tecnologies de virtualització i millores en el consum d'energia. La principal característica d'aquests processadors és que contenen dos nuclis i poden processar diverses tasques a la vegada. El rendiment és molt millor que els processadors d'un únic nucli. A més la seva arquitectura és de 64-bits. El microprocessador AMD Turion 64 X2 és una versió de baix consum del processador AMD Athlon 64 X2 destinada als ordinadors portàtils.



Fabricació: A partir de 2005
Dissenyat per: AMD
Fabricant: GlobalFoundries
Velocitat de CPU: 1,0 GHz a 3,2 GHz
Velocitat de FSB: de 1000 MT / s
Mida Processos: 90nm a 65nm
Conjunt d'instruccions: MMX, SSE, SSE2, SSE3, x86-64, 3DNow!
Microarquitectura: microarquitectura K9
Nuclis: 2
Socket (s): Socket 939 i Socket AM2

1.2.2.2 TURION X2

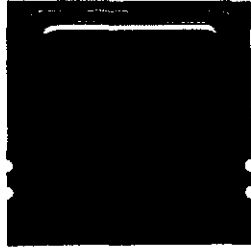
El processador AMD Turion 64 X2 [6] és la versió de doble nucli del Turion i per tant la versió de baix consum per portàtil del AMD Athlon 64 X2. Llançat al mercat el maig de 2006 per a competir amb els Core Duo d'Intel i posteriorment els Core 2 Duo, els dos de la plataforma Centrino Duo. Va ser la primera CPU per portàtils en combinar doble nucli i 64 bits.



Fabricació: 2006
Fabricante: AMD
Velocidad de CPU: 1.6 GHz a 2.2 GHz
Velocidad de FSB: 1600 MT/s
Procesos: 90 nm
Conjunt d'instruccions: AMD64
Microarquitectura: K8
Socket: Socket S1
Cores: Taylor i Trinidad

1.2.2.3 OPTERON

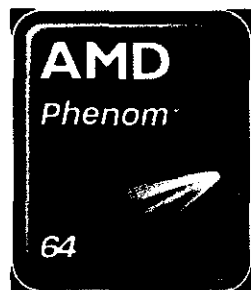
AMD Opteron [7] és la versió per a servidors corporatius del K8, i encara que va ser concebuda per la companyia per competir contra la línia IA-64 Itanium d'Intel, donats els baixos volums de venda i producció d'aquesta darrera, competeix actualment amb la línia Xeon d'Intel .



Fabricació: Desde abril de 2007
Fabricant: AMD
Velocitat de CPU: 1.4 GHz a 3.2 GHz
Velocitat de FSB: 800 MHz a 1000 MHz
Processos: 0.13µm to 45nm
Conjunt d'instruccions: x86, x86-64
Nuclis: 1, 2, o 4

1.2.2.4 PHENOM

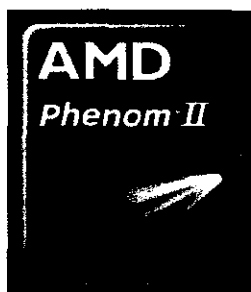
Phenom [8] és el nom donat per AMD a la primera generació de processadors de tres i quatre nuclis basats en l'arquitectura K10. Aquest nom va ser donat a conèixer a finals d'abril del 2007, substituint així a la sèrie d'alt rendiment AMD (Athlon 64 X2). Els primers dos models de la sèrie 8000 (Phenom X3 8400@2.1GHz i el X3 8600@2.3GHz) van ser llançats al mercat el març del 2008. Aquests microprocessadors compten amb tres nuclis i AMD afirma que milloren el rendiment fins en un 30% respecte a un microprocessador AMD de doble nucli a igual freqüència.



Fabricació: Des de 2007
Dissenyat per: AMD
Fabricant: GlobalFoundries
Velocitat de CPU: 1.8 GHz a 3.0 GHz
Velocitat de FSB: 1.6 GHz a 2.0 GHz
Mida Processos: 65 nm a 45 nm
Conjunt d'Instruccions: MMX, SSE, SSE2, SSE3, SSE4a, x86-64, 3DNow!
Microarquitectura: K10
Nuclis: 3, 4
Socket(s): Socket AM2+

1.2.2.5 PHENOM II

AMD per fi aconsegueix fer el salt de la fabricació de 65 nm a 45 nm amb el seu nou Phenom II [9], llançat a principis del 2009, encara que aquest xip no es pot igualar als poderosos processadors Intel Core i7 ni als Core 2 Quad de major velocitat. Aquest xip però com la majoria dels AMDs té un preu assequible. Funciona amb la plataforma AM2+, i incorpora les mateixes instruccions que els Phenom.



Fabricació: Des de desembre de 2008
Dissenyat per: AMD
Fabricats: GlobalFoundries
Velocitat de CPU: 2.5 GHz a 3.2 GHz
Velocitat de FSB: 1800 MHz a 2000 MHz
Mida Processos: 45 nm
Conjunt d'Instruccions: x86-64
Microarquitectura: AMD K10
Nuclis: 3 o 4
Socket(s): Socket AM2+ i Socket AM3

1.2.3 APPLE – IBM - MOTOROLA

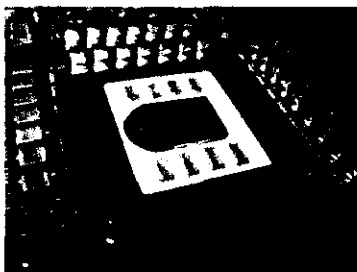
PowerPC és una família de processadors d'arquitectura RISC⁸, desenvolupada inicialment per Apple, Motorola i IBM. Pensada com a alternativa a la família i386 d'Intel, ha tingut un cert ressò gràcies al fet de ser els únics processadors utilitzats per Apple Macintosh fins fa poc, ja que actualment inclouen els processadors d'Intel.

1.2.3.1 POWERPC G5

L'última generació és el PowerPC G5 [10] (nom comú del PowerPC 970 i del PowerPC 970FX) que és un microprocessador d'alt rendiment amb arquitectura RISC de 64 bits, pertany a la família PowerPC, dissenyat i fabricat per IBM el 2002. El PowerPC 970 està construït usant tecnologia de 130nm, i el 970FX, de 90 nm. Contenen més de 58 milions de transistors. Estan basats en el desenvolupament dels Power4 d'IBM. Addicionalment, són capaços de processar instruccions de 32 bits en mode natiu.

Té un rendiment excepcional en comparació amb altres processadors i amb una capacitat de direcció de memòria fins a 8 GB.

Per mantenir les convencions de nom adoptades per Apple a la seva gamma d'ordinadors, va denominar a aquest processador G5 el juny de 2003. El terme G5 en aquest context s'identifica amb la cinquena generació de processadors PowerPC utilitzats per Apple, en la seva gamma Power Mac G5. Des de llavors, el PowerPC 970FX ha reemplaçat al PowerPC 970 en els ordinadors d'Apple.



1.2.4 SUN

1.2.4.1 ULTRASPARC T1 NIAGARA

El UltraSparc T1 Niagara, poderós processador per a servidors, que genera un gran estalvi d'energia per la seva relació rendiment / energia. El seu fabricant Sun Microsystems abans també ha tret altres processadors multi nucli per als seus servidors.

⁸ Reduced Instruction Set Computer, representa una estratègia de disseny de CPU insistint en la idea que les instruccions simplificades que "fan menys" encara poden oferir un major rendiment si aquesta simplicitat pot ser utilitzat per executar instruccions molt ràpidament.

1.2.5 SONY - TOSHIBA – IBM

1.2.5.1 CELL BE

El processador CELL BE [11], és un processador multi nucli dissenyat per les empreses IBM, Toshiba i Sony Computer Entertainment, una aliança coneguda com a "ITS" durant un període de 4 anys a partir del març del 2001.



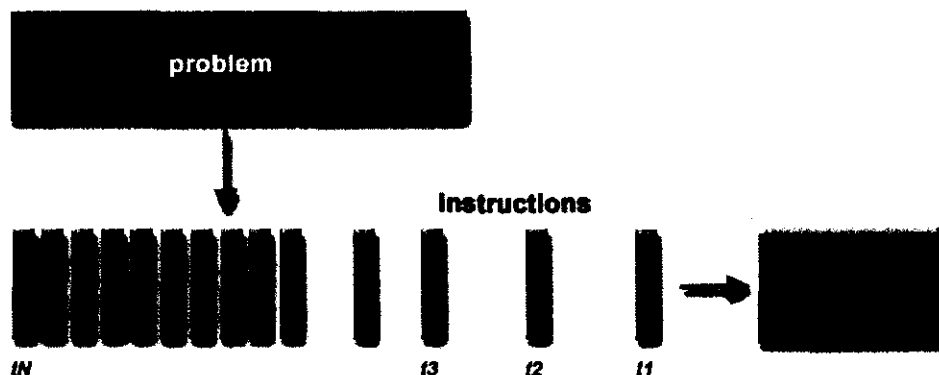
CELL BE és l'abreviatura de **CELL Broadband Engine Architecture**, comunament abreujat CBEA en la seva totalitat o en part CELL BE.

És una part important en el nostre projecte per tant el veurem en més detall en el capítol 3.

1.3 TÈCNIQUES DE LA PROGRAMACIÓ MULTI NUCLI

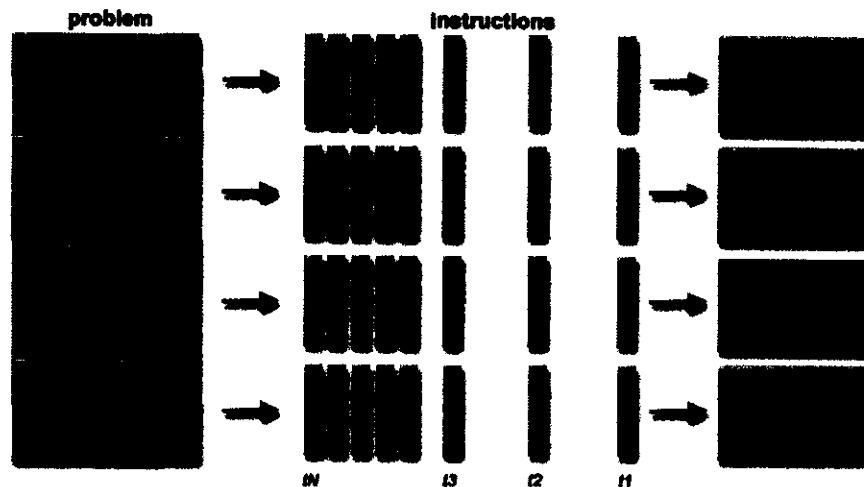
Tradicionalment, el programari era escrit en sèrie:

- Per a ser executat en un únic equip amb una única CPU.
- Un problema es divideix en una sèrie de discretes instruccions.
- Les instruccions s'executen una darrera l'altra.
- Només una instrucció pot executar-se en qualsevol moment.



En el sentit més simple, la computació paral·lela és l'ús simultani de múltiples recursos computacionals per a resoldre un problema:

- Per ser executat utilitzant múltiples CPU o una CPU amb varis nuclis.
- Un problema es divideix en parts diferenciades que es poden resoldre simultàniament
- Cada part es desglossen en una sèrie d'instruccions.
- O les instruccions de cada una de les parts s'executa simultàniament en diferents CPU.



El càlcul de recursos poden incloure:

- Un sol ordinador amb diversos processadors.
- Un sol ordinador amb un processador amb diversos nuclis.
- Un nombre arbitrari d'ordinadors connectats per una xarxa.
- Una combinació d'ells.

El problema computacional normalment demostra característiques com la capacitat de ser:

- Trencat a part o en peces discretes de treball que es poden resoldre al mateix temps.
- Executar més d'un programa o instruccions en qualsevol moment.
- Solucionat en menys temps amb múltiples recursos de càlcul que amb un únic recurs de càlcul.

Actualment, les tècniques o models⁹ per programar processadors multi nuclis són:

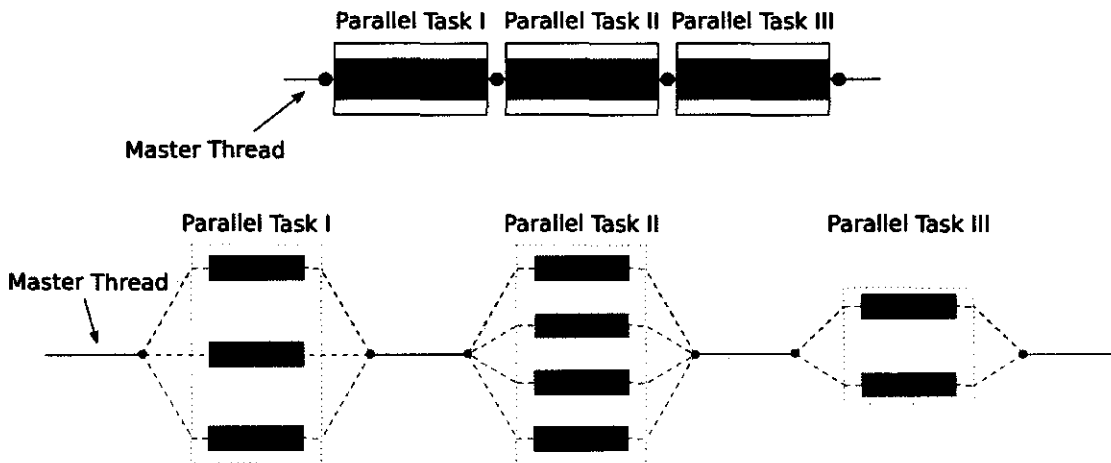
⁹ Un model de programació es una eina (bé pot ser una forma de utilitzar eines pròpies del sistema operatiu o una peça de software específica) que permet el pas de les aplicacions escrites seguint un model/filosofia natural del programador a aplicacions adaptades al maquinari disponible. Hi ha desenes de models de programació diferents, cada un d'ells es més o menys adequat a diferents propòsits i arquitectures.

1.3.1 OPENMP

OpenMP [13] (Open Multip-Processing) és una interfície de programació d'aplicacions (anomenada API) que suporta la programació multiprocés amb memòria compartida multi-plataforma en C/C++ i Fortran a moltes arquitectures, incloent les plataformes Unix i Microsoft Windows. Consisteix en un conjunt de directives de compilador, biblioteques de rutines, i variables d'entorn que afecten al comportament en temps d'execució.

Definit conjuntament per un grup dels principals fabricants de maquinari i programari, OpenMP és un model portable i escalable que dóna als programadors una interfície simple i flexible per a desenvolupar aplicacions paral·leles per a plataformes que van des de l'escriptori fins als supercomputadors.

OpenMP és una implementació de *multithreading*, un mètode de paral·lelització pel qual el *thread* pare (*thread* mestre) es "bifurca" en un nombre determinat de *threads* fills (*threads* esclaus) i la tasca es divideix entre ells. Els *threads* llavors s'executen concurrentment, amb l'entorn d'execució ubicant els *threads* a diferents processadors.



La secció de codi que es vol que s'executi en paral·lel és marcada corresponentment, amb una directiva de preprocessador que farà que els *threads* es creïn abans que la secció sigui executada. Cada *thread* té un "id" (identificador) associat que pot ser obtingut fent servir una funció (anomenada `omp_get_thread_num()` a C/C++ i `OMP_GET_THREAD_NUM()` a Fortran). L'id del *thread* és un enter, i el *thread* pare té una id de "0". Després de l'execució del codi paral·lelitzat, els *threads* s'uneixen de nou al *thread* pare, que continua endavant fins al final del programa.

Per defecte, cada *thread* executa la secció de codi paral·lelitzada independentment. Es poden fer servir "construccions de compartició de treball" per a dividir una tasca entre els *threads* de forma que cada *thread* executi la part de codi que té assignada. Fent servir OpenMP d'aquesta forma es pot aconseguir tant el paral·lelisme de tasques com el paral·lelisme de dades.

L'entorn d'execució assigna els *threads* a processadors depenent de l'ús, la càrrega de la màquina i altres factors. El nombre de *threads* pot ser assignat per l'entorn d'execució basant-

se en variables d'entorn o en codi fent servir funcions. Les funcions OpenMP estan incloses en un fitxer de capçalera amb el nom "omp.h" a C/C++.

Directives

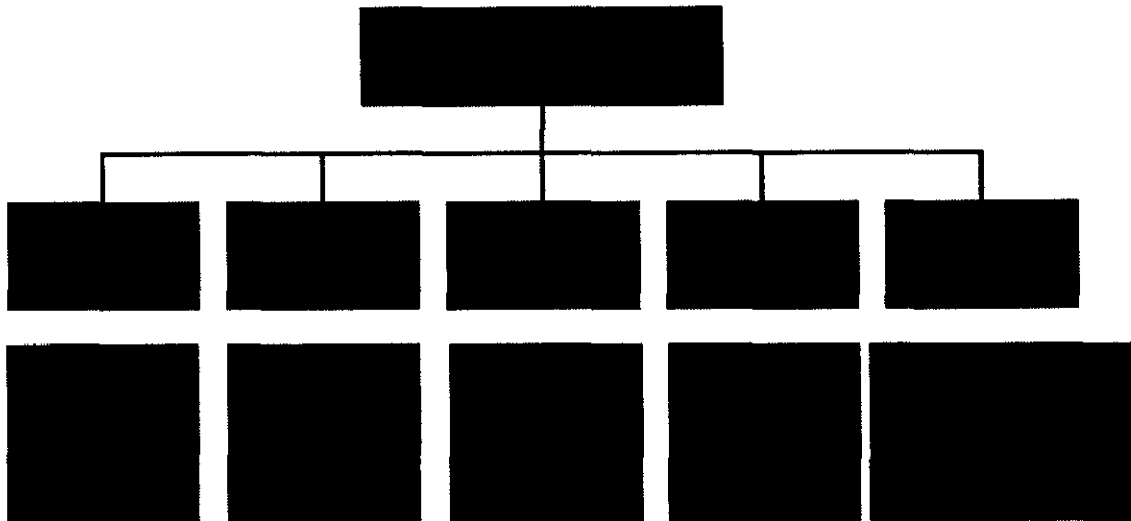
També se solen anomenar constructors:

- **parallel:** Aquesta directiva ens indica que la part de codi que la comprèn pot ser executada per diversos *threads*.
- **for:** Igual que paral·lel però optimitzat pels bucles for. El seu format és:

```
# pragma parallel for [clàusula, ... , clàusula]
```
- **section i sections:** Indica seccions que es poden executar en paral·lel però per un únic *thread*.
- **single:** La part de codi que defineix aquesta directiva, només es pot executar un únic *thread* de tots els llançats, i no ha de ser obligatòriament el *thread* pare.
- **master:** La part de codi definida, només es pot executar pel *thread* pare.
- **critical:** Només un *thread* pot estar en aquesta secció. Defineixen seccions crítiques o condicions de carrera.
- **atomic:** S'utilitza quan l'operació només afecta a una posició de memòria, i ha de ser actualitzada només per un *thread* simultàniament. Operacions tipus `x++` o `--x` són les que fan servir aquesta clàusula.
- **flush:** Aquesta directiva resol la consistència, en exportar a tots els *threads* un valor bàsic d'una variable que ha realitzat un altre *thread* en el processament paral·lel.

Funcions

- **omp_set_num_threads:** Fixa el nombre de *threads* simultanis.
- **omp_get_num_threads:** retorna el nombre de *threads* en execució.
- **omp_get_max_threads:** retorna el nombre màxim de *threads* que llançarà el nostre programa a les zones paral·leles. És molt útil per reservar memòria per a cada *thread*.
- **omp_get_thread_num:** retorna el nombre del *threads* dins l'equip (valor entre 0 i `omp_get_num_threads () -1`)
- **omp_get_num_procs:** retorna en nombre de processadors del nostre ordinador o disponibles (per a sistemes virtuals).
- **omp_set_dinamic:** Valor booleà que ens permet especificar si volem que el nombre de *threads* creixi i decreixi dinàmicament.



Avantatges

- Simple.
- Les dades de disseny i la descomposició es maneja automàticament per les directives.
- Increment del paral·lisme: poden treballar en una part del programa en un moment, no hi ha canvis dràstics al codi que es necessita.
- Codi Unificat per ambdues aplicacions de sèrie i paral·lel: Les directives de OpenMP són tractades com comentaris quan s'utilitzen en compiladors seqüencials.

Desavantatges

- En l'actualitat només s'executa de manera eficient en plataformes multiprocessador en memòria compartida.
- Requereix un compilador que suporti OpenMP.
- Escalabilitat de memòria està limitada per l'arquitectura.
- Manca un gestor d'errors fiable.

1.3.2 MPI

La Interfície de Pas de Missatges (conegut àmpliament com MPI [14], sigles en anglès de Message Passing Interface) és un estàndard que defineix la sintaxi i la semàntica de les funcions contingudes en una biblioteca de pas de missatges dissenyada per ser utilitzada en programes que explotin l'existència de múltiples processadors.

El pas de missatges és una tècnica utilitzada en programació concurrent per aportar sincronització entre processos i permetre l'exclusió mútua, de manera semblant a com es fa amb els semàfors.

La seva característica principal és que no necessita memòria compartida, per la qual cosa és molt important en la programació per a sistemes distribuïts.

Els elements principals que intervenen en el pas de missatges són el procés que envia, el que rep i el missatge.

Depenent de si el procés que envia el missatge espera que el missatge sigui rebut, es pot parlar de pas de missatges síncron o asíncron. En el pas de missatges asíncron, el procés que envia, no espera que el missatge sigui rebut, i continua la seva execució, essent possible que torni a generar un nou missatge i enviar-lo abans que s'hagi rebut l'anterior. Per aquest motiu es solen emprar bústies, en què s'emmagatzemen els missatges a espera que un procés els rebi. Generalment emprant aquest sistema, el procés que envia missatges només es bloqueja o per, quan finalitza la seva execució, o si la bústia està plena. En el pas de missatges síncron, el procés que envia el missatge espera que un procés el rebi per continuar la seva execució. Dins el pas de missatges síncron s'engloba la crida a procediment remot, molt popular a les arquitectures client / servidor.

La Interfície de Pas de Missatges és un protocol de comunicació entre ordinadors. És l'estàndard per a la comunicació entre els nodes que executen un programa en un sistema de memòria distribuïda. Les implementacions en MPI consisteixen en un conjunt de biblioteques de rutines que poden ser utilitzades en programes escrits en els llenguatges de programació C, C + +, Fortran i Ada. L'avantatge de MPI sobre altres biblioteques de pas de missatges, és que els programes que utilitzen la biblioteca són portables (atès que MPI ha estat implementat per a gairebé tota arquitectura de memòria distribuïda), i ràpids, (perquè cada implementació de la biblioteca ha estat optimitzada per al maquinari en la qual s'executa).

Amb MPI el nombre de processos requerits s'assigna abans de l'execució del programa, i no es creen processos addicionals mentre l'aplicació s'executa. A cada procés se li assigna una variable que s'anomena **rank**, la qual identifica cada procés, en el rang de 0 a **p-1**, on **p** és el nombre total de processos. El control de l'execució del programa es realitza mitjançant la variable **rank**; la variable **rank** permet determinar quin procés executa determinada porció de codi. En MPI es defineix un **Comunicador** com un conjunt de processos, els quals poden enviar missatges l'un a l'altre, el **Comunicador** bàsic s'anomena **MPI_COMM_WORLD** i es defineix mitjançant un macro del llenguatge C. **MPI_COMM_WORLD** agrupa tots els processos actius durant l'execució d'una aplicació. Les crides de MPI es divideixen en quatre classes:

1. Crides utilitzades per inicialitzar, administrar i finalitzar comunicacions. Aquestes funcions són: **MPI_Init**, **MPI_Comm_size**, **MPI_Comm_rank** y **MPI_Finalize**.

MPI_Init: Permet inicialitzar una sessió MPI. Aquesta funció ha de ser utilitzada abans de trucar a qualsevol altra funció de MPI.

MPI_Comm_size: Permet determinar el nombre total de processos que pertanyen a un Comunicador.

MPI_Comm_rank: Permet determinar l'identificador (**rank**) del procés actual.

MPI_Finalize: permet acabar una sessió MPI. Aquesta funció ha de ser l'última crida a MPI que un programa realitzi. Permet alliberar la memòria usada per MPI.

2. Crides utilitzades per transferir dades entre un parell de processos. La transferència de dades entre dos processos s'aconsegueix mitjançant les crides anomenades **MPI_Send** i **MPI_Recv**. Aquestes crides retornen un codi que indica el seu èxit o fracàs.

MPI_Send: Permet enviar informació des d'un procés a un altre.

MPI_Recv: Permet rebre informació des d'un altre procés.

Ambdues funcions són bloquejants, és a dir que el procés que realitza la crida es bloqueja fins que l'operació de comunicació es completi. Les versions no bloquejants de **MPI_Send** i **MPI_Recv** són **MPI_Isend** i **MPI_Irecv**, respectivament. Aquestes crides inicien l'operació de transferència però la seva finalització ha de ser realitzada de forma explícita mitjançant crides com **MPI_Test** i **MPI_Wait**.

MPI_Wait: És una crida bloquejant i retorna quan l'operació d'enviament o recepció es completa.

MPI_Test: Permet verificar si l'operació d'enviament o recepció ha finalitzat, aquesta funció primer va ha revisar l'estat de l'operació d'enviament o recepció i després retorna.

3. Crides per transferir dades entre diversos processos.

MPI té crides per a comunicacions grupals que inclouen operacions tipus difusió (**broadcast**), recol·lecció (**gather**), distribució (**scatter**) i reducció. Algunes de les funcions que permeten fer transferència entre diversos processos es presenten a continuació.

MPI_Barrier: Permet realitzar operacions de sincronització. En aquestes operacions no existeix cap mena d'intercanvi d'informació. Sol emprar-se per donar per finalitzada una etapa del programa, assegurant-se que tots els processos han acabat abans de donar començament a la següent.

MPI_Bcast: Permet a un procés enviar una còpia de les seves dades a altres processos dins d'un grup definit per un **Communicator**.

MPI_Scatter: Estableix una operació de distribució, en la qual una dada es distribueix en diferents processos.

MPI_Gather: Estableix una operació de recol·lecció, en la qual les dades són recollides en un sol procés.

MPI_Reduce: Permet que el procés arrel reculli dades des d'altres processos en un grup, i els combini en un sol document de dades.

4. Crides utilitzades per a crear tipus de dades definits per l'usuari.

Per definir nous tipus de dades es pot utilitzar l'anomenada **MPI_Type_struct** per crear un nou tipus o es pot utilitzar l'anomenada **MPI_Pack** per empaquetar les dades.

La primera classe de crides permeten inicialitzar la biblioteca de pas de missatges, identificar el nombre de processos (**size**) i el rang dels processos (**rank**). La segona classe de crides inclou operacions de comunicació punt a punt, per a diferents tipus d'activitats d'enviament i recepció. La tercera classe de crides són conegudes com a operacions grupals, que proveeixen operacions de comunicacions entre grups de processos. L'última classe de crides proveeix flexibilitat en la construcció d'estructures de dades complexes. En MPI, un missatge està conformat pel cos del missatge, el qual conté les dades a ser enviades, i el seu embolcall, que indica el procés font i el destí. El cos del missatge en MPI es conforma per tres peces d'informació: **buffer**, **tipus de dada** i **count**. El **buffer**, és la localitat de memòria on es troben

les dades de sortida o on s'emmagatzemen les dades d'entrada. El **tipus de dada**, indica el tipus de les dades que s'envien en el missatge. En casos simples, aquest és un tipus bàsic o primitiu, per exemple, un nombre enter, i que en aplicacions més avançades pot ser un tipus de dada construït a través de dades primitives. Els tipus de dades derivats són anàlegs a les estructures de C. El **count** és un número de seqüència que al costat del tipus de dades permeten a l'usuari agrupar ítems de dades d'un mateix tipus en un sol missatge. MPI estandarditza els tipus de dades primitius, evitant que el programador es preocupi de les diferències que existeixen entre ells, quan es troben en diferents plataformes. L'embolcall d'un missatge en MPI típicament conté l'adreça destí, la direcció de la font, i qualsevol altra informació que es necessiti per transmetre i lliurar el missatge. L'embolcall d'un missatge en MPI, consta de quatre parts: la **font**, el **destí**, el **Comunicador** i una **etiqueta**. La **font** identifica el procés transmissor. El **destí** identifica el procés receptor. El **Comunicador** especifica el grup de processos als quals pertanyen la **font** i el **destí**. L'**etiqueta (tag)** permet classificar el missatge. El camp etiqueta és un enter definit per l'usuari que pot ser utilitzat per a distingir els missatges que rep un procés. Per exemple, es tenen dos processos A i B. El procés A envia dos missatges al procés B, ambdós missatges contenen una dada. Una de les dades és utilitzada per realitzar un càlcul, mentre l'altre és utilitzada per a imprimir-lo en pantalla. El procés A utilitza diferents etiquetes per als missatges. El procés B utilitza els valors d'etiquetes definits en el procés A i identifica quina operació haurà de realitzar amb la dada de cada missatge.

Avantatges

- Estandardització.
- Portabilitat: multiprocessadors, xarxes, heterogenis, etc.
- Bones prestacions.
- Àmplia funcionalitat.
- Existència d'implementacions lliures (mpich, LAM-MPI, etc). L'avantatge de MPI sobre altres biblioteques de pas de missatges, és que els programes que utilitzen la biblioteca són portables (atès que MPI ha estat implementat per a gairebé tota arquitectura de memòria distribuïda), i ràpids, (perquè cada implementació de la llibreria ha estat optimitzada per al maquinari en la qual s'executa).
- Escalabilitat. Els ordinadors amb sistemes de memòria distribuïda són fàcils d'escalar, mentre que la demanda dels recursos creix, es pot afegir més memòria i processadors.

Desavantatges

- L'accés remot a memòria és lent.
- La programació pot ser complicada

1.3.3 INTEL THREADING BUILDING

Intel Threading Building Blocks (Intel TBB [15]) és una biblioteca basada en plantilles per C++ desenvolupada per Intel per facilitar l'escriptura de programes que explotin les capacitats de paral·lelisme dels processadors amb arquitectura multi nucli.

Aquesta biblioteca proporciona algorismes i estructures de dades que permeten al programador evitar en part les complicacions derivades de l'ús dels paquets nadius de gestió de *threads* d'execució en els que la creació, sincronització i destrucció dels *threads* és explícita i dependent del sistema. En comptes d'això, la biblioteca abstreu l'accés als múltiples processadors permetent que les operacions siguin tractades com tasques que es reparteixen automàticament i dinàmicament entre els processadors disponibles mitjançant un gestor en temps d'execució.

Aquesta aproximació fa que Intel TBB s'inclogui en la família de solucions per a la programació paral·lela que permeten desacoblar la programació de les característiques particulars de la màquina.

Intel TBB implementa *task stealing* (robatori de tasques) per balancejar la càrrega de treball sobre els nuclis de processament disponibles per tal d'incrementar l'aprofitament dels nuclis i la escalabilitat dels programes. Inicialment la càrrega de treball es divideix uniformement entre els nuclis de processament disponibles. Si algun d'ells ha complert la seva feina mentre altre encara té una càrrega significativa en la seva cua de tasques, el gestor de tasques reassigna part d'aquest treball al nucli inactiu. Aquesta capacitat de reassignació dinàmica desacobla la programació de la màquina, permetent que les aplicacions escrites amb aquesta biblioteca s'escalin per utilitzar tots els nuclis de processament disponibles sense cap canvi en el codi font o els executables.

Intel TBB, seguint l'exemple de la STL (Standard Template Library - Biblioteca estàndard de plantilles), està basada en l'ús de plantilles ja que s'espera que el polimorfisme en temps de compilació sigui més eficient que el tradicional polimorfisme en temps d'execució.

Intel TBB aporta, entre altres, la següent col·lecció de components per a la programació paral·lela:

- Algorismes bàsics: `parallel_for`, `parallel_reduce`, `parallel_scan`.
- Algorismes avançats: `parallel_while`, `parallel_do`, `pipeline`, `parallel_sort`.
- Contenedors: `concurrent_queue`, `concurrent_vector`, `concurrent_hash_map`.
- Reserva de memòria: `scalable_allocator`, `cache_aligned_allocator`.
- Exclusió mútua: `mutex`, `spin_mutex`, `queuing_mutex`, `spin_rw_mutex`, `queuing_rw_mutex`, `recursive_mutex`.
- Operacions atòmiques: `fetch_and_increment`, `fetch_and_decrement`, `fetch_and_add`, `compare_and_swap`, `fetch_and_store`.
- Presa de temps: `tick_count`.
- Gestor de tasques: `task`.

1.3.4 CELLS (SuperScalar)

OpenMP i MPI, vist anteriorment, es caracteritzen per ser dos models d'àmbit molt general. Adreçats cadascun al seu tipus de sistema (memòria compartida i sistemes distribuïts, respectivament), no estan creats específicament per cap arquitectura, llenguatge, o aplicació

en concret. Aquest model que es presenta en aquest apartat no és tan general, es específicament pel CELL BE.

El CELLSs [16], desenvolupat per membres de la UPC i del BSC¹⁰, és un model de programació per aprofitar l'execució de tasques al CELL. La idea és que el programador anoti funcions que poden ser executades en paral·lel, i en temps d'execució es construeix un graf de dependències per a planificar l'execució en paral·lel de les diferents funcions. No obstant, el que ara ens interessa és el que es fa internament. Quan el sistema operatiu detecta que es vol executar una d'aquestes funcions, esperarà a que hi hagi un SPE lliure (si es que no n'hi ha cap) per a enviar-li la tasca o l'executarà directament al PPE.

CELLSuperScalar / SMPs, conformen un model de programació basat en GridSuperscalar¹¹, però pensats per processadors multi nucli. L'objectiu principal del model és facilitar la programació en aquestes arquitectures per fer més propers els recursos de la informàtica en general i de la Supercomputació en concret, pels programadors no especialitzats. Gràcies a això, el programador es pot desentendre dels detalls referents a l'arquitectura del sistema en què executarà la aplicació. Això implica una programació molt més senzilla ja que no ha d'enfrontar-se a problemes com la gestió de memòria o comunicació entre els *threads*.

L'objectiu del CELLSs / SMPs és proporcionar una forma fàcil d'expressar aplicacions amb Paral·lelisme de manera seqüencial. Això és, una aplicació o programa que efectua una sèrie de càlculs d'un en un i en un ordre determinat.

El model està pensat des del punt de vista de l'encapsulació. Un concepte completament adquirit pels programadors experts i fàcil d'entendre pels que no ho són tant. La idea consisteix a determinar les parts del codi que poden ser executades de manera simultània i posar-les en una funció a la qual anomenarem tasca (TASK).

El programa principal (*thread* pare) registre les tasques en un graf on es gestionen les dependències de dades i s'executen en les altres CPU (*threads* treballadors) quant sigui possible.

Les biblioteca de CELLSs s'encarrega de gestionar quines tasques han estat ja executades i quines estan disponibles. El programador invoca les tasques en l'ordre en que les hagués invocat en una arquitectura amb un processador. El model s'encarrega d'executar una tasca fins que totes les dades que aquesta té com a entrada hagin estat calculades i estiguin disponibles.

Per resumir el funcionament d'una aplicació CELLSs, podríem distingir les següents fases d'execució típiques en una aplicació amb aquest model.

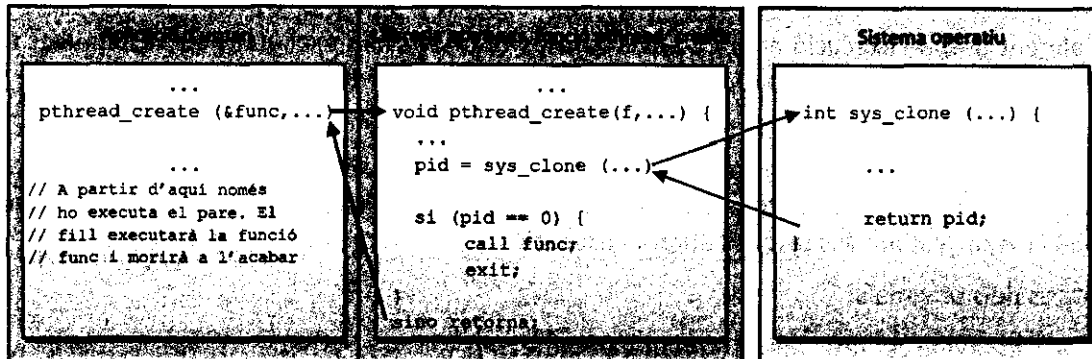
En primer lloc tenim la inicialització, com en tota aplicació cal llegir o generar les dades i preparar les estructures de dades per a l'execució de l'algorisme.

Després comença l'aplicació en si, el programa va invocant tasques. En algun moment és possible que sigui necessari establir punts de sincronització, esperar que certes tasques acabin

¹⁰ Barcelona Supercomputing Center

¹¹ La tecnologia informàtica Grid té com a objectiu oferir eines i mecanismes que permetin l'intercanvi, la selecció, i l'agregació d'una àmplia varietat de recursos informàtics distribuïts geogràficament de forma transparent.

A la següent figura es presenta un petit esquema d'aquesta situació, mostrant la crida que fa el programador a la part d'aplicació, el salt a la llibreria, l'entrada al sistema operatiu (*trap*, en color vermell) i la crida a la funció especificada pel programador.



Per altra banda, l'operació per a destruir un *pthread* (*pthread_exit*) l'ha d'inserir el programador al final de la funció que vol que executi el *pthread*, i el que fa és una crida a sistema *exit*, que s'encarrega de finalitzar un *thread*, independentment de la resta de *threads* que hi hagi al grup de la víctima, a diferència de la crida a sistema *exit*, que finalitza tot un procés amb el seu grup de *threads*, és a dir, tota una aplicació *multithread*. Aquesta darrera és la que insereix automàticament el compilador al final d'un programa *main*. Per tant, l'avantatge d'utilitzar *threads* és que ens garanteixen que abans d'executar la funció es farà una crida a sistema, i al finalitzar també, i podrem prendre les decisions que ens interessin en aquests punts.

La biblioteca ofereix tres mecanismes de sincronització:

- **Mutexs:** Bloqueja l'accés a les variables per altres *threads*. Això imposa un accés exclusiu d'un *thread* a una variable o un conjunt de variables. Funcions:
 - pthread_mutex_destroy:** Destruïx un objecte mutex, alliberant el recursos que tenia.
 - pthread_mutex_init:** Inicialitza l'objecte mutex apuntat per mutex d'acord amb les característiques especificades en el mutex *mutexattr*. Si *mutexattr* és NULL, s'utilitzen els atributs per defecte.
 - pthread_mutex_lock:** Bloqueja un mutex determinat. Si el mutex està desbloquejat, és bloquejat i es converteix en propietat del *thread* que l'ha bloquejat.
 - pthread_mutex_unlock:** Desbloqueja un mutex determinat. El mutex se suposa que està bloquejat i es propietat el *thread* que executa *pthread_mutex_unlock*.

Exemple:

| | | | |
|---|-------------|---|---|
| <pre> int counter=0; /* Function C */ void functionC() { counter++ } </pre> | | <pre> /* Note scope of variable and mutex are the same */ pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER; int counter=0; /* Function C */ void functionC() { pthread_mutex_lock(&mutex1); counter++ pthread_mutex_unlock(&mutex1); } </pre> | |
| counter = 0 | counter = 0 | counter = 0 | counter = 0 |
| counter = 1 | counter = 1 | counter = 1 | Thread 2 bloquejat. Thread 1 té en exclusiva l'ús de la variable counter |
| | | | counter = 2 |

- **Joins:** Fer que un *thread* esperi fins que altres són complets, funció **pthread_join**.
- **Condicció de les variables (signals):** És una variable de tipus **pthread_cond_t** i s'utilitza amb les funcions adequades per a l'espera i després, continuació del procés. El mecanisme permet que la variable de condició del *threads*, suspengui l'execució i l'abandonament del processador fins que alguna condició sigui verdadera.

Creant /Destuïnt

```

pthread_cond_init
pthread_cond_t cond
pthread_cond_destroy

```

Esperant la condició:

```

pthread_cond_wait
pthread_cond_timewait

```

Despertant un *thread* sobre la base de la condició

```

pthread_cond_signal
pthread_cond_broadcast

```


2 MOTIVACIÓ DEL PROJECTE I OBJECTIUS

En aquest segon capítol es fa una introducció al tema del treball d'aquest projecte, de les eines que s'usaran, així com dels objectius que es volen assolir.

La motivació del projecte es entrar en contacte amb el grup de recerca de nanos que ha dissenyat una software cache híbrida des de zero per al processador CELL BE.

L'objectiu principal d'aquest projecte es dissenyar i implementar un fase de generació de codi per a aquest processador multi nucli per executar programes sobre la software cache híbrida i en les SPEs del processador, dintre de la infraestructura del compilador Nanos Mercurium C/C++ desenvolupada també pel mateix grup.

Nanos Mercurium C/C++ és un compilador font a font inicialment desenvolupat per omplir el buit de suport C++ en OpenMP. Va resultar ser útils per altres projectes relacionats amb les transformacions font a font. Proporciona un entorn de programació que facilita el desenvolupament de les fases per al compilador.

En la nova fase implementada es permetrà a partir de noves directives **pragma**, que un codi escrit per un programador, es transformi en el codi necessari que permeti maximitzar els recursos del CELL BE i la software cache híbrida junts. D'aquesta manera evitem que el programador hagi de conèixer en profunditat el compilador, la software cache híbrida o l'arquitectura del CELL BE.

A continuació s'expliquen les tasques que s'han de completar per assolir aquest objectiu. En aquesta llista es mencionen conceptes que no han sigut explicats encara. En els capítols següents es tractaran amb més profunditat:

- Aprendre l'arquitectura del processador multi nucli CELL BE
- Aprendre la software cache híbrida del grup de recerca de nanos.
- Aprendre l'estructura i disseny del compilador Nanos Mercurium C/C++ del grup de recerca de nanos.
- Dissenyar el nou mòdul per desenvolupar la part del nostre projecte dintre del compilador.
- Implementar el nou mòdul: Desenvolupament del mòdul dintre del compilador.
- Avaluació: Disseny del joc de proves per provar el correcte funcionament de tot el mòdul.

En els capítols següents veurem cadascun d'aquests aspectes.

La resta de la memòria veurem com hem assolit tots aquests objectius.

3 TREBALL RELACIONAT

En aquest capítol s'analitza el treball relacionat i les propostes existents a dia d'avui, al camp del processador multi nucli CELL BE, dels compiladors de recerca en general i específics per al CELL BE, els mètodes de programació, les optimitzacions, altres implementacions de software cache,...

3.1 PLATAFORMES DE COMPILACIÓ

El compilador Nanos Mercurium C/C++ no es l'únic compilador de recerca, n'hi ha d'altres, com per exemple els compiladors com el TRIMARAN[18] o el ORC[19]. Els dos són compiladors lliures destinats a investigadors i educadors.

Per exemple el projecte del compilador lliure de recerca (ORC) proporciona una infraestructura de compilació de font oberta pel processador ItaniumTM (IA-64) per a la comunitat de recerca. Aquest projecte és una col·laboració entre Intel Corp i l'Acadèmia Xinesa de Ciències. Volen proporcionar una infraestructura comuna per fomentar i facilitar la recerca i l'arquitectura del compilador. En el disseny del ORC, destaquen en els aspectes següents: compatibilitat amb altres eines de codi obert, la solidesa de tota la infraestructura, la flexibilitat i la modularitat ràpida de prototipus de noves idees.

I en canvi TRIMARAN es un compilador per la recerca en arquitectures ILP¹², anomenada HPL-PD. El sistema és actualment orientat a través de EPIC¹³ (Explicitly Parallel Instruction Computing) i suporta una varietat de compiladors de recerca, incloent les instruccions de scheduling, assignació de registres, etc.

Pel que fa a compiladors per treballar amb el CELL BE a part del Nanos Mercurium C/C++ usat en el nostre projecte existeixen altres compiladors o eines pel CELL BE: GNU toolchain, IBM XLC i GNU ADA.

Les utilitats del GNU toolchain, inclosos els compiladors, l'assemblador, el linkador i eines diverses, estan disponibles tant per la unitat del processador PowerPC (PPE) com pel conjunt d'instruccions del SPU. També en les eines de GNU trobem una implementació del compilador GNU ADA.

¹² (Instruction Level Parallel) - Més d'una operació executada per cicle de rellotge en una sola CPU.

¹³ ILP sota control del compilador:

- Una sola instrucció pot contenir moltes operacions.
- El compilador determina i especifica el funcionament de les dependències entre operacions que poden executar-se simultàniament.

3.2 PROGRAMABILITAT I MEMÒRIES LOCALS

A part de la software cache híbrida del grup de recerca nanos, també hi ha altres opcions com la Flexicache[20], que també proposa un disseny de software cache. La principal qüestió és com tractar els salts directes/indirectes. Els mecanismes que proposa aquesta software cache tot i que cau en tècniques de software cache, són molt diferent a les presentades en la software cache híbrida explicada i utilitzada en aquest projecte.

El treball en HotPages/FlexCache és similar al de la software cache híbrida, però no es tant potent i simple. Es basa en mecanismes de complexitat considerable i les necessitats d'anàlisi d'un compilador amb cert grau de precisió. A més a més, aquest no proporciona una solució on el compilador elimina totalment el control al voltant del codi de les referències de memòria.

La software cache també s'utilitza com a tècnica per reduir la potència i la reducció d'àrea dels systemes embedded [21][22]. Explícitament els sistemes manejats per software cache es postulen com una solució per consideracions de potència en dispositius informàtics.

3.3 MODELS DE PROGRAMACIÓ

A part del models de programació vistos en aquesta memòria, també n' existeixen d'altres com el model MapReduce, que és un simple i flexible model de programació paral·lela proposat per Google per al processament de dades a gran escala en un entorn de computació distribuïda. Hi ha un grup de recerca a la universitat de Wisconsin-Madison [23], que presenten un disseny i implementació del MapReduce per a l'arquitectura CELL BE.

Eichenberger [24] descriu una varietat de tècniques per optimitzar els compilador del CELL BE extraient paral·lelisme i simplificant la gestió de memòria. Els temes inclouen extracció de paral·lelisme SIMD¹⁴, codi automàtic i particionament de dades de nuclis usant pragmas OPENMP [25], i la implementació d'una software cache gestionada amb el suport d'una memòria compartida simple en els SPEs.

Blagojevic [26] també descriu unes tècniques de scheduling per diferents granularitats de paral·lelisme pel processador CELL BE.

També tenim el model de programació ja explicat, el CELLs [16] que és un model de programació per aprofitar l'execució de tasques al CELL BE. La idea és que el programador anoti funcions que poden ser executades en paral·lel, i en temps d'execució es construeix un graf de dependències per a planificar l'execució en paral·lel de les diferents funcions.

¹⁴ SIMD (Single Instruction, Multiple Data) és una tècnica emprada per aconseguir el paral·lelisme a nivell de dades, com en processador vectorial.

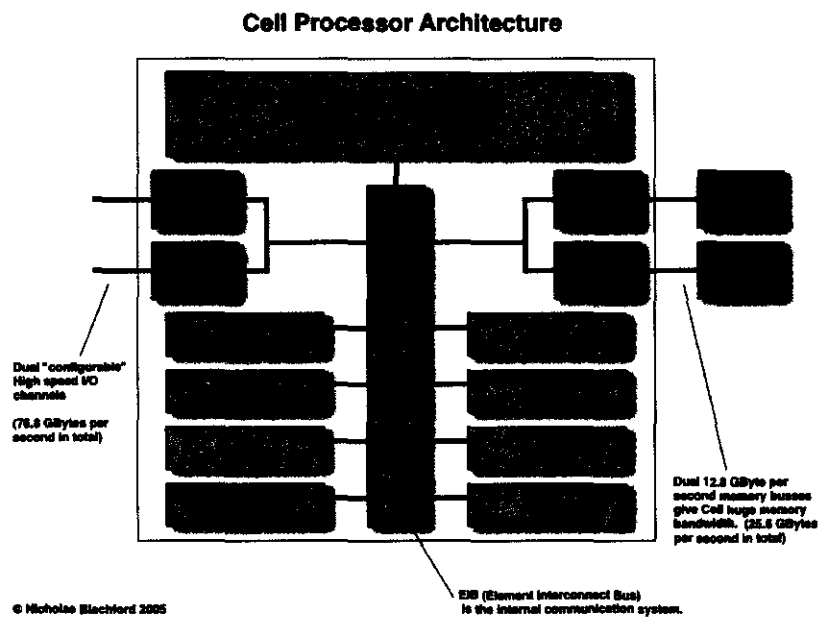
4 PROCESSADOR CELL BE

En aquest capítol veurem en detall el disseny i l'estructura del processador CELL BE [27] usat en el projecte, els models de programació que suporta, juntament amb les eines de suport que té tant en compilació com en execució.

4.1 ESTRUCTURA

Els components d'aquest processador són:

- 1 Power Processor Element (PPE).
- 8 Synergistic Processor Elements (SPEs). Elements de procés sinèrgic, terme de marketing referit a unitats de co-processament vectorial.
- Bus d'Interconnexió dels Elements (EIB).
- Controlador d'Accés Directe a Memòria (DMAC).
- 2 Controladors de memòria Rambus XDR.
- Una interfície Rambus FlexIO (Input / Output).

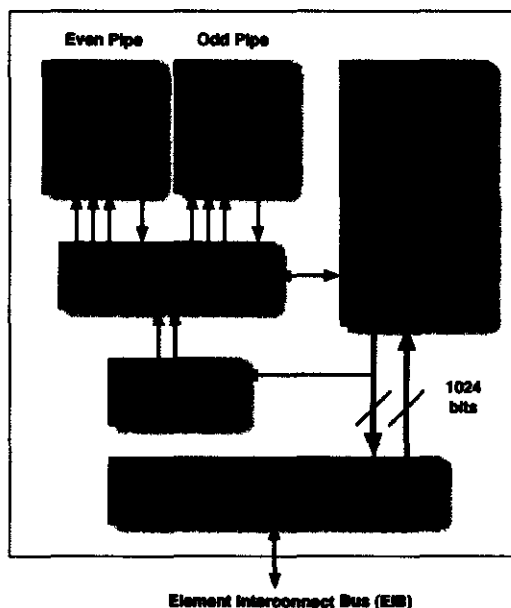


El PPE és el nucli principal, aquest s'encarrega de coordinar el treball de tots els 8 nuclis (SPEs), mitjançant la tecnologia SMT (*Simultaneous Multi-Threading*), equivalent al *HyperThreading* de Intel. SMT, és una tècnica per millorar l'eficiència global de CPUs superescalars amb maquinari de *multithreading*.

EL PPE està basat en l'arquitectura PowerPC de 64 bits, té 32 KB de cache L1 i 512 KB de cache L2, té també tecnologia de doble *thread* i pot executar dues instruccions per *thread*. Aquest processador està fet com els RISC¹⁵ clàssics, o sigui no és com els PowerPC tradicionals, per això no té implementat la predicció de salts i l'execució d'instruccions és en ordre; el que estalvia una quantitat considerable de transistors, passant tot aquest treball al compilador.

Cell SPE Architecture

Each SPE is an independent vector CPU capable of 22 GFLOPs or 32 GOPs (32 bit @ 4GHz.)



Els SPE, els processadors auxiliars, són unitats de càlcul vectorial. Aquests poden executar fins a dues instruccions per cicle. Cada SPE té 128 registres de 128 bits cadascun, 4 unitats de coma flotant, 4 unitats aritmètiques senceres i una memòria local de 256 KB (aquesta memòria és SRAM com les memòries cache, però no és una d'elles, és a dir, no es tracta d'una cache, sinó d'un bloc de 256K en el qual el programador pot fer el que vulgui). Una memòria cache és una memòria que es carrega amb anticipació del que presumiblement el programari necessiti. Al no utilitzar memòria cache, es simplifica el disseny del SPE. Els SPEs tenen memòria local per prendre les dades que requereixen més ràpidament.

El bus d'interconnexió d'elements, EIB, està compost per 4 canals de dades de 128 bits i permet la comunicació entre tots els elements del processador, també permet carregar i moure 16GB de dades per segon cap al CELL BE i cap a fora del CELL BE respectivament. Per mantenir ple aquest ample de banda, el processador CELL BE utilitza en els seus controladors

¹⁵ *Reduced Instruction Set Computer*. Ordinador amb conjunt d'instruccions reduïdes.

de I/O (entrada/sortida), la memòria dissenyades per l'empresa Rambus (coneguda per haver fabricat les memòries RAM més ràpides pel Pentium 4 (les RIMM, que no van tenir però acceptació en el mercat). La memòria XDR de Rambus és bastant ràpida, arribant a velocitats molt superiors a les memòries convencionals.

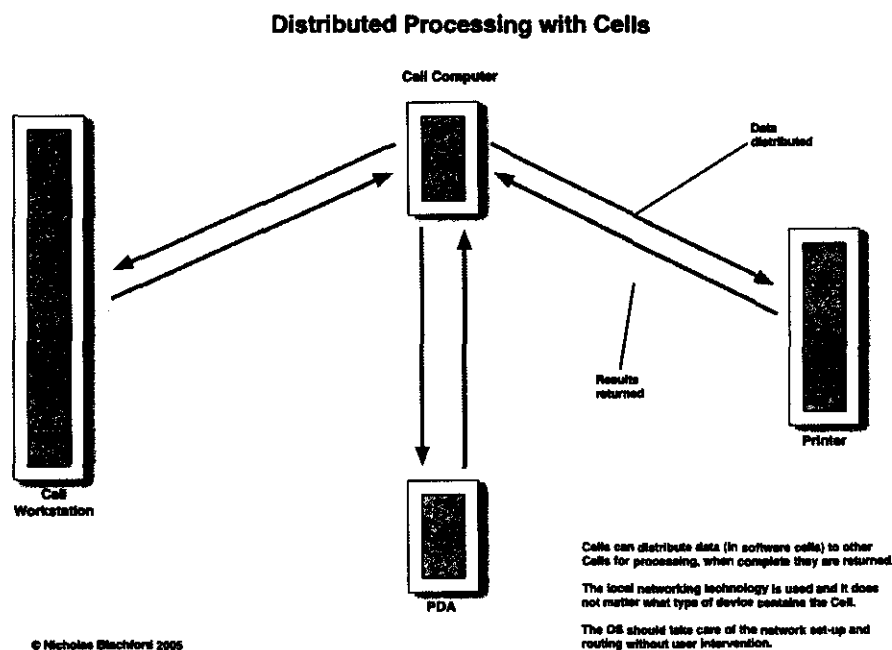
La velocitat a la qual arriba és de 4.8Ghz. La tecnologia FlexIO, també de Rambus, és una interfície d'entrada i sortida bastant ràpida. Està constituït per 12 connexions punt a punt unidireccionals de 1byte, 7 d'aquestes connexions són de sortida i 5 són d'entrada. El FlexIO pot tenir una velocitat des 400Mhz fins 8GHz.

El xip CELL BE ha estalviat molts transistors en no implementar: memòria cache per als SPEs, execució fora d'ordre, predicció de salts, etc., deixant tot aquest treball al compilador; amb la finalitat de posar més processadors (SPEs) la qual cosa augmenta el poder de processament, i a més el xip és més senzill i gasta menys energia.

De totes maneres el xip és un monstre amb els seus 234 milions de transistors, la majoria dedicats al poder de processament per lo anteriorment comentat, i com utilitza molts nuclis, la generació de calor es dispersa per tot el processador.

Altres característiques que té el CELL BE és que és escalable, va ser dissenyat per poder treballar amb altres CELL BE. Un PPE d'un CELL BE té el potencial de comunicar-se amb un PPE o un SPE d'un altre CELL BE que es trobi en la mateixa placa mare, a la mateixa xarxa o en qualsevol part del món si tots dos estan connectats a Internet.

Podem veure en la figura següent més clarament aquest concepte:



Per últim, mencionar que entre el CELL BE i els processadors de dos o més nuclis que han de llançar Intel i AMD també existeix una diferència; en aquests últims, els nuclis tenen capacitats predictives/especulatives, capacitats de les que manca els nuclis del CELL BE. Això significa que els nuclis del CELL BE necessiten un volum de circuiteria molt menor, amb lo que es poden integrar molts més nuclis dintre del processador; com a contrapartida, els nuclis dels

processadors de Intel i AMD són apreciablement més ràpids. La conseqüència pràctica es que els programes existents (que consten d'un thread d'execució) seran més ràpids en un processador Intel o AMD i un programa multi-thread serà més ràpid en el CELL BE.

Una cosa molt important també, es el fet que el compilador per al CELL BE haurà de generar dues versions diferents pel codi, una per si executa el codi al PPE i l'altre si l'executa al SPE, degut a que els accessos a memòria canvien si s'executa a un lloc a l'altre, per tant el compilador sempre generar aquestes dues versions diferenciades.

4.2 SUPORT DE COMPILACIÓ I EXECUCIÓ DEL PROCESSADOR CELL BE

La dificultat de programació és un dels principals obstacles per l'àmplia acceptació dels sistemes multi nucli, sense suport del maquinari per a la transferència transparent de dades entre memòries locals i globals.

La proposta d'arquitectures, com el processador CELL BE, mostren una tendència cap al disseny de nous subsistemes de memòria, proporcionant als nuclis de càlcul memòries locals on la transferència de dades cap a/des de memòria principal són realitzades explícitament sota el control del programador.

En termes de programació, el CELL BE i els sistemes per igual poden requerir esforços considerables de programació i optimització de codi. Les solucions general basades en compilador sovint són difícils d'implementar a causa de la manca d'informació suficient en el temps de compilació per generar el codi correcte i eficient.

Alguns models que s'apliquen, ja explicats, per aquestes transformacions en el CELL BE són:

- OPENMP
- Pthreads
- CELLS (SuperScalar)

En les màquines basades en memòria cache, les optimitzacions més significatives que tenen el major guany en quan a rendiment són les que milloren el programa que utilitza la jerarquia de cache.

El principal problema del CELL BE es portar les dades a executar a un SPE, per solucionar aquest problema hi ha dues tècniques:

- Tiling
- Software cache

4.2.1 TILING

Loop tiling, també conegut com **blocking**, és una optimització de bucle utilitzada pels compiladors per a l'execució de determinats tipus de bucles més eficientment.

Consisteix en fer particions d'un espai d'iteracions d'un bucle en petits trossos o blocs, per tal d'ajudar a garantir que les dades utilitzades en un bucle es quedin en la memòria cache fins que es reutilitzin. El particionament de l'espai d'iteracions del bucle duu a un particionament dels grans arrays a blocs petits, per tant adequat accedir a elements del array de la mida de la memòria cache, millorant l'augment de la reutilització i l'eliminació dels requisits de la mida de cache.

A continuació veiem un exemple:

Original: Programa de multiplicació de matrius

```
for (i=0; i<N; i++)
  for (j=0; j<N; j++)
    c[i] = c[i] + a[i,j]*b[j];
```

Després de loop tiling 2*2:

```
for (i=0; i<N; i+=2)
  for (j=0; j<N; j+=2)
    for (ii=i; ii<min(i+2,N); ii++)
      for (jj=j; jj<min(j+2,N); jj++)
        c[ii] = c[ii] + a[ii,jj]*b[jj];
```

En el bucle original l'espai d'iteració és N per N. L'accés a la part de l'array a[i, j] és també N per N. Si N és massa gran i la mida de la memòria cache de la màquina és massa petit, accedir als elements de l'array en una iteració del bucle (per exemple, i = 1, j = 1 a N) poden creuar les línies de cache, causant molts misses de cache. Per tant si fem loop tiling 2*2 evitem això.

4.2.2 SOFTWARE CACHE

4.2.2.1 SOFTWARE CACHE TRADICIONAL DEL CELL BE

La software cache pot ajudar a aplicar una programació paral·lela de memòria compartida quan el model de dades de les entitats de referència no es pot predir fàcilment.

Des del punt de vista d'un programador, l'accés a les dades utilitzant software cache és similar a usar les instruccions ordinàries **LOAD** i **STORE**, a diferència de la típica interfície DMA¹⁶ de la SPU per a la transferència de dades.

¹⁶ DMA - Direct Memory Access, en català Accés directe a memòria.

La software cache promet augmentar la programabilitat i el rendiment en determinades aplicacions com en les referències de memòria irregulars en arquitectures multi nucli del processador CELL BE on el xip de memòria és un recurs molt valuós.

Arquitectures heterogènies com el CELL BE tenen restriccions en la quantitat de memòria disponible en el xip per a ells. Les SPUs en el CELL BE estan equipades amb 256 KB de memòria local. El programador en general fa un accés directe a memòria (DMA) per SPUs.

Normalment el buffer cíclic s'utilitza per amagar la superposició de la latència computacional de la DMA en les comunicacions. Tanmateix, en aplicacions com ara amb les referències de memòria irregulars pot arribar a ser molt difícil.

La software cache pot arribar a ser molt útil tant en termes d'augment de programabilitat i en un major rendiment per aquests escenaris.

Veurem ara la necessitat de la software cache en aquest exemple de codi executat per la SPU.

```
for(i = 0; i < 100000; ++i)
{
    a[i] = b[i] + c[i] * d[f(i)];
}
```

En aquest cas, **b** i **c** poden ser obtingudes amb anterioritat, però l'esquema d'accés a **d** no es pot predir. **d [f (i)]** han de ser obtinguts en cada iteració, la qual cosa resulta una enorme disminució del bucle. Cada accés a **d** requereix un accés d'alta latència a la memòria principal. Mentre les SPEs no tenen el hardware cache, és possible aplicar una petita software cache referint-se explícitament a ella. Per exemple, l'anterior exemple es pot aplicar de la següent manera:

```
for(i = 0; i < 100000; ++i)
{
    t = cache lookup(d[f(i)]);
    a[i] = b[i] + c[i] * t;
}
```

Un exemple d'implementació de `cache_lookup` pot ser com aquest:

```
inlinevector cache lookup(addr)
{
    /* obtenir el valor si no el tenim */
    if (cache directory[addr&key mask] != (addr&tag mask))
        miss handler(addr)
    /* retornar el valor */
    return cache data[addr&key mask][addr&offset mask];
}
```

Les funcions de software cache afegixen sobrecarrega (càlculs de control) en comparació amb les transferències de dades ordinàries a la DMA, per tant, la transferència de la DMA és preferible en cas que el patró d'accés a les dades sigui seqüencial.

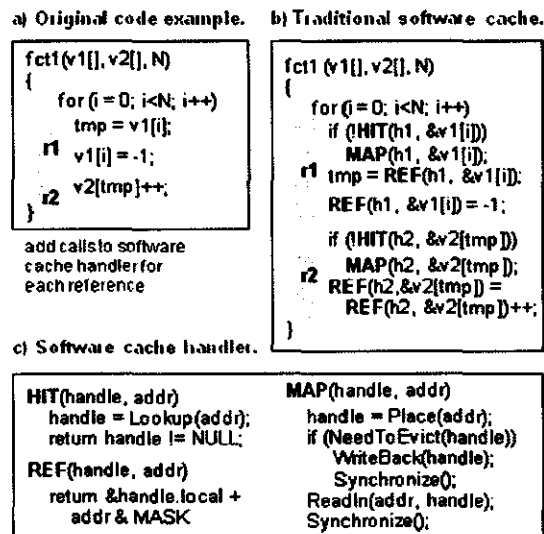
Els dos reptes en l'ús de la software cache de manera eficaç serien els següents:

- La quantitat d'espai ocupat per la software cache ha de ser el mínim possible ja que l'emmagatzematge local és un recurs preciós per a la SPU.
- Ser útil en termes de rendiment: els cicles perduts en buscar i altres treballs de càlcul han de ser compensats pels cicles adquirits per garantir que la línia de cache requerida es gairebé tot el temps present a la cache.

La software cache té els següents avantatges:

- Millor rendiment en algunes aplicacions degut al principi de localitat de referència per l'estalvi de les transferències de dades redundant si les dades corresponents ja està en local.
- La topologia reconfigurable i el comportament al qual implica que pot ser fàcilment optimitzada per a que coincideixi amb els patrons d'accés a dades (a diferència de la majoria de hardware cache).

Veiem un altre exemple per l'execució de arquitectures software cache tradicional:



El codi original en l'aparat a) de la figura mostra dues referències, v1[i] i v2[tmp]. Suposant que els arrays estan en memòria global, l'aparat b) de la figura representa el mateix codi amb totes les crides requerides per al controlador de la software cache tradicional. Abans de cada referència r1 i r2, es necessari comprovar si la dada es resident a la software cache (usant la macro HIT). Quan no sigui present, cridarem el controlador de miss (usant la macro MAP). Com es mostra en l'aparat c) de la figura, el controlador de miss MAP primer busca una línia de cache per treure-la fora, possiblement escrivint la línia a memòria global, i llavors carrega la línia sol·licitada. Una vegada la dada ha arribat, la dada pot ser accedida, usant la macro REF, en mode de lectura o escriptura.

Evidentment, el codi transformat a l'apartat **b)** de la figura es lluny de ser òptim, especialment per les referències d'alta localitat espacial com $v1[i]$ amb $i=0..N$. Per referències d'alta localitat, és trivial calcular el nombre de dades útils presents a les actuals línies de cache. En altres paraules, es pot calcular fàcilment el nombre de iteracions de bucle pels quals les actuals línies de cache poden proporcionar dades per les seves referències. Donat tal nombre de iteracions sense un MISS, voldríem iterar sobre aquests càlculs sense cap software cache. De totes maneres, aquesta transformació de codi no es possible amb una interfície de software cache tradicional. Primer, hem de ser capaços de trobar una línia de cache a la cache d'emmagatzematge, alliberant aquesta només quan totes les referències d'alta localitat estiguin fets amb ella. Segon, la cache ha de tenir almenys una línia de cache per diferents referències d'alta localitat en el bucle, si volem eliminar tots els codis de control del bucle més interior. Així necessitem tenir algun control de la geometria de la cache.

El codi de l'apartat **b)** de la figura es també subòptim amb respecte la segona referència, $v2[tmp]$, on tmp es igual que el valor original de $v1[i]$. Aquest patró d'accés correspon a un accés indirecte, el qual típicament exposa molt poca localitat de dada. El perquè d'aquest esquema d'accés irregular és perquè la dada es molt poc probable que es trobi a cache. El rendiment només es pot aconseguir mitjançant l'exercici de la major quantitat d'accessos irregulars com sigui possible en paral·lel (sense sincronització o bloqueig de DMA) per la màxima superposició de les comunicacions. Altre cop, les interfícies de cache tradicionals no són adequades, elles típicament proporcionen només una petita associativitat, que limita directament el nombre d'accessos irregulars concurrents.

El CELL SDK 3.0 d'IBM proporciona aquesta software cache tradicional [28][29] com una macro d'una biblioteca que pot ser usat per programadors d'aplicacions de dues maneres: un mode síncron i un altre asíncron. La software cache pot ser configurada en base l'associativitat, mode d'accés (només lectura o de lectura-escritura), la mida de la línia de cache, nombre de línies i tipus de dades.

El mode síncron (o mode segur) proporciona al programador un conjunt de funcions per l'accés a les dades simplement utilitzant la direcció efectiva de les dades. La llibreria de software cache realitza la transferència de dades entre la LS i la memòria principal de forma transparent per al programador i gestiona les dades que ja són a la LS.

La interfície asíncrona (o mode no segur) permet al programador ocultar la latència d'accés a memòria per la superposició de la transferència de dades i quantitat de càlcul. Això proporciona una manera més eficient d'accedir a LS en comparació al mode segur. La software cache proporciona funcions de mapeig d'adreces efectives a adreces de LS. El programador ha d'usar les adreces de LS més tard per accedir a les dades, a diferència en mode segur on s'utilitzen les adreces efectives.

També hi ha una disposició per definir múltiples caches, cadascun configurat diferentment per adaptar-se a les necessitats dels programadors.

Veient tots els problemes que no soluciona una software cache tradicional, veiem la següent implementació i disseny d'una software cache híbrida que permet solucionar els problemes vistos.

4.2.2.2 SOFTWARE CACHE HÍBRIDA DEL GRUP DE RECERCA NANOS

En la software cache híbrida [29] dissenyada per aquest grup proposen un ordre jeràrquic, una arquitectura híbrida de software cache que han dissenyat des de zero que classifica en temps de compilació els accessos a memòria en dues categories, d'alta localitat i irregular. Degut a que les optimitzacions del compilador van dirigides a aquests dos models tenen diferents objectius i necessitats; i han dissenyat dues estructures cache que millor responen a aquests diferents patrons d'accés i els requisits d'optimització. En particular, el seu disseny inclou:

- (1) Una memòria d'alta localitat amb una variable de configuració, línies que poden ser dipositades, i un sofisticat mecanisme de retorn d'escriptura, i
- (2) una cache transaccional ràpida, totalment amb cerques associatives, línies curtes, i una eficaç política d'escriptura.

El seu enfocament es dirigeix cap a les referències de memòria d'una de les dues estructures específiques de cache per a l'accés als seus respectius patrons.

Les estructures específiques de cache que permeten l'optimització d'alt nivell en les optimitzacions del compilador serien: desenrotllar bucles, reordenar referències de cache, i / o transformacions dels bucles per tal d'eliminar pràcticament la sobrecarrega (càlculs de control) de la software cache en el bucle més intern.

L'avaluació que han fet del rendiment d'aquesta software cache indica que les millores a causa de l'estructura optimitzada de software cache combinat amb la proposta d'optimitzacions de codi es tradueix en un factor d'acceleració de 3,5 a 8,4 en comparació amb un enfocament de software cache tradicional.

En l'apartat següent s'explicarà aquesta software cache en molt més detall.

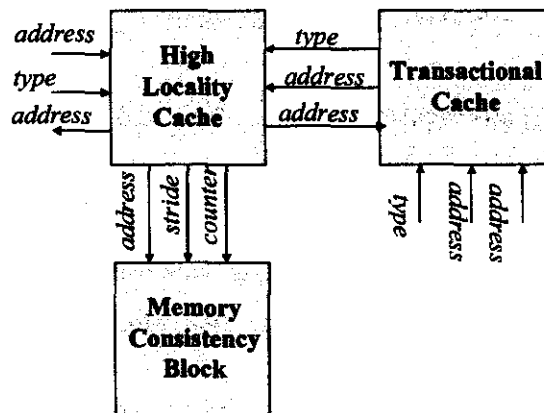
5 SOFTWARE CACHE HÍBRIDA

En aquest capítol veurem en detall el disseny i implementació de la software cache híbrida [29] usada en el projecte del grup de recerca nanos, juntament amb les transformacions de codi per aquesta software cache.

5.1 DISSENY DE LA SOFTWARE CACHE HÍBRIDA

Començarem detallant primer de tot la software cache del grup de recerca nanos utilitzada ja que la generació de codi d'aquest projecte es per aquesta software cache.

Començarem descrivint en aquesta secció, el disseny de la jeràrquica software cache híbrida [18] utilitzada. La Figura següent mostra l'arquitectura d'alt nivell de la seva software cache.



Les referències a memòria exposen un alt nivell de localitat que s'assignen pel compilador a la Cache d'Alta Localitat (*High-Locality Cache*), i els altres accessos irregulars s'assignen a la Cache Transaccional (*Transactional Cache*). La consistència de memòria (*Memory Consistency Block*) s'aplica a les estructures de dades necessàries per mantenir una coherència relaxada del model d'acord amb el model de memòria OpenMP [30].

La cache és accessible a través d'un sol bloc, ja sigui de la Cache d'Alta Localitat o la Cache Transaccional. Ambdues caches són coherents entre si. L'enfocament híbrid és jeràrquic en el qual la Cache Transaccional es obligada a comprovar les dades en la Cache d'Alta Localitat durant el procés de recerca (lookup).

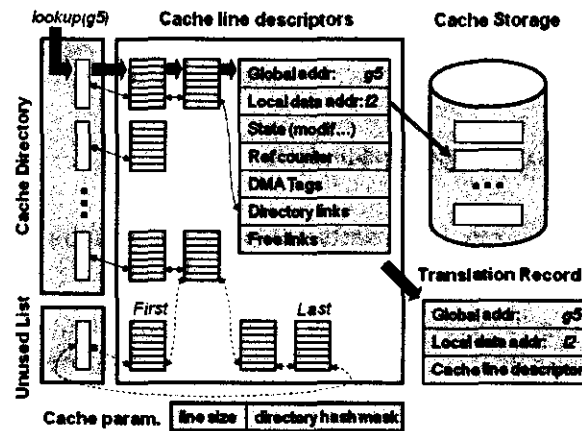
5.1.1 Cache d'Alta Localitat

La Cache d'Alta Localitat permet optimitzacions del compilador per referències de memòria que exposin un alt grau de localitat espacial, és a dir, accessos d'alta localitat. Està dissenyat

per línies de cache usant comptadors de referències explícites, lliurant bons coeficients d'èxit (HIT), i maximitzant la superposició entre els càlculs i la comunicació.

5.1.1.1 Estructures de la Cache d'Alta Localitat

La Cache d'Alta Localitat està composta per les següents sis estructures de dades:



- 1) *Cache Storage* per emmagatzemar dades d'aplicació
- 2) *Cache Line Descriptors* per descriure cada línia en la cache
- 3) *Cache Directory* per recuperar les línies
- 4) *Cache Unused List* per indicar les línies que poden ser reutilitzades
- 5) *Cache Translation Record* per preservar cada referència d'adreça resolta pel lookup a la cache.
- 6) *Cache Parameters* per registrar els paràmetres de configuració global.

La *Cache Storage* és un bloc d'emmagatzematge de dades organitzades com N línies de cache, on N es el total d'emmagatzematge de la cache dividit per configuració, es poden guardar entre 16 i 128 línies de cache de mides des de 512 a 4K bytes, dins dels 64KB que té la *Cache Storage*.

Cada línia de cache s'associa amb un únic *Cache Line Descriptor* que descriu tot el que cal saber sobre línia.

La *Cache Translation Record* conserva la informació generada pel procés de lookup i més tard es usat quan s'accedeix a les dades per la referència actual. Conté 3 elements, l'adreça base global de la referència original, l'adreça base local en la cache d'emmagatzematge (*Cache Storage*), i un punter a la línia del descriptor de cache (*Cache line descriptors*).

Han implementat una eficient estructura de cache plenament associativa usant l'estructura de directori de cache (*Cache Directory*). Conté un nombre prou elevat de llistes de links dobles (128 a la seva aplicació), on cada llista pot contenir un nombre arbitrari de descriptors de línia

de cache. Una funció de hash s'aplica a l'adreça base global per localitzar aquesta en la llista corresponent, que és llavors travessada per trobar una coincidència possible (*match*).

La *Cache Unused List* és una llista de links dobles que conté tots el descriptor de la línia de cache en desús.

La *Cache Parameters* inclou paràmetres com ara el *Cache Directory Hash Mask*, una màscara utilitzada per la *Cache Directory* per associar una adreça base global amb la seva llista específica vinculada.

5.1.1.2 Model operatiu de la Cache d'Alta Localitat

El model operatiu de la Cache d'Alta Localitat és integrat per totes les operacions que s'executen sobre l'estructura de cache i implementa les operacions primitives de recerca (*lookup*), col·locació (*placement*), comunicació i de mecanismes de sincronització i de coherència (*consistency*).

L'operació de *lookup* per a un referència donada *r*, un registre de traducció *h*, i una adreça global *g* es divideix en dues fases diferents. La primera fase comprova si la adreça global *g* es troba en la línia de cache actual pel registre de traducció *h*. Quan aquest és el cas, tenim un èxit (*HIT*) i ja està. En cas contrari, tenim una situació en la qual el registre de traducció necessitarà apuntar a una nova línia de cache en la memòria local. A continuació el procés de *lookup* entra en la seva segona fase. La segona fase accedeix a la *Cache Directory* per determinar si la referència de la línia de cache és ja resident a la *Cache Storage*. Quan tenim un èxit (*HIT*), actualitzem el registre de traducció *h* i ja està. En cas contrari, es produeix una falta (*MISS*) i continuarem amb les operacions de col·locació i de comunicació.

El comptador de referències es sovint actualitzat durant el procés de *lookup*. Sempre que un registre de traducció deixa d'apuntar a una línia específica d'un Descriptor de cache, el comptador de referència d'aquest Descriptor es decrementa en un. De la mateixa manera, sempre que un registre de traducció comenci apuntant a un nou descriptor de línia de cache, el comptador de referències d'aquest nou descriptor s'incrementa en un.

La col·locació de codi es invocat quan una nova línia és requerida. Les línies lliures son descobertes quan els seus comptadors de referències arriben a 0. Les línies lliures són immediatament insertades al final de la llista de línies de la *Unused Cache*. Les línies modificades són llavors tornades a memòria global. Quan una nova línia és necessària, agafem la línia de sobre de la llista de línies de la *Unused Cache* després d'assegurar que la realització de la comunicació d'escriptura de la línia a la memòria global s'hagi completat, si la línia s'havia modificat.

5.1.2 Cache Transaccional

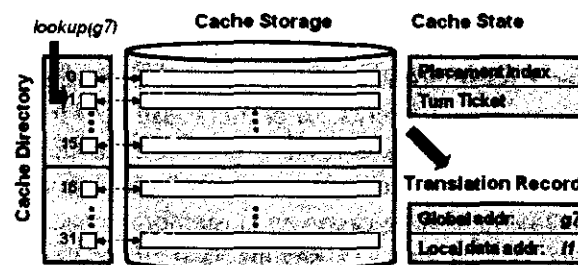
La Cache transaccional està destinada a les referències de memòria que no exposin qualsevol localitat espacial. Perquè els coeficients de MISS siguin alts, aquesta cache està dissenyada per oferir molt poca sobrecarrega de HIT i MISS permeten la superposició de càlculs i comunicació. El disseny introdueix estructures molt simples que permeten el suport per recerca (*lookup*), col·locació, comunicació, coherència i mecanismes de sincronització i traducció.

En aquesta configuració, l'emmagatzematge transaccional s'organitza en una petita cache de capacitat de 4KB, totalment associativa, i amb línies de cache de 32 128-bytes.

5.1.2.1 Estructures de la cache transaccional

La memòria transaccional està composta per les següents quatre estructures de dades:

- 1) *Cache Directory* per recuperar les línies.
- 2) *Cache Storage* per mantenir les dades de l'aplicació.
- 3) *Translation Record* per preservar els resultats de *lookup* per cada referència.
- 4) *Cache State* per estats addicionals de cache.



El *Cache Directory* s'organitza com un vector de 32 4-byte entrades. Cada entrada conté l'adreça base global associada amb l'entrada de línia de cache. L'índex de l'entrada en l'estructura de directoris també s'utilitza com a índex en *Cache Storage* per trobar les dades relacionades amb l'esmentada entrada. La *Cache Storage* es organitza com 32 línies de cache, on cada línia de 128-bytes es alinea en un límit de 128 bytes.

Per augmentar la concurrència, el *Cache Directory* i les estructures d'emmagatzematge són lògicament dividides en dues particions de la mateixa mida; la *Cache Turn Ticket* indica quina partició és activament usada. Dins de la partició activa, la *Cache Placement Index* apunta a la línia de cache que serà usada en el següent MISS.

A un nivell més alt, la partició activa es fa servir per posar línies de memòria requerides per la transacció actual, mentre que l'altra partició es usa com a buffer de línies de cache de transaccions prèvies mentre les seves dades modificades són tornades a la memòria global.

5.1.2.2 Model operatiu de la cache transaccional

Una transacció per a nosaltres és un conjunt de càlculs i comunicacions relacionades que passaran com una unitat (però mai es podrà fer *rollback*). Les operacions en una transacció passen en 4 etapes consecutives: (1) inicialització, (2) comunicació dins de la memòria local, (3) càlculs associats amb la transacció, i (4) la propagació de qualsevol estat modificat de nou a la memòria global.

Durant la inicialització, en el pas 1, la *Cache Turn Ticket* agafa l'altra partició. La *Cache Placement Index* s'estableix en la primera línia de cache de la nova partició activa. En la seva

configuració, el seu valor es 0 o 16, quan el *ticket* és, respectivament, el que apunta a la partició 0 o 1. A més, totes les entrades a la *Cache Directory* en la nova partició activa són esborrades.

Al pas 2, les dades corresponents a cada referència de memòria global es posen en la memòria local, utilitzant seqüències de cerca (*lookup*) i possiblement crides al controlador de MISS. El procés de cerca (*lookup*) d'una determinada referència *r*, registre de traducció *h*, i adreça global *g* primer procedeix amb un estàndard lookup a la cache d'alta localitat, ja que no volem repetir dades en les dues estructures de cache. Aquesta primera cerca es pot evitar si l'adreça *g* es garanteix que no es troba en la cache d'alta localitat. Quan passa un HIT, el camp de l'adreça base local en el registre de traducció *h* es simplement configurat perquè apunti a la subsecció apropiada de la línia en l'emmagatzematge de la cache d'alta localitat. Quan passa un MISS, però, es procedeix al control de l'adreça *g* contra l'entrada del directori de la cache transaccional. Aquest *lookup* és ràpid en les arquitectures amb unitats SIMD (Single Instruction, Multiple Data), tals com les SPEs.

El pas 3 procedeix amb els càlculs, utilitzant el mateix registre de traducció que s'ha vist en la secció 5.1.1.

En el pas 4, cada posició d'emmagatzematge modificada per un STORE en el pas 3 es directament propagada dins la memòria global. Aquest enfocament elimina la necessitat d'estructures de dades addicionals (com ara la de Dirty Bits). Aquestes comunicacions asíncrones es produeixen independentment que es produeixi un HIT o un MISS en el pas 2. A més, només la modificació de bytes de dades (no d'entrades de línia, a menys que la línia s'hagi modificat) son transferides dins de la memòria global durant el pas 4.

Per tal de garantir la coherència en i entre les transaccions, cada dada transferida es marcada amb l'índex de la línia de cache que la utilitza (des de 0 a 31), i es situa just després de l'operació de transferència de dades. Totes les transferències de dades etiquetades amb la mateixa etiqueta es veuen obligades pel maquinari a estrictament ser portades en l'ordre en que van ser programades. El codi de sincronització es produeix en precisament dos llocs. La primera sincronització es col·loca entre el pas 2 i 3, per garantir que arribin les dades abans de ser usades. Quan la partició 0 es activa, esperem les operacions de transferència de dades amb les etiquetes [0 ... 15], i esperem per les etiquetes [16 ... 31], en cas contrari. Per la transferència de dades iniciada en el pas 4, el codi de sincronització es col·loca al principi de la següent transacció amb el mateix valor per la *Cache Turn Ticket*, sincronitzant amb les operacions de transferència de dades etiquetades amb els números [0 ...15], o [16 ... 31].

5.1.3 Consistència de Memòria

Per cada línia de cache en la Cache d'Alta Localitat, el bloc coherent de memòria conté informació sobre quines dades han estat modificades sent mantingudes usant un estructura de dades *Dirty Bits*. Sempre que un línia de cache ha de ser desallotjada, el procés d'escriptura a memòria global està compost per tres passos. La línia de cache en la memòria principal és llegida, llavors una operació de fusió és aplicada entre les dues versions, entre la resident a la *Cache Storage* i la recentment transferida, i finalment, la sortida de la fusió es enviat a memòria principal. Totes les transferències de dades són síncrones i atòmiques.

5.2 TRANSFORMACIONS DE CODI PER AQUESTA SOFTWARE CACHE

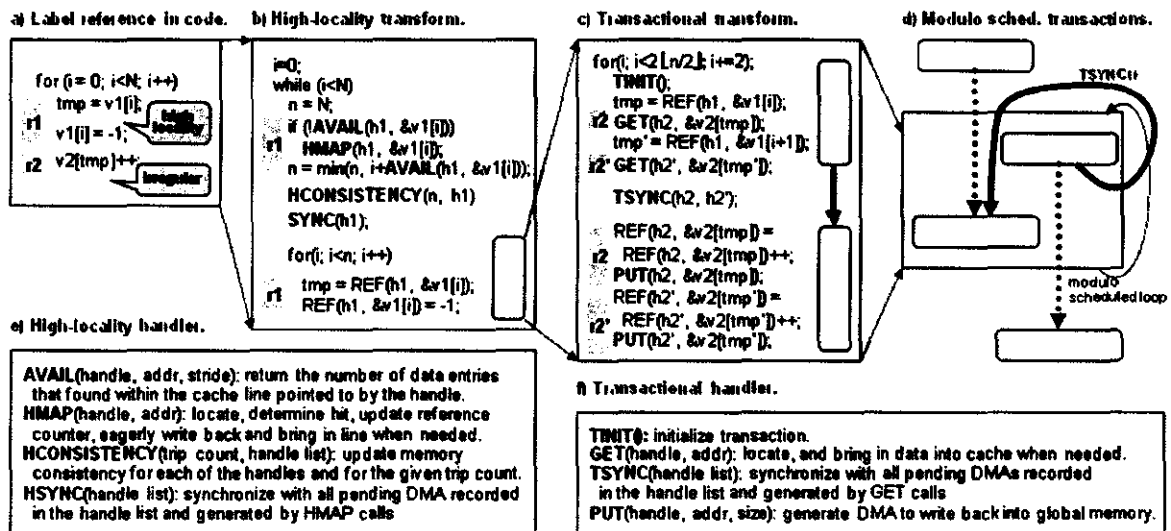
Es descriuen en aquesta secció el tipus de tècniques de transformació de codi que per ara es permeten utilitzar en la software cache híbrida utilitzada. S'utilitza aquí la distinció entre els esquemes de referències de memòria d'alta localitat i accessos irregulars. Accessos amb l'esquema d'alta localitat s'assignen a la Cache d'Alta Localitat i tots els accessos irregulars s'assignen a la Cache transaccional.

Els objectius de transformació de codi és en l'execució de bucles.

Les transformacions de codi es realitzen en tres fases ordenades:

- 1) Es classifiquen les referències de memòria entre d'alta localitat o accessos irregulars;
- 2) Transformar el codi per optimitzar les referències de memòria d'alta localitat, i
- 3) Transformar el codi per optimitzar referències de memòria irregulars.

Veiem a continuació aquest procés mitjançant un exemple.



5.2.1 Classificació d'accessos a memòria

A la Fase 1, els accessos a memòria es classifiquen com accessos d'alta localitat o accessos irregulars. En el nostre exemple tenim accessos a $v[i]$ amb $i = 0..N$ i $v2[tmp]$ amb $tmp = v[i]$ que estan etiquetats, respectivament, com accessos d'alta localitat i accessos a memòria irregulars.

5.2.2 Transformacions dels accessos d'alta localitat

En la fase 2, es transforma el bucle original `for` en dos bucles niats que bàsicament realitzen un fragmentació de l'espai d'iteracions. Com veiem a l'apartat **b)** de la figura, el bucle exterior `while` itera sempre i quan no s'hagin visitat totes les N iteracions originals. El bucle interior itera a través d'un subconjunt dinàmic de iteracions, n , on n és calcula com el major nombre

d'iteracions perquè cap de les referències d'alta localitat experimenti un MISS. Com es mostra en el bucle interior, no hi ha sobrecarrega de cache (les despeses de REF detallades en l'apartat 1c de la figura són sense cost).

Detallem ara els treballs introduïts per cada referència d'alta localitat en el cos del bucle extern. Pensant en l'apartat b) de la figura veiem els treballs relacionats amb la referència $r1$ amb el registre de traducció $h1$ i l'adreça de memòria global $g1=v1[i]$. En primer lloc, calcula la disponibilitat amb la macro AVAIL del nombre de iteracions pels quals l'adreça $g1$ serà present a l'actual línia de cache assenyalat pel registre de traducció $h1$. Si el número es 0, tenim un MISS (error), i invoquem el controlador HMAP dels MISS.

Tingueu en compte que un MISS no implica comunicació, la línia de cache pot estar present a la cache degut a altres referències. Una vegada que la nova línia és instal·lada en el registre de traducció $h1$, recalculam AVAIL i actualitzem l'actual fragmentació dinàmica factor n per tenir en compte el nombre d'iteracions pels que la referència $r1$ no experimentarà un MISS.

Després de processar totes les referències d'alta localitat, la dinàmica de fragmentació factor n és definitiva, i es pot utilitzar per informar al bloc coherent de memòria de totes les localitzacions de memòria que seran afectades per referències d'alta localitat a l'interior del bucle `for`. S'ha de tenir en compte que tot aquest treball es realitza en paral·lel amb les possibles sol·licituds asíncrones a la DMA pel controlador HMAP del MISS.

L'última operació consisteix en realitzar la sincronització amb tots els DMAs pendent, utilitzant les etiquetes de comunicació trobades en els descriptors de línies de cache associats amb cada referència d'alta localitat.

Una tasca addicional és per calcular la configuració òptima per a la cache d'alta localitat. Si més no, necessitem una línia de cache per diferents referències de memòria d'alta localitat, però poc més permet millorar la latència oculta pel procés d'escriptura a memòria principal. Nota, però, que sempre pot reduir-se un o més accessos a memòria d'alta localitat per ser tractades com a referències irregulars. A la pràctica, seleccionem la mida de la línia més gran que satisfaci cada referència d'accés a memòria d'alta localitat present en el bucle.

5.2.3 Transformacions per accessos irregulars

En la fase 3, transformem el bucle interior per tal d'optimitzar la cache sobrecarregada pels accessos a memòria irregular. La primera tasca es determinar les transaccions. En aquesta software cache, l'abast d'una transacció és un bloc, o un subconjunt. Les grans transaccions són beneficioses ja que potencialment augmenten el nombre de MISSES concurrents, augmentant així la superposició de la comunicació. En general, una transacció pot contenir molts diferents accessos irregulars ja que hi ha entrades en una única partició de la cache transaccional, 16 en aquesta configuració. Degut al seu enfocament en els bucles, les operacions més grans s'aconsegueixen principalment a través de desenrotllar bucles. En el exemple, es desenrotlla el bucle `for` interior per un factor de 2 a fi d'incloure dues referències $v2[tmp]$ i $v2[tmp']$ dins d'una única transacció.

El codi generat per una transacció estretament segueix de prop les quatre etapes descrites al punt 5.1.2.2. Com es veu en l'apartat c) de la figura, en primer lloc, s'inicialitza l'operació (etapa 1) i a continuació, es procedeix a l'adquisició de forma asíncrona de dades de cada referència irregular $r2$ i $r2'$ usant la macro GET (etapa 2). Una vegada que totes les referències

irregular s'han tractat, emetem l'operació TSYNC per sincronitzar a tots els pendants DMAs expedits per l'operació GET. Accedim a les dades usant la macro REF (etapa 3) i escrivim el retorn a memòria principal usant la macro PUT (etapa 4).

Conceptualment, el treball dins d'una transacció pot ser visualitzada com dues tasques, els passos 1 i 2, seguit dels passos 3 i 4, com es mostra en el costat dret del codi de la figura apartat c). La sincronització entre les dues tasques pot ser visualitzada com una dependència entre les dues. Podem veure que el codi de l'aparat c) està lluny de ser òptim. Encara que els MISSES estiguin en paral·lel dins la Tasca 1, la dependència entre les dues tasques dependents encara serialitzades es la major part del treball.

Per augmentar encara més la concurrència, s'aplicarà una tècnica coneguda com a *modulo scheduling* o *double buffering*. Al apartat d) de la figura il·lustra l'essència d'aquesta tècnica. En lloc de processar d'una operació darrera l'altra, es mantenen dues operacions consecutives al moment. En un determinat estat d'equilibri de la iteració (per exemple en el quadre ombrejat en l'apartat d) de la figura), s'iniciarà un nova transacció (Passos 1 i 2) i després completarà l'operació de la iteració anterior del bucle (Passos 3 i 4). Com a conseqüència, la sincronització no es produeix entre les dues tasques en les mateixes iteracions, com en l'apartat c) de la figura, però es produeix entre les dues tasques en dues iteracions consecutives.

6 COMPILADOR NANOS MERCURIUM C/C++

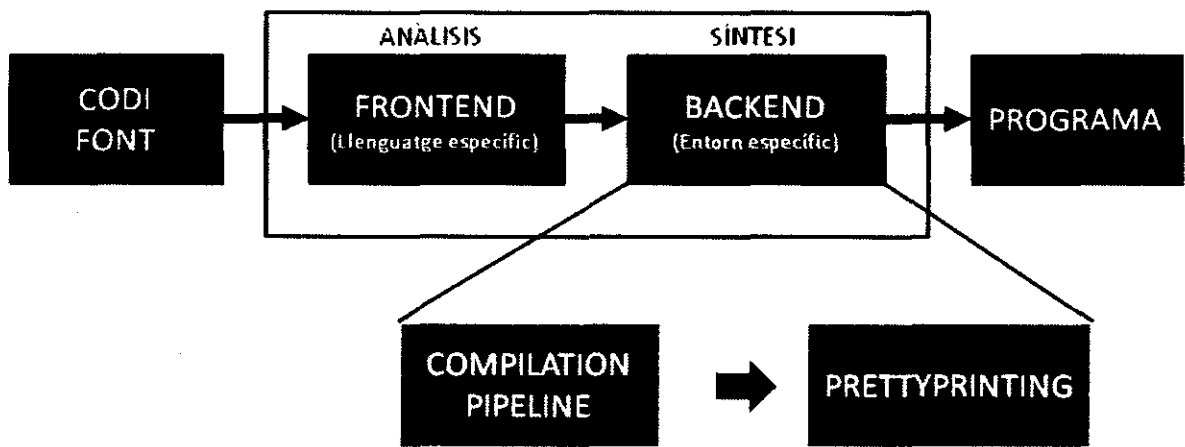
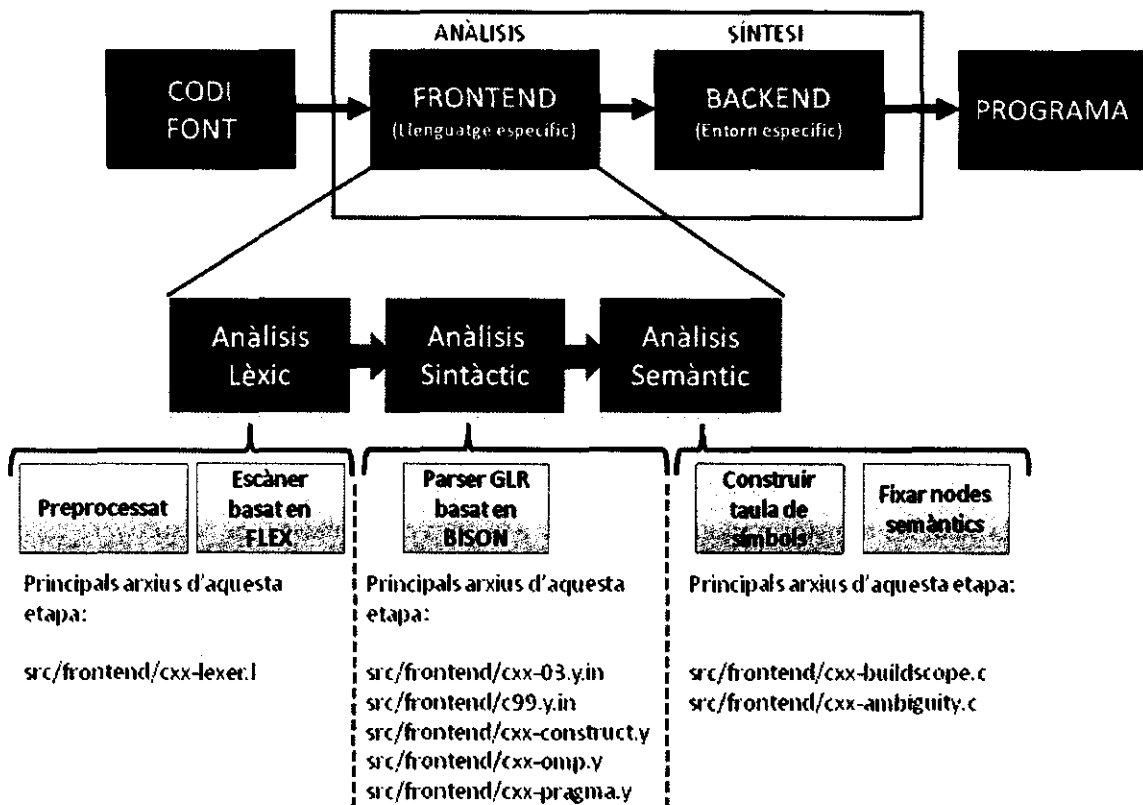
En aquest capítol veurem en detall el compilador Nanos Mercurium C/C++, comparant les parts típiques d'un compilador amb aquest compilador i explicant en cada fase del compilador tradicional quines parts i fitxers intervenen del compilador Nanos Mercurium C/C++.

6.1 INTRODUCCIÓ AL COMPILADOR NANOS MERCURIUM C/C++

Una vegada vista les transformacions de codi per a la correcta utilització de la software cache híbrida i el processador CELL BE, s'utilitzarà el compilador Nanos Mercurium C/C++ per realitzar aquestes transformacions a un codi d'entrada. Aquestes transformacions les realitzarem en l'etapa de backend del compilador Nanos Mercurium C/C++.

El Nanos Mercurium C/C++ és un compilador font a font inicialment desenvolupat per omplir el buit de suport C++ en OpenMP. Aquest compilador proporciona un entorn de programació que facilita el desenvolupament de fases per al compilador.

Per entendre bé com funciona el compilador Nanos Mercurium C/C++ [31][32] introduïrem les parts d'aquest compilador amb el que s'espera trobar quan parlem de compiladors tradicionals. A continuació veiem en la següents figures, les típiques parts d'un compilador i les eines i els fitxers que utilitza el compilador escollit en cadascuna de les fases.



El Codi Font que entra en el compilador Nanos Mercurium C/C++ és el llenguatge d'alt nivell C/C++. La sortida que genera el compilador Nanos Mercurium C/C++ és també llenguatge d'alt nivell C/C++, a diferència d'altres compiladors que tenen com a sortida codi màquina, codi binari o codi assembler.

Cada fitxer font d'entrada, abans de qualsevol transformació, és en primer lloc reconegut (etapa d'Anàlisi Lèxic). En C/C++ això implica executar el preprocessador (`gcc -E` o `cpp` en el cas del compilador Nanos) i, a continuació, reconèixer la seva sortida. El reconeixement es realitza en dues fases. En el primer pas es realitza un reconeixement ambigu, utilitzant un

*parser*¹⁷ GLR¹⁸ construïts amb una versió modificada de bison (2.3-rufi), on són considerades totes les possibles interpretacions de construccions sintàcticament ambigües. Aquest procés, com és comú en els compiladors, produeix un AST¹⁹ que pot contenir ambigüitats degut al lliure context del reconeixedor. El segon pas del reconeixedor implica l'eliminació d'aquestes ambigüitats. El compilador recull informació semàntica que serà utilitzada quan tracti amb ells. Una vegada que l'arbre és podat i lliure d'ambigüitats aquest alimenta, juntament amb la informació semàntica recollida fins el moment, al *pipeline* de fases del compilador. El *pipeline* de fases del compilador està formada per diverses llibreries de càrrega dinàmica, cada una implementant una fase. Aquestes fases seran escrites en C++, per part de l'usuari que implementa les transformacions. Cada fase té la representació intermèdia (incloent el AST i la informació semàntica) resultants de la fase anterior. Una vegada que les fases són executades, la transformació del AST resultant són *prettyprinted* (impreses) al fitxer de sortida.

El flux més detallat del compilador Nanos Mercurium C/C++ es descriu de la següent manera:

FRONTEND

- 1) En primer lloc, el codi font és preprocessat.
- 2) El Font preprocessat es reconegut.
- 3) L'arbre ambigu és llavors analitzats semànticament.

BACKEND

- 4) S'inicien les fases del *pipeline* de compilació.
- 5) L'arbre modificat de sintaxis abstracte és *prettyprinted* (imprès) en un arxiu.

- 1) Anàlisi lèxic: En aquesta fase l'arxiu font d'entrada es preprocessat usant la configuració del preprocessador. Normalment s'utilitzen `gcc -E` o `cpp`.

Aquest pas és obligatori ja que els arxius d'entrada no preprocessats poden ser incomplets o fins i tot arxius invàlids. Es converteixen els caràcters en *tokens* (unitats bàsiques del llenguatge, com ara identificadors o símbols) per facilitar el treball del reconeixedor, i s'elimina aspectes irrellevants (espais, comentaris, etc.). Generalment es defineix mitjançant expressions regulars.

El compilador també reconeix les línies de `#pragma`.

- 2) Anàlisi sintàctic: Quan el codi ha estat preprocessat aquest es reconegut. Construeix un arbre que representa l'estructura sintàctica del programa, és a dir, que els *tokens* apareixen en l'ordre correcte. Aquest fase de construcció del AST s'utilitzarà en tots els processos posteriors. Aquest pas fa un reconeixement amb una gramàtica lliure de context, per la qual cosa l'arbre generat en aquest cas és ambigu. Ser ambigu té diversos inconvenients, principalment que l'arbre és completament inútil. També té alguns avantatges, permet adonar-se de quin grau tenen algunes ambigüitats en C++. Aquestes ambigüitats són marcades.

¹⁷ Reconeixedor

¹⁸ GLR - Generalized Left-to-right Rightmost derivation parser. És una derivació del *parser* LR (Left-to-right), per manejar nondeterministic i gramàtiques ambigües.

¹⁹ Abstract Syntax Tree - Arbre de sintaxis abstracte

S'utilitza un *parser* GLR. Actualment, aquest reconeixedor requereix una versió especial del bison-2.3, amb suport per la directiva `default-marge`. Aquesta directiva es justament una opció de productivitat per estalviar escriure en la gramàtica però aquesta no afegeix cap altre funcionalitat que no existeix en el bison-2.3. Aquest reconeixedor es construeix després d'alguna modificació de la gramàtica estàndard i aquesta accepta aplicacions sintàcticament correctes de C99 (ISO/IEC 9899:1999) i C++ (ISO/IEC 14882:2003).

El reconeixedor intenta suportar diverses extensions sintàctiques de GNU (mcxx intenta durament acceptar qualsevol cosa acceptada per GCC 4.1

Principals fitxers de compilador involucrats en aquests passos són:

`src/frontend/cxx-lexer.l`

`src/frontend/c99.y.in`

`src/frontend/cxx03.y.in`

`src/frontend/cxx-construct.y`

`src/frontend/cxx-omp.y`

`src/frontend/cxx-pragma.y`

- 3) Anàlisi semàntic: Amb la finalitat de fixar el AST generat en la fase anterior, és necessari recollir informació dels símbols que ens permetrà fer realitat el significat exacte del codi.

Aquest anàlisi és l'encarregat d'afegir informació semàntica a l'arbre, i crear la taula de símbols (amb les variables, noms de funcions, etc.). Fa diverses comprovacions (com ara la de tipus entre variables, les crides que es fan a operacions, etc.) per a determinar que el programa és consistent.

Fa aquest procés en dues fases, primer la construcció de l'arbre en la fase anterior i, a continuació, la poda que permet un manteniment més senzill de la gramàtica (sempre la part més dura de qualsevol reconeixedor) i simplificar la part semàntica (és té que tractar amb els elements sintàctics sempre de la mateixa manera). Fer les dues coses al mateix temps és possible, però és molt més complex.

Aquest procés construeix el *scope* (taula de símbols) informació necessària per a la desambiguació.

La taula de símbols és una estructura de dades que conté un registre per a cada identificador utilitzat en el codi font, amb camps que contenen informació rellevant per a cada símbol (atributs).

- Quan l'Anàlisi Lèxic detecta un símbol de tipus identificador, l'ingressa a la taula de símbols.
- S'ingressa informació per als atributs dels símbols, i es fa servir aquesta informació de diverses maneres.
- Pot ser necessari incorporar noves entrades a la taula de símbols més endavant.

Principals fitxers involucrats en el compilador en aquest pas són:

`src/frontend/cxx-buildscope.c`

`src/frontend/cxx-ambiguity.c`

- 4) *Pipeline* de fases del compilador: A partir de l'arbre construït a l'anàlisi sintàctic i omplert d'informació per l'anàlisi semàntic, es genera un codi intermedi, que podrà ser optimitzat o transformat.

Una vegada que l'arbre és podat i lliure d'ambigüitats aquest alimenta, juntament amb la informació semàntica recollida fins el moment, a aquesta fase, i pot ser iniciada.

El compilador executa cada fase seqüencialment. Normalment aquesta fase modifica el AST amb la finalitat de fer transformacions en el codi actual. Aquesta fase, com es pot entendre, no són personalitzables a les necessitats de tots. Estan escrites en C++ i el compilador proporciona una mena de SDK per fer-les. Aquestes fases es poden utilitzar de manera seqüencial, les dades poden calcular-se i passar-se a les fases següents, d'això el nom de *pipeline*. Això permet tenir fases separades treballant en una informació intermèdia addicional, al costat del AST i de la informació del símbols.

- 5) Finalment es genera la sortida: L'últim pas a fer pel compilador es l'anomena't *prettyprinting*. El AST resultant després del *pipeline* de fases del compilador es imprès dins d'un arxiu de sortida, o per la sortida estàndard.

6.2 FITXERS DE L'ENTORN DEL COMPILADOR

El codi del compilador està agrupat en directoris per a facilitar l'organització del codi i la compilació, com hem vist en l'aparat anterior.

Hi han varis tipus de fitxers:

Definició de símbols (extensió *.l): Especificació dels tokens en format LEX.

Definició de gramàtica (extensió *.y): Especificació d'un conjunt de regles de gramàtica en format YACC (bison).

Capçalera de C o C ++(extensió *.h o *.hpp): Fitxer de capçalera en llenguatge C++ o C. Aquests fitxers contenen macros, capçaleres de funcions o declaracions dels tipus utilitzats en l'entorn.

Codi C o C++ (extensió *.c o *.cpp): Fitxers de codi C++ o C. Aquests fitxers implementen rutines declarades en algun fitxer de capçalera i altres rutines internes, que no són visibles fora d'aquest fitxer (es declaren amb la paraula clau *static*).

7 DISSENY DE LA NOVA FASE

En aquest capítol s'explicarà com es desenvolupen noves fases en el compilador Nanos Mercurium C/C++, a continuació veurem el nostre disseny de noves directives pragmas que el programador afegirà en el codi per poder aplicar les transformacions corresponents, i finalment veurem el disseny de l'algorisme tenint en compte les transformacions necessàries per a la software cache híbrida i el processador CELL BE.

7.1 DESENVOLUPANT NOVES FASES DEL COMPILADOR

El compilador ha estat dissenyat de manera que les noves fases de compilació es poden desenvolupar en diversos nivells d'abstracció. El nivell més baix implica tractar amb els nodes del AST, utilitzant el nom d'atributs, per la qual cosa la capa exacta de l'arbre no ha de ser coneguda.

Per sobre d'aquest nivell, les llibreries dels compiladors proporcionen més classes abstractes per fer front a construccions comunes del llenguatge C i C++, igual que pels bucles, les definicions de funcions, identificadors, etc. Construccions de nivell superior compten amb codis especialitzats amb **#pragma** genèrics i construccions d'OpenMP. El treball difícil de reconeixement es tasca del compilador, tots els altres processos poden ser construïts sobre d'això.

Molts dels processos realitzats per una fase escrita per l'usuari consistiran en transformacions de codi. Tractar directament amb el AST és complicat, el programador només té que generar una cadena vàlida de representació de codi, reconeixent aquest en el context adequat i a continuació substituir el AST. Això estalvia al programador conèixer els detalls exactes de l'estructura.

Exemple:

Com un simple exemple de com dur a terme una transformació considerem el següent codi.

```
int k ;
#pragma mypragma unroll times ( 4 )
for ( k = 0 ; k < 100 ; k++)
{
    a [k] = a [k] + 1 ;
}
```

Volem que el compilador desenrotlli aquest codi quatre vegades. Per mitjà d'un arxiu de configuració, la compilador és conscient del prefix `mypragma` (en cas contrari no en fa cas).

```
class MyPragmaPhase : public PragmaCustomCompilerPhase
{
    public:
        MyPragmaPhase ( ) : PragmaCustomCompilerPhase ( "mypragma" ) {
            on_directive_post["unroll"].connect(functor(&MyPragmaPhase::unroll_
                postorder, *this ) );
        }
        ...
};
```

La classe *PragmaCustomCompilerPhase* proporciona una interfície fàcil per construccions de Pragma. El programador pot obligar a alguna construcció o directiva d'algun codi executar-se ja sigui en preorder (abans de qualsevol transformació, incloses les que puguin afectar els nodes interiors) com en postorder (després que tots els nodes interior ja s'han transformat). *unroll_postorder* es durà a terme en postorder per construccions del *unroll*.

```
void unroll_postorder ( PragmaCustomConstruct pragma_construct )
{
    /* Codi d'inicialització de variables o més */
    // Codi Font resultant de la transformació
    replaced_for
        << " { "
        << unrolled_for // el unrolled_for és omplert més avall
        << tail_for // el tail_for és omplert més avall
        << " } ";
    // Escriu al unrolled_for
    unrolled_for
        << " for ( " << for_statement.get_iterating_init ( )
        << " ( " << induction_var << " + " << unroll_times << " )<= ( " << upper_bound << " ); "
        << induction_var << "+= ( ( " << loop_step << " )_ " << unroll_times << " ) { "
        << unrolled_for_body // el unrolled_for_body es omplert més avall
        << " } ";
    // Almenys una iteració serà en el unrolled_for
    unrolled_for_body << loop_body ;
    // Creem el unrolled_loop_body
    Symbol induction_var_symbol = induction_var.get_symbol ( ) ;
    for ( int i = 1 ; i < unroll_times ; i ++ )
    {
        // Creem un mapa de situació
        ReplacedExpression replacements ;
        // Reemplacem cada variable d'inducció ind_var per ind_var+1
        replacements.add_replacement ( induction_var_symbol , induction_var << " + " << i ) ;
        // Ho reemplacem al arbre
        Statement replaced_statement = replacements.replace ( loop_body ) ;
        // I creem el font
        unrolled_for_body << replaced_statement ;
    }
}
```

```

// Creem el tail_for
tail_for
  << " for ( ; " << for_statement . get_iterating_condition ( ) << " ; "
  << for_statement.get_iterating_expression ( ) << " ) "
  << loop_body ;
// Ara recorrem el nou font creat
AST_t replaced_tree = replaced_for.parse_statement (
pragma_construct.get_scope_link ( ) );
pragma_construct .get_ast( ).replace ( replaced_tree );
}

```

Una vegada que s'aplica la transformació, el codi resultant és el següent.

```

int k;
{
  for ( k=0; (k+4) <= ( (100) - 1 ); k+= ( (1) * 4 ) ) {
    a [ k ] = a [ k ] + 1;
    a [ ( k + 1 ) ] = a [ ( k + 1 ) ] + 1;
    a [ ( k + 2 ) ] = a [ ( k + 2 ) ] + 1;
    a [ ( k + 3 ) ] = a [ ( k + 3 ) ] + 1;
  }
  for ( ; k < 100; k++ )
    a [ k ] = a [ k ] + 1;
}

```

7.2 DESENVOLUPANT LA NOSTRA FASE DEL COMPILADOR

El que es pretén en aquest projecte es dissenyar i implementar un fase de generació de codi per al processador multi nucli CELL BE, dintre de la infraestructura de compilació Nanos Mercurium C/C++ desenvolupada pel grup de recerca nanos.

Aquesta fase consisteix en modificar el compilador de tal manera que generem el codi necessari quan un programador ens entri certes directives que veurem a continuació, facilitant així la utilització màxima dels recursos del CELL BE sense que el programador en qüestió tingui els coneixements interns del compilador, de l'estructura del CELL BE o de la software cache híbrida.

Això implica marcar uns nous pragmas. El disseny d'aquest pragma es el següent:

```

# pragma hlc hlcached(@r) tcached (@i)
@r – llista de variables amb accessos a la cache d'alta localitat
@i – llista de variables amb accessos irregulars a la cache transaccional.

```

L'usuari ens haurà d'indicar quines adreces són de cada tipus, perquè puguem fer la generació de codi corresponent al tipus d'accés.

Exemple:

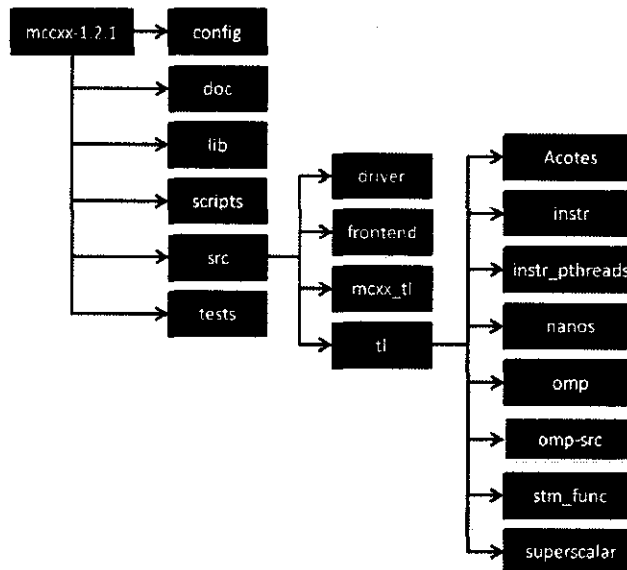
```

#pragma hlc hlcached(v1) tcached(v2)
for(i=0 ; i < N ; i++){
    tmp = v[i];
    v1[i] = 1;
    v2[tmp]++;
}

```

7.3 UBICACIÓ NOVA FASE DINS DE L'ESTRUCTURA DEL COMPILADOR

L'estructura de carpetes del compilador Nanos Mercurium C/C++ es distribueix de la següent forma:



La carpeta **omp-src** es on desenvoluparem el nostre projecte, és una replica de la **omp** per mantenir l'estructura, ja que algunes transformacions ens eren útils, com per exemple les variables **shared**, seran les nostres **hlcached** i les variables **private** seran les **tcached**..

Aquesta carpeta **omp-src** conté moltes classes, però per la nostre generació hem modificat només algunes d'elles. A la classe *tl-loop.cpp* hi ha la part més important de la nostre implementació en la funció **get_loop_distribution_code** cridada des de la classe *tl-for.cpp* en la seva funció **for_postorder**.

7.4 DISSENY GENERACIÓ DE CODI

7.4.1 Transformació de codi

Per veure com afrontarem les transformacions abans hem de veure un exemple senzill.

D'un programa original com el següent:

```
#pragma hlc hlcached(a, b)
for(i=0 ; i < VectorSize ; i++ ){
    a[i] = b[i];
}
```

El compilador ha de generar les transformacions següents:

```
void tuned_STREAM_Copy_spe(int *pVectorSize, double *c, double*a)
{
    int j, VectorSize;

    // Inicialitzacions varies
    _OPEN_MEMORY_TRACER_LIB (0);

    _DECLARE_MEMORY_TRACER_VARS ();
    _DECLARE_MEMORY HOLDER_VARS ();
    _DECLARE_MEMORY_TRANSLATION_VARS ();

    _INIT_MCELL(0,_trans_01,_mask_01,_baddr_01);
    _INIT_MCELL(1,_trans_02,_mask_02,_baddr_02);
    _INIT_MCELL(2,_trans_03,_mask_03,_baddr_03);

    // Inicialitzar el lowe_bound, upper_bound i el step per trossejar el bucle
    _lb_01 = 0;
    _ub_01 = VectorSize;
    _step_01 = 1;
    _NORMALIZE_LOOP(_ub_01,_lb_01,_step_01,&_norm_ub_01,&_norm_lb_01);

    _start_01 = 0;
    _end_01 = 0;
    _NEXT_ITERS(&_start_01,&_end_01,_norm_ub_01,_norm_lb_01,_STATIC_0,_work_01);
    while (_work_01) {
        _sub_start_01 = _start_01;
        _work_inside_chunk_01 = TRUE;
        while (_work_inside_chunk_01) {
            _ind_01 = _sub_start_01 * _step_01;

            // Mirem si tenim la variable a
            _INIT_LOOKUP(_lookup_01);
            _LOOKUP(0, _trans_01, _mask_01, _baddr_01, &a[_ind_01],
            double,_lookup_01);
```

```

// Si no la tenim
if(__builtin_expect(_lookup_01, 0)){
    // La portem de memòria principal
    _MMAP_NEW(0, _trans_01, _mask_01, _baddr_01, &a[_ind_01],
double, _READ, _GLOBAL_ | _STRIDED, EXECUTOR | RECORD);
}

// Mirem si tenim la variable b
_INIT_LOOKUP(_lookup_01);
_LOOKUP(1, _trans_02, _mask_02, _baddr_02, &c[_ind_01],
double, _lookup_01);
// Si no la tenim
if(__builtin_expect(_lookup_01, 0)){
    // La portem de memòria principal
    _MMAP_NEW(1, _trans_02, _mask_02, _baddr_02, &c[_ind_01],
double, _WRITE, _GLOBAL_ | _STRIDED, EXECUTOR | RECORD);
}
_next_iters_01 = LS_PAGE_SIZE;
_NEXT_MISS(0, _trans_01, _mask_01, _baddr_01, &a[_ind_01], double,
sizeof(double), _next_iters_01);
_NEXT_MISS(1, _trans_02, _mask_02, _baddr_02, &c[_ind_01], double,
sizeof(double), _next_iters_01);
_sub_end_01 = _sub_start_01 + _next_iters_01;
if (_sub_end_01 > _end_01) _sub_end_01 = _end_01;
_work_inside_chunk_01 = (_sub_end_01 < _end_01);
// Modifiquem els dirty bits de la variable c que es on escriurem, ja que
// més tard a de ser tornada a memòria principal
_UPDATE_DIRTY_BITS(1, _trans_02, _mask_02, _baddr_02, &c[_ind_01],
double, sizeof(double), sizeof(double), (_sub_end_01 - _sub_start_01));

_MEM_BARRIER();
for (_ind_01 = _sub_start_01; _ind_01 < _sub_end_01; _ind_01 = _ind_01 + 1)
{
    j = _ind_01 * _step_01;
    // Fem LOAD de la variable a i ho guardem a __double_tmp01
    _LD(0, _trans_01, _mask_01, _baddr_01, &a[j], double,
__double_tmp01);
    // Guardem a la variable c el valor de a __double_tmp01
    _ST(1, _trans_02, _mask_02, _baddr_02, &c[j], double,
__double_tmp01);
}
_sub_start_01 = _sub_end_01;

}
_start_01 = _end_01;
// Calculem les següents iteracions
_NEXT_ITERS(&_start_01, &_end_01, _norm_ub_01, _norm_lb_01, _STATIC_0, _work_01);
}
_DECREMENT_REFERENCE(0); _DECREMENT_REFERENCE(1); _DECREMENT_REFERENCE(2);
_CLOSE_MEMORY_TRACER_LIB ();
}

```

Totes les funcions com `_NEXT_ITERS()`, `_MEM_BARRIER()`, etc són macros ja desenvolupades pel grup de recerca, per tant no ens centrarem en la seva implementació, sinó en que fan i quan hem de cridar-les.

Analitzant el següent codi, veiem amb més detall que hem de fer per cada variable del cos del bucle. La resta són inicialitzacions diverses pel CELL, i tot el codi que es genera per desenrotllar el bucle en x processos pera cada SPEs del processador.

Per cada variable tenim:

- 1) Hem de tenir la reserva de memòria `_INIT_MCELL` per tantes variables com tinguem.
- 2) Per cada variable:

```
_INIT_LOOKUP(...);  
//Mirem si la variable està a memòria  
_LOOKUP(...);  
// Si tenim un MISS (No tenim la variable)  
IF(...)  
{  
    // Portem la variable de Memòria principal  
    _MMAP_NEW(...)  
}  
_NEXT_MISS(...)
```

- 3) Per les variables que facin STORE, es a dir que s'emmagatzemin dades dins d'elles es farà:

```
// Marcar que s'ha modificat perquè al final tornem la línia a memòria principal  
_UPDATE_DIRTY_BITS(...)
```

- 4) Pel que fa al cos del bucle es farà LOAD o STORE (macros `_LD` i `_ST`), en funció de si la variable es de lectura, escriptura o lectura/escriptura a la vegada. Si la variable a més és d'accés irregular per cada LOAD farem GET abans i per cada STORE farem GET abans i PUT després.

Exemple:

```
a = b + 1;
```

El codi a generar seria:

```
_GET(0, &b, _baddr_01, _trans_01, _mask_01);
```

```

_LD(0, &b, _baddr_01, _trans_01, _mask_01, __double_tmp01);

__double_tmp02 = __double_tmp01 + 1;

_GET(1, &a, _baddr_02, _trans_02, _mask_02);

_ST(1, &a, _baddr_02, _trans_02, _mask_02, __double_tmp02);

_PUT(1, &a, _baddr_02, _trans_02, _mask_02);

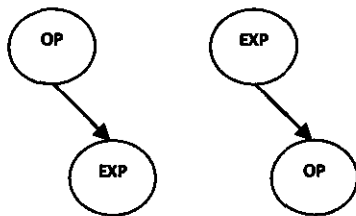
```

7.4.2 Transformacions del cos del bucle

En el cos del bucle, pot haver-hi qualsevol tipus d'expressions. En el nostre projecte hem acotat aquestes expressions al rang necessari per les primeres proves, per tant hi ha restriccions a les expressions entrades en el cos del bucle.

Les expressions reconegudes i tractades són:

Expressions unàries



On OPU és operador unari ++ o –

On EXP pot ser una variable *a*, o *a*[*j*], o *a*[*b*[*j*]], i les seves derivacions.

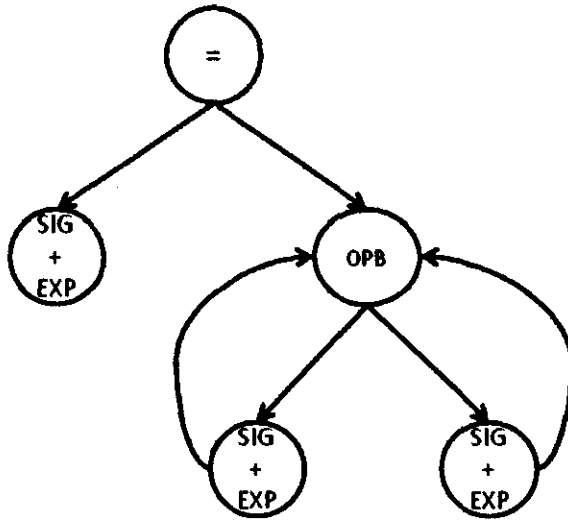
Exemples:

```

var++;
var--;
*var;
var[i]++;
var[i]--;
var1[var2[i]]++; //... i els seus símls

```

Expressions binàries



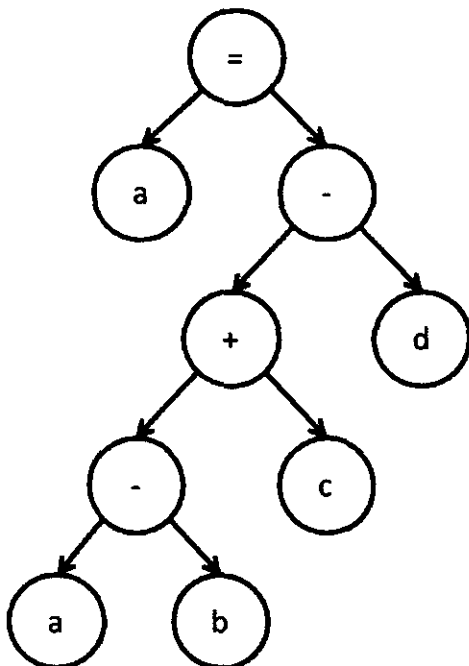
On **OPB** és un operador binar que pot ser SUMA (+) o RESTA (-).

SIG pot ser POSITIU(+), NEGATIU (-) o PUNTER (*).

A la vegada **SIG+EXP** com es veu en la figura pot tornar a descompondre's en dos operands + **OPB**, en la part dreta de l'assignació.

Veiem un exemple:

$a = a - b + c - d;$



Exemples:

```

var1 = var1 + var2;
var1 = - var1 - var2;
var1 = 10 + var2;
//... i els seus similis
var1[i] = var2[i];
var1[i] = var2[var1[i]];
var1[i] = var2[var1[var3[[i]]]];
var1[var2[i]] = 10;
//... i els seus similis
var1[j][j]=10;
//... i els seus similis

```

Farem un estudi de les diferents situacions que ens trobarem com a expressions dins del bucle i analitzarem les diferents transformacions depenen del tipus d'expressió i del tipus d'accés de les variables.

Casuístics d'exemple

Usant les macros `_ST` (store), `_LD` (load), `_GET` i `_PUT`, si les variables `a`, `b` i `c` fossin regulars.

| | | |
|---|-------------------------|--|
| 1 | <code>a[j] = 10;</code> | <code>tmp01=10;</code> <code>_ST (&a[j], tmp01);</code> |
|---|-------------------------|--|

| | | |
|---|---------------------------|--|
| 2 | <code>a[j] = b[j];</code> | <code>_LD (&b[j], tmp01);</code> <code>_ST (&a[j], tmp01);</code> |
|---|---------------------------|--|

| | | |
|---|------------------------------|--|
| 3 | <code>a[j] = b[a[j]];</code> | <code>_LD (&a[j], tmp01);</code> <code>_LD (&b[tmp01], tmp02);</code> <code>_ST (&a[j], tmp02);</code> |
|---|------------------------------|--|

| | | |
|---|---------------------------------|--|
| 4 | <code>a[j] = b[a[c[j]]];</code> | <code>_LD (&c[j], tmp01);</code> <code>_LD (&a[tmp01], tmp02);</code> <code>_LD (&b[tmp02], tmp03);</code> <code>_ST (&a[j], tmp03);</code> |
|---|---------------------------------|--|

| | | |
|---|---------|---|
| 5 | a[j]++; | <pre> _LD (&a[j], tmp01); tmp02 = tmp01 + 1; _ST (&a[j], tmp02); </pre> |
|---|---------|---|

| | | |
|---|------------|---|
| 6 | a[b[j]]++; | <pre> _LD (&b[j], tmp01); _LD (&a[tmp01], tmp02); _LD (&b[j], tmp03); tmp04 = tmp02 + 1; _ST (&a[tmp01], tmp04); </pre> |
|---|------------|---|

| | | |
|---|-------------------------------|--|
| 7 | a[b[c[j]]] = b[a[c[j]]] + 10; | <pre> _LD(&c[j],tmp01); _LD(&a[tmp01],tmp02); _LD(&b[tmp02],tmp03); _LD(&c[j],tmp04); _LD(&b[tmp04],tmp05); tmp06=tmp03 + 10; _ST(&a[tmp05],tmp06); </pre> |
|---|-------------------------------|--|

| | | |
|---|------------------------------------|--|
| 8 | a[j] = -b[j] - a[j] - c[j] - a[j]; | <pre> _LD(&a[j],tmp01); _LD(&c[j],tmp02); _LD(&a[j],tmp03); _LD(&b[j],tmp04); tmp05 = -tmp01 - tmp02 - tmp03 - tmp04; _ST(&a[j],tmp05); </pre> |
|---|------------------------------------|--|

| | | |
|---|----------------------------------|--|
| 9 | a[j] = -b[j] + c[j] - c[j] + 10; | <pre> _LD(&c[j],tmp01); _LD(&c[j],tmp02); </pre> |
|---|----------------------------------|--|

| | | |
|--|--|--|
| | | <pre> _LD(&b[j],tmp03); tmp04 = 10 -tmp01 + tmp02 -tmp03; _ST(&a[j],tmp04); </pre> |
|--|--|--|

Si les variables a, b i c fossin irregulars tindríem el mateix només que amb la següents modificacions degut a que hem de fer GET de la dada per anar a memòria a buscar i portar la dada a la cache quan la necessitem, i amb PUT hem de retornar la dada a memòria principal.

| | | |
|----|------------|--|
| 10 | a[j] = 10; | <pre> tmp01=10; _GET(&a[j]); _ST (&a[j], tmp01); _PUT(&a[j]); </pre> |
|----|------------|--|

| | | |
|----|--------------|---|
| 11 | a[j] = b[j]; | <pre> _GET(&b[j]); _LD (&b[j], tmp01); _GET(&a[j]); _ST (&a[j], tmp01); _PUT(&a[j]); </pre> |
|----|--------------|---|

| | | |
|----|-------------------------------|--|
| 12 | a[b[c[j]]] = b[a[c[j]]] + 10; | <pre> GET(&c[j]); _LD(&c[j],tmp01); _GET(&a[tmp01]); _LD(&a[tmp01],tmp02); _GET(&b[tmp02]); _LD(&b[tmp02],tmp03); _GET(&c[j]); _LD(&c[j],tmp04); _GET(&b[tmp04]); _LD(&b[tmp04],tmp05); tmp06=tmp03 + 10; </pre> |
|----|-------------------------------|--|

| | | |
|--|--|--|
| | | <code>_GET(&a[tmp05]);</code> <code>_ST(&a[tmp05],tmp06);</code> <code>_PUT(&a[tmp05]);</code> |
|--|--|--|

Tots aquests casos, són el mateix amb el cas matrius, per tant podríem tenir 12 casos més substituint `a[j]` per `a[i][j]`. I en el `_LD`, `_ST`, `_GET` o `_PUT`, en comptes de tenir `&a[j]`, tindriem `&a[i][j]`.

8 IMPLEMENTACIÓ

En aquest capítol, es pretén donar una visió general de les classes i estructures usades del compilador Nanos Mercurium C/C++ en l'implementació del nostre algorisme i l'explicació en pseudocodi d'aquest.

8.1 ENTORN DE DESENVOLUPAMENT

El llenguatge emprat en el desenvolupament de l'aplicació ha estat el llenguatge C++.

El mòdul desenvolupat està a dins de `/src/tl/omp-src/`

8.2 INTRODUCCIÓ A TL

Anteriors versions del compilador Nanos Mercurium C/C++ usaven un template²⁰ per fer les transformacions. Aquest template era anomenat TPL. Encara que aquest tipus de template, permetia proves ràpides d'algunes construccions sempre es va quedar curt en expressivitat.

Per això es va impulsar un compilador de fases escrita en C++.

Encara que aquest nom no és oficial, TL significa Transformation Library (Biblioteca de transformació).

8.2.1 Compilador de fases

El compilador de fases s'executa després que el compilador ha analitzat el codi d'entrada. El compilador de fases està escrit en C++ i són llibreries que es carreguen dinàmicament.

Cada fase del compilador rep un DTO²¹ amb una informació bàsica per treballar: la unitat de traducció del AST²² i un *scope link* adequat.

Les fases afegeixen informació a aquest DTO, per la qual cosa les fases posteriors poden utilitzar la informació sintetitzada per les anteriors. Això fa que aquesta part es comporti més com un *pipeline* de fases.

²⁰ Mena de llenguatge interpretat intercalat amb fragments de codi.

²¹ Data Transfer Object (Objecte de transferència de dades) és un patró de disseny utilitzat per transferir dades entre subsistemes de l'aplicació.

²² Abstract Syntax Tree (Arbre de sintaxis abstracte)

Una fase que vol aplicar una transformació té que modificar el AST. Per fer això pot utilitzar diversos enfocaments, però el principal mètode utilitzat en mcxx²³ es la creació de codi font sobre la marxa. Aquest nou font pot ser llavors més tard recorregut i usat com qualsevol arbre procedent de la font original. Els arbres generats novament són usats normalment per substituir els ja existents, però també es pot utilitzar com a un pas intermedi per altres transformacions.

8.2.1.1 AST

El AST és la representació sintetitzada del codi font. En contrast amb molts altres compiladors, el AST del mcxx intenta captar molts més detalls sintàctics (normalment innecessaris). Això afegeix complexitat al compilador, però fa feliços als usuaris, especialment quan han de comprovar el codi de sortida, tractant de comprendre el que està malament.

El AST és justament un doble link d'arbre 4-ari. Encara que normalment quatre fills són suficients per a la majoria de situacions, a vegades un cinquè o sisè es necessari.

Definició del AST

El tipus de AST es defineix en un tipus opac struct AST_tag. Aquest es declara en 'frontend/cxx-ast-decls.h'. L'actual implementació es troba a 'frontend/cxx-ast.c'. La definició es la següent:

```
struct AST_tag
{
    node_t node_type;
    int num_children;
    struct AST_tag* parent;
    struct AST_tag* children[4];
    int line;
    const char* filename;
    const char* text;
    int num_ambig;
    struct AST_tag** ambig;
    struct type_tag* expr_type;
    char expr_is_lvalue;
    extensible_struct_t* extended_data;
};
```

²³ Compilador Nanos Mercurium C++/C

node_type: Aquest és el tipus d'arbre, es un enter que identifica l'entitat representada en aquest arbre. Està representat simbòlicament per un **enum**. Aquest tipus d'enum es creat després del contingut de l'arxiu '**frontend/cxx-asttype.h**'. Aquest fitxer conté un nom de tipus d'arbre per línia i aquest es usat per generar el **cxx.asttype.c** en temps d'execució. Dos valors especials que existeixen per un tipus d'arbre id són: **AST_INVALID_NODE** representa un arbre amb un tipus id 0 i representa un arbre mal format internament i **AST_LAST_NODE** és el tipus més alt d'id possible i tot arbre vàlid hauria de tenir un valor entre **AST_INVALID_NODE** i **AST_LAST_NODE**.

num_children: Nombre de fills. Aquest valor és fixat per la macro **ASTMake**.

Parent: Un punter al pare, excepte per al nivell arrel de l'arbre.

children: És un *array* als 4 fills de l'arbre.

Line: Número de la línia de l'arbre en el fitxer **filename**.

filename: Nom de l'arxiu on aquest arbre es recorregut.

text: Aquest camp guarda informació textual relacionada amb el *token*. Moltes vegades aquest camp es **NULL** i aquest es usat només pel nom simbòlic (com variables) en el codi.

num_ambig: Aquests dos camps **num_ambig** i **ambig** només serveixen quan un **node_type** és **AST_AMBIGUITY**. **num_ambig** és el nombre d'ambigüitats del arbre.

ambig: és un conjunt d'arbres que representen a totes les possibles interpretacions.

exp_type: Aquest camp emmagatzema un punter a un tipus que representa el tipus de l'expressió.

exp_is_lvalue: Quan el camp **exp_type** no és **NULL**, aquest camp representa si l'expressió és un **lvalue** o no.

extended_data: Aquest camp emmagatzema l'estructura.

8.2.1.2 SCOPE LINK

No estem emmagatzemant el context relacionat amb el AST i només amb una referència a un arbre és bastant inútil perquè no disposem de la informació simbòlica.

En lloc d'emmagatzemar el *scope* en un arbre, fer-ho portaria informació contextual que podria ser problemàtica quan es mouen els arbres lliurement, per tant s'ha definit una estructura de *scope link* que és capaç, donat un arbre, recuperar el seu *scope* mitjançant l'ús d'un mapa.

En aquest mapa tots els contextos són guardats. Quan un arbre demana el seu context, l'arrel realitza la recerca per l'arbre.

Aquesta estructura és declarada a '**frontend/cxx-scopelink-decls.h**' com un tipus opac definit a '**frontend/cxx-scopelink.c**'.

```

typedef
struct scope_link_tag
{
    Hash* h;
    decl_context_t global_decl_context;
} scope_link_t;

```

h : L'emmagatzemament de hash de les entrades `scope_link_entry_t`.

global_decl_context: És el context global usat quan cap altre context es trobat a l'arbre.

Les entrades del `scope_link` només mantenen un context.

```

typedef struct scope_link_entry_tag
{
    decl_context_t decl_context;
} scope_link_entry_t;

```

8.2.2 Classes *WRAPPER*

En lloc de treballar amb les estructures internes del compilador TL ofereix diverses classes de C++ embolicant totes aquestes estructures. El seu objectiu és una interfície més agradable i també per proporcionar més llibertat.

La següent taula resumeix la relació entre les classes i les entitat *wrapped*.

| Entitat del compilador | Classe Wrap |
|------------------------|---------------|
| AST | TL::AST_t |
| type_t* | TL::Type |
| scope_entry_t* | TL::Symbol |
| decl_context_t | TL::Scope |
| scope_link_t* | TL::ScopeLink |

Classe **TL::Object** és una classe base per a tots aquests tipus i proporciona mètodes virtuals . Actualment, només **TL::AST_t** proporciona aquestes facilitats.

8.2.3 Creant/Omplint el SOURCE

Al dur a terme transformacions, la principal tècnica consisteix en la classe **TL::Source**. Aquesta classe, pot ser vista com una poderosa cadena, s'utilitza per crear el codi font sobre la marxa.

Una característica interessant d'aquesta classe és el fet que permet omplir la cadena després d'haver-la utilitzat, primer establim el codi i llavors l'omplim.

Per exemple:

```
// Definició de els source
Source src;
Source initialization_src, execution_src;
src
    << initialization_src
    << execution_src
    ;
...
// Omplim els sources
initialization_src
    << "int a;"
    ;
execution_src
    << "a = 3;"
    ;
```

Quan el *source* `src` és reconegut aquest usará el contingut de `initialization_src` i `execution_src` en el moment del recorregut. `TL::Source::operator<<` és usat per afegir o bé un altre `TL::Source`, un `std::string` o un `int`.

8.2.4 Recorrent al SOURCE creat

Un `TL::AST_t` pot ser obtingut recorrent un `TL::Source`.

El recorregut és a vegades una operació una mica fràgil i aquesta requereix context. Mentre algunes estructures són menys exigents en el context que d'altres, reconèixer el codi ha de ser, en la mesura del possible, sempre vàlid (algunes vegades no es possible, per exemple, quan referenciem una variable que encara no existeix en el context original, aquesta serà declarada més tard). Quan el reconeixement falla per això, recorrem el fals codi definint un nou context amb la informació necessària.

Dues coses són necessàries sempre quan reconeixem un codi: un arbre de referència (`TL::AST_t`) i un vàlid *scope link* (`TL::ScopeLink`). L'arbre de referència més el *scope link* són usats per recuperar el context vàlid quan el nou arbre sigui recorregut. Noteu que el codi que declara les entitats, crearà nous símbols en el context del reconeixedor, i també farà que es creïn nous contextos.

Scope link sempre es pot obtenir del DTO (hi ha, actualment, només un *scope link* en el compilador). L'arbre de referència és un arbre amb el mateix context que el codi reconegut, normalment és l'arbre que serà substituït.

8.2.5 Tractar amb els constructors del llenguatge comú

Per fer front a construccions del llenguatge comú en C i C++, TL proporciona classes *wrapping* (embolcall). Totes elles procedeixen de `TL::LangConstruct`. Cada `TL::LangConstruct` és construïda usant un arbre i un *scope link*. Es resumeixen aquestes classes:

TL::Declaration : Una declaració d'una entitat en C o C++. Una declaració sempre es definida per **TL::DeclarationSpec** i per zero o més **TL::DeclarationEntity**.

TL::DeclarationEntity: Una entitat declarada en una declaració.

TL::DeclarationSpec: Envolta l'especificació d'una declaració d'una declaració donada. Aquesta permet obtenir la **TL::TypeSpec** implicats en la declaració.

TL::Expression: Representa alguna expressió en C/C++. Aquesta classe permet navegar fàcilment a través de l'expressió i aconseguir el seu tipus (sempre que hagi estat calculat correctament).

TL::FunctionDefinition: Aquesta classe envolta tota la definició d'una funció.

TL::IdExpression: Aquesta classe envolta una ocurrència d'una referència a un símbol.

TL::ParameterDeclaration: Aquest tipus especial de declaració és just per paràmetres obtinguts usant **TL::DeclarationEntity** que representa les declaracions de funcions.

TL::Statement: Una declaració genèrica. Permet saber si la declaració és una declaració composta i obtenir les seves declaracions interiors.

TL::ForStatement: Una classe especialitzada per **TL::ForStatement** per tractar bucles for.

TL::TypeSpec: En **TL::DeclarationSpec**, és l'arbre que representa el tipus bàsic de la declaració.

8.2.6 Functors i signals

En diversos llocs el compilador defineix una estratègia de *callback*. Per aplicar amb seguretat *callback* en C++, TL proporciona un subconjunt de servies *callback*, anomenades **TL::Functor** i **TL::Signal1**.

TL::Signal1 és l'entitat que gestiona la *callback*. Pot ser connectat per diversos *functors* i el seu codi s'executarà quan rebí el *signal*. Un **TL::Functor** envolta una funció de recepció d'un paràmetre (o un objecte implícit, per les funcions membre). Això permet un enfocament homogeni quan es tracta amb aquests estils de crides de retorn (*callback*).

Un *functor* pot ser creat explícitament, per la planificació d'una nova classe que hereta de **Functor <RET, Type>** on **Ret** és el tipus de retorn i **Type** és el paràmetre de tipus. La majoria de les vegades **Ret** és void. Una excepció important a aquesta són **TL::Predicate<T>** que són exactament **Functor<bool,T>** i són normalment utilitzat per provar les propietats, en particular quan es travessa els arbres.

Una altra manera d'obtenir un *functor* es usant la funció auxiliar **TL::functor** que es pot aplicar a molts tipus de *functor* d'objectes similars i retorna un adequat **TL::functor** per a això.

Qualsevol classe que hereta de la `TL::Functor <Ret, type>` haurà d'implementar `Ret TL::functor <Ret, type>::operator() (Type& T) const`.

8.2.6.1 PREDICATS SOBRE EL AST

El predicat més habitual és `TL::PredicateAST<ATTR-NAME>`. El paràmetre `ATTR-NAME` és un atribut d'estat del AST indicant la propietat de la mateixa. Per exemple `TL::PredicateAST <LANG_IS_FOR_STATEMENT>` retorna `true` per als arbres que representen un `for-statement`.

La llista d'atributs relacionats amb els arbres és definida a `'cxx-attrnames.def'`. Per la majoria dels atributs hi ha documentació i estan relacionats amb l'arbre de la fase semàntica als fitxers `'cxx-buildscope.c'` i `'cxx-exprtype.c'`.

8.2.7 Travessant l'arbre

El recorregut per l'arbre, troba els ítems potencials per ser transformats i són implementats usant `TL::DepthTraverse` (una subclasse de `TL::Traverse`, encara que cap altra classe deriva d'aquesta).

Un `TL::TraverseFunctor` és una classe que defineix dos mètodes `preorder` i `postorder` que s'executaran (funció `run`) en el moment de `preordre` i `postordre` al passar pel node que coincideix amb els `functor`.

La coincidència d'un arbre és definida per `TL::TraverseASTFunctor`. Aquest `functor` rep un `AST_t` i retorna un `ASTTraverseResult`. Aquest valor indica si el node actual coincideix (si ho fa, llavors les seves funcions de `postorder` i `preordre` del `functor` relacionat, seran cridades).

Un `TL::TraverseASTFunctor` està relacionat amb un `TL::TraverseFunctor` per mitjà del mètode `TL::DepthTraverse::add_functor`.

Una vegada que tots els `funcctors` que ens interessin són afegits en l'objecte `TL::DepthTraverse`, podem disparar el recorregut usant `TL::DepthTraverse::traverse`.

8.3 PSEUDOCODI DE L'ALGORISME

```
// Inici algorisme implementat per cada expressió del cos del bucle
{Prerequisits: #pragma dissenyat indicant tipus d'accessos a memòria de les variables
implicades ; Bucle FOR; Expressions vistes a apartat 2.3.5.2}
```

VA = Variables d'accessos d'alta localitat

VI = Variables d'accessos irregulars

```
// Agafem totes les expressions dintre del cos del bucle (loop_body) i les guardem a la variable
expressió
```

```
expressió = expressions cos del bucle;
```

```
per cada expressió fer
```

```
  si expressió es correcte llavors
```

```
    /* Recorrem tota l'expressió mirant quines variables seran LOAD i quines
    STORE i les guardarem en dues llistes per més endavant fer el procés tot
    unificat. */
```

```

/* També mirem quin tipus de càlcul o expressió són: suma, resta, ... i els signes
dels operands */
varsLD = Llista variables a fer LOAD;
varsST = Llista variables a fer STORE;
// Comencem la transformació del cos del bucle
// Per cada variable de les dues llistes hem de crear unes variables per al CELL
per cada varsLD fer
    si NOT enter llavors
        // Per cada variable de la llista er els seus corresponents
        init_mCELL();
        init_lookup();
        next_miss();
        updateDirtyBits();
        Loads();
        CàlculsTemporals();
    sino
        CàlculsTemporals();
    fi si
fi per
per cada varsST fer
    si NOT ARRAY llavors
        // Per cada variable de la llista er els seus corresponents
        init_mCELL();
        init_lookup();
        next_miss();
        updateDirtyBits();
        Stores();
    Sino
        /* Si es un array hem de fer LOADS de les variables de dins de
l'array */
        // Per cada variable de la llista er els seus corresponents
        init_mCELL();
        init_lookup();
        next_miss();
        updateDirtyBits();
        Loads(); //variables dintre del array
        /* Variable primera. Ex: a[c[i]] es LOAD de c[i] i ho guardem a
tmp i llavors STORE (a[tmp])
        Stores();
    fi si
sino
    següent;
fi si
fi per

//Fi algorisme

```

// Funcions

init_mCELL(); → Per cada variable inicialitza les variables necessàries per al CELL BE que són el `_trans_01`, `_mask_01` i `_baddr_01`.

init_lookup(); → Per cada variable, inicialitzem una variable de lookup, mirem si existeix la dada necessària a cache (macro `_LOOKUP`), si no existeix la portem de memòria principal (macro `_MMAP_NEW`).

next_miss(); → Executa la macro `_NEXT_MISS`

updateDirtyBits(); → Marca les línies de cache que haurà de tornar a memòria principal al final de l'execució, és a dir les dades que executaran STORE.

Stores() → Guardem el resultat a cache.

Per cada dada irregular també executarem GET abans del STORE de la dada, per portar la dada de memòria principal a la cache, i després del STORE farem PUT per retornar la dada a memòria principal.

Loads() → Carreguem la dada de cache.

Per cada dada irregular també executarem GET abans de LOAD de la dada, per portar la dada de memòria principal a la cache.

CàlculsTemporals() → Aquesta funció fa els càlculs pertinents de l'expressió a una variable temporal per més tard poder fer STORE a la variable destí d'aquesta variable temporal.

9 AVALUACIÓ

Després d'implementar el nostre algorisme, es desitja conèixer el grau d'efectivitat. Per aquest motiu en aquest apartat veurem la metodologia emprada per a la realització dels jocs de proves que siguin representatius als programes reals.

Els jocs de proves s'han construït amb l'objectiu de comprovar si l'algorisme gestiona bé totes les possibles expressions del cos del bucle. El punt més important de la fase d'avaluació és tenir una visió general de com funciona l'algorisme. El que es volia estressar en els jocs de proves era la correctesa de la generació de codi en la part d'expressions del cos del bucle i dels dos tipus d'accessos a memòria per part de les variables (accessos regular i accessos irregulars).

Per tant per cada joc de proves, s'ha provat els dos tipus de variables, sempre tenint en compte que la variable del bucle (d'inducció) l'hem considerada privada i no pertany a cap dels dos tipus mencionats.

La metodologia seguida per generar els jocs de proves s'ha fet tenint en compte diversos aspectes que afecten a l'algorisme de transformació de codi i que depenen de:

- Tipus de variables: int, float, ...
- Tipus d'operadors unaris / binaris: +, -, ++, --, *, ...
- Número de variables: Total de variables implicades.
- Tipus d'accessos a memòria: Regular o irregular.
- Número de bucles.

A la següent taula englobem aquests per conjunts i per cadascun veiem els noms dels jocs de proves a on s'estressa i el també el número de cas que és fa referència, aquests números de cassos es podem veure en l'apartat de disseny 7.4.2.

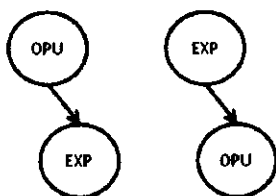
| | | |
|---------------------------|--|------------------------------|
| Expressions amb constants | prova10*.c prova11*.c prova12*.c prova13_v4.c prova13_v5.c prova14*.c prova15_v2.c prova16*.c prova17*.c | 1,7,10 i 12 |
| Expressions amb vectors | prova01*.c prova02*.c prova03*.c prova04*.c prova05*.c prova06*.c prova07*.c prova08*.c prova09*.c prova10*.c prova11*.c prova12*.c prova13*.c prova14*.c prova15*.c prova16*.c prova17*.c | 1,2,3,4,5,6,7,8,9,10,11 i 12 |

²⁴ Veure casuístiques d'exemple apartat 7.4.2

| | | |
|-------------------------------|--|------------------------------|
| Expressions amb matrius | prova01*.c prova02*.c prova03*.c prova04*.c prova05*.c prova06*.c prova07*.c prova08*.c prova09*.c prova10*.c prova11*.c prova12*.c prova13*.c prova14*.c prova15*.c prova16*.c prova17*.c | 1,2,3,4,5,6,7,8,9,10,11 i 12 |
| Expressions operadors unaris | prova04*.c prova05*.c | 5 i 6 |
| Expressions operadors binaris | prova07*.c prova08*.c prova09*.c prova10*.c prova13*.c prova14*.c prova15*.c prova16*.c prova17*.c | 7, 8, 9 i 12 |

Les expressions reconegudes avaluades i estressades en el cos del bucle són les següents:

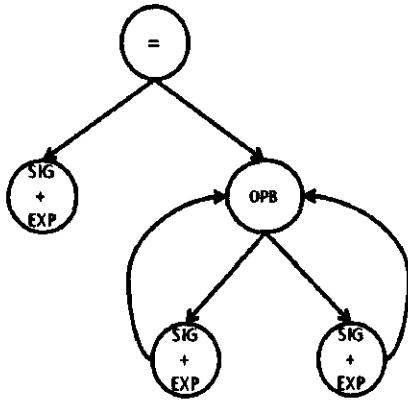
Expressions unàries



On OPU es operador unari ++ o --.

On EXP pot ser una variable a o a[j] o a[b[c[j]]] i les seves possibles derivacions.

Expressions binàries...



On OPB es un operador binari que pot ser SUMA (+) o RESTA (-).

On SIG pot ser signe POSITIU (+), NEGATIU (-) o PUNTER (*).

On SIG+EXP pot tornar-se a descompondre's en un OPB + dos SIG+EXP com es mostra en la figura.

Queda fora de l'avaluació per falta de temps i perquè es va acotar en l'apartat de disseny, tot el que queda fora de lo que hem mencionat fins ara. Per exemple els operadors MULTIPLICACIÓ i DIVISIÓ, les condicions de if dintre del bucle, les funcions, no es tracten en el nostre algorisme, per tant no funciona el nostre algorisme per aquestes situacions, i per tant no fan la generació de codi correctament.

Tenim un total de 17 jocs de proves, amb les seves diverses variants.

Veiem a continuació un exemple amb el codi generat, on el fitxer prova01.c és el codi original i el fitxer mcc_prova01.c és el codi de sortida del compilador transformat. Per generar aquest fitxers s'executa la comanda per compilar el programa, una vegada haguem compilat el compilador amb les nostres modificacions.

// Comanda per compilar el programa

mcc -c prova01.c -k -v

Per compilar el compilador, executem **make** i **make install** dins de la carpeta mcxx-build.

PROVA01.C - ORIGINAL

```
/* CODI ORIGINAL */
```

```
void f(){  
    int j;  
    int VectorSize;  
    int a[100];  
    int b[100];
```

```

j=0;
VectorSize=100;
#pragma hlc hlcached(a,b)
for (j=0; j<VectorSize; j++)
{
    a[j] = b[j];
}
}

```

MCC PROVA01.C - GENERAT

```

/* Codi generat pel compilador*/
/*...*/
void f()
{
    /* Inicialitzacions diverses */
    /* ... */
    _OPEN_MEMORY_TRACER_LIB(0);
    _DECLARE_MEMORY_TRACER_VARS();
    _DECLARE_MEMORY HOLDER_VARS();
    _DECLARE_MEMORY_TRANSLATION_VARS();
    _INIT_MCELL(1, _trans_01, _mask_01, _baddr_01);
    _INIT_MCELL(2, _trans_02, _mask_02, _baddr_02);
    _lb_01 = 0;
    _ub_01 = (VectorSize) - 1;
    _step_01 = 1;
    _NORMALIZE_LOOP(_ub_01, _lb_01, _step_01, &_norm_ub_01, &_norm_lb_01);
    _start_01 = 0;
    _end_01 = 0;
    _NEXT_ITERS(&_start_01, &_end_01, _norm_ub_01, _norm_lb_01, _STATIC_, 0, _work_01);
    while (_work_01)
    {
        _sub_start_01 = _start_01;
        _work_inside_chunk_01 = TRUE;
        while (_work_inside_chunk_01)
        {
            _ind_01 = _sub_start_01 * _step_01;
            /* BUCLE 1 */
            _INIT_LOOKUP(_lookup_01);
            _LOOKUP(1, _trans_01, _mask_01, _baddr_01, &b[_ind_01], double_, _lookup_01);
            if (__builtin_expect(_lookup_01, 0))
            {
                _MMAP_NEW(1, _trans_01, _mask_01, _baddr_01, &b[_ind_01], double_, _READ_,
                    _GLOBAL_ | _STRIDED_, EXECUTOR | RECORD);
            }
            _INIT_LOOKUP(_lookup_01);
            _LOOKUP(2, _trans_02, _mask_02, _baddr_02, &a[_ind_01], double_, _lookup_01);
            if (__builtin_expect(_lookup_01, 0))
            {
                _MMAP_NEW(2, _trans_02, _mask_02, _baddr_02, &a[_ind_01], double_, _READ_,
                    _GLOBAL_ | _STRIDED_, EXECUTOR | RECORD);
            }
        }
    }
}

```

```

_next_iters_01 = LS_PAGE_SIZE;
/* BUCLE 2 */
_NEXT_MISS(1, _trans_01, _mask_01, _baddr_01, &b[_ind_01], double, sizeof(double),
_next_iters_01);

_NEXT_MISS(2, _trans_02, _mask_02, _baddr_02, &a[_ind_01], double, sizeof(double),
_next_iters_01);

_sub_end_01 = _sub_start_01 + _next_iters_01;
if (_sub_end_01 > _end_01)
{
_sub_end_01 = _end_01;
}
_work_inside_chunk_01 = (_sub_end_01 < _end_01);

_UPDATE_DIRTY_BITS(2, _trans_02, _mask_02, _baddr_02, &a[_ind_01], double,
sizeof(double), sizeof(double), (_sub_end_01 - _sub_start_01));

_MEM_BARRIER();
for (_ind_01 = _sub_start_01;
_ind_01 < _sub_end_01;
_ind_01 = _ind_01 + 1)
{
j = _ind_01 * _step_01;
_LD(1, _trans_01, _mask_01, _baddr_01, &b[j], double, __double_tmp01);
__double_tmp02 = __double_tmp01;
_ST(2, _trans_02, _mask_02, _baddr_02, &a[j], double, __double_tmp02);
}
_sub_start_01 = _sub_end_01;
}
_start_01 = _end_01;
_NEXT_ITERS(&_start_01, &_end_01, _norm_ub_01, _norm_lb_01, _STATIC_0, _work_01);
}
_DECREMENT_REFERENCE(0);
_DECREMENT_REFERENCE(1);
_DECREMENT_REFERENCE(2);
_CLOSE_MEMORY_TRACER_LIB();
}

```

10 COST I PLANIFICACIÓ

En aquest capítol es presenta la planificació del projecte, juntament amb un anàlisi econòmic. La planificació està feta tenint en compte que només ha desenvolupat el projecte una persona. Al ser un projecte de recerca, és difícil preveure les etapes i la duració d'aquestes, ja que depenen de com vagi evolucionant l'aprenentatge de les parts implicades. Aquest aprenentatge fa que una etapa que en principi podia semblar més costosa no ho sigui tant, o a l'inrevés. Tot i això, es va intentar establir un ordre de feines a realitzar, amb la seva durada aproximada, en la planificació inicial, i finalment veurem en la planificació final petites desviacions i els seus motius.

10.1 COST DEL PROJECTE

Aquest projecte està plantejat com un projecte de recerca. Farem l'anàlisi del cost del projecte segons el maquinari i programari emprat i les hores que han sigut necessàries per a la realització del projecte.

El maquinari utilitzat ha sigut un portàtil i suposa un cost afegit a la realització del projecte de 900 € (Sistema operatiu Windows + Microsoft Office 2007 basic inclosos).

Pel que fa al programari, s'han utilitzat eines de lliure distribució que no suposen un cost afegit:

- Sistema Operatiu Fedora versió 9 (Sulphur)
- Nucli Linux 2.6.27.21-78.2.41.fc9.i686
- Compilador Nanos Mercurium C/C++ versió 1.2.1.0
- Compilador GCC 4.3.0
- Flex versió 2.5.35
- Bison versió 2.3-rofi
- Editor Eclipse versió 3.4.0
- Planificació - Diagrames GANNT : Planer 0.14.2
- Openoffice 3.0

Tant el maquinari com sobretot el programari utilitzat han suposat un estalvi econòmic important. No obstant, el coneixement necessari per a treballar amb aquest programari s'ha traduït en una inversió en hores per aprendre a utilitzar-ho, i sobretot la fase d'aprenentatge

de l'estructura i disseny dels components del projecte: el compilador Nanos Mercurium C/C++, la software cache híbrida i el Processador CELL BE, que són la base del projecte.

El projecte s'ha dividit en les següents fases:

- 1) Aprendre l'arquitectura del processador multi nucli CELL BE i de la software cache híbrida.
- 2) Aprendre l'estructura i disseny del compilador ja existent.
- 3) Dissenyar el nou mòdul per desenvolupar la part del nostre projecte dintre del compilador.
- 4) Implementar el nou mòdul: Desenvolupament del mòdul dintre del compilador.
- 5) Avaluació: Disseny del joc de proves per provar el correcte funcionament de tot el mòdul.
- 6) Documentació: Aquesta fase té com a objectiu l'escriptura d'aquest document.

10.2 PLANIFICACIÓ INICIAL

La nostre planificació inicial la veiem a la taula següent:

| | | | | | |
|--------------|-----------|-----------|----------|-----------|------------------|
| 1 | 40 hores | | | | 40 hores |
| 2 | 100 hores | | | | 100 hores |
| 3 | 80 hores | | | | 80 hores |
| 4 | | 300 hores | | | 300 hores |
| 5 | | 25 hores | 45 hores | | 70 hores |
| 6 | | | | 120 hores | 120 hores |
| TOTAL | | | | | 710 hores |

A continuació adjuntem el diagrama de gannt de la planificació inicial.

10.3 PLANIFICACIÓ FINAL

| 1 | 40 hores | | | | 40 hores |
|--------------|-----------|-----------|----------|-----------|------------------|
| 2 | 120 hores | | | | 120 hores |
| 3 | 60 hores | | | | 60 hores |
| 4 | 20 hores | 340 hores | | | 360 hores |
| 5 | | 50 hores | 55 hores | | 105 hores |
| 6 | | | | 110 hores | 110 hores |
| TOTAL | | | | | 795 hores |

Al ser un projecte d'investigació el perfil de la persona que fa el projecte és el d'un analista.

El cost final del projecte es de 645 hores * 50 €/hora = 32.250 €

+ 900 € (maquinari+software)

33.150 €

A continuació adjuntem el diagrama de gannt de la planificació final.

| | | 2009, H1 | | | | | | | | | | | | | | |
|-----|---|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|------|
| | | jun 2008 | 2008, H2 | ago 2008 | set 2008 | oct 2008 | nov 2008 | des 2008 | 2009, H1 | gen 2009 | feb 2009 | mar 2009 | abr 2009 | may 2009 | jun 2009 | 2009 |
| | | jun 2008 | 2008, H2 | ago 2008 | set 2008 | oct 2008 | nov 2008 | des 2008 | 2009, H1 | gen 2009 | feb 2009 | mar 2009 | abr 2009 | may 2009 | jun 2009 | 2009 |
| WBS | Nom | Feina | | | | | | | | | | | | | | |
| 1 | Apendre CELL BE + Software Cache iBanda | :20d | | | | | | | | | | | | | | |
| 3 | Disseny | :50d | | | | | | | | | | | | | | |
| 4 | Implementació | :110d | | | | | | | | | | | | | | |
| 5 | Avaluació | :20d | | | | | | | | | | | | | | |
| 6 | Documentació | :20d | | | | | | | | | | | | | | |

10.4 DESVIACIONS DE LA PLANIFICACIÓ

Com es pot veure en les taules, hi ha una desviació de la planificació entre la inicial i la final, els motius són:

- Falta d'experiència en planificacions.
- L'estudi del compilador Nanos Mercurium C/C++ ens ha suposat un esforç més important del que pensàvem, degut a que no hi ha gaire documentació.
- La part d'implementació es la que pateix la desviació més gran, degut a alguns problemes amb alguns errors del compilador per part de la nostre implementació que com he comentat abans no hi ha gaire documentació, i quan sorgia algun problema era complicat de resoldre.
- Per que fa al temps, es va planificar que s'acabaria el 25/03/09, però degut a treballar a jornada completa i altres inconvenients, s'ha endarrerit 3 mesos.

11 CONCLUSIONS I TREBALL FUTUR

Els objectius plantejats a l'inici del Projecte s'han complert satisfactòriament, tot i que hi ha una sèrie de punts que, per falta de temps, no s'han afrontat amb la profunditat desitjada. A més, a mesura que s'ha anat desenvolupant el projecte han sorgit idees que podrien servir si es volgués continuar aquest treball. Aquests dos aspectes són els que es presenten en aquest darrer capítol d'aquesta memòria.

11.1 CONCLUSIONS

L'estudi, disseny i implementació d'un algorisme que ens generes les transformacions de codi pertinents per un codi d'entrada per a ser executat en el CELL BE era l'objectiu inicial del projecte. Podem dir que aquest objectiu s'ha cobert.

Les principals limitacions de l'algorisme són les expressions reconegudes en el cos del bucle per a la generació de codi corresponent, això es va acotar en l'apartat de disseny, degut a la infinitat d'expressions, condicions, funcions, etc que pot contindre el cos del bucle. Aquesta acotació va ser en funció dels tipus de codis proposats per fer la generació.

A nivell personal el projecte m'ha aportat coneixements molt diversos: models de programació, arquitectura cell be, diferents línies de recerca, etc. El que més m'ha costat es l'inici del projecte, degut a que els components que s'han treballat són tots de línies de recerca, dificultant trobar documentació, per tant la majoria de documentació que s'ha estudiat són papers dedicats al CELL BE, la software cache híbrida i altres,...

11.2 TREBALL FUTUR

Com a treball futur s'ha pensat en els següents aspectes depenen de les necessitats i dificultats dels codis que vulguin transformar els programadors. Els més importants serien:

- Optimitzacions de les nostres transformacions, ja que el codi no es òptim. Per exemple en l'expressió `a[b[j]]++`; fa dos LOADS a `b[j]`, degut a que la variable es de lectura i escriptura a la vegada.

| Expressió | Codi a generar |
|-------------------------|---|
| <code>a[b[j]]++;</code> | <pre> _LD (&b[j], tmp01); _LD (&a[tmp01], tmp02); _LD (&b[j], tmp03); tmp04 = tmp02 + 1; _ST (&a[j], tmp04); </pre> |

Un altre exemple diferent seria:

| Expressió | Codi a generar |
|--|--|
| <code>a[b[c[j]]] = b[a[c[j]]] + 10;</code> | <pre> _LD(&c[j],tmp01); _LD(&a[tmp01],tmp02); _LD(&b[tmp02],tmp03); _LD(&c[j],tmp04); _LD(&b[tmp04],tmp05); tmp06=tmp03 + 10; _ST(&a[tmp05],tmp06); </pre> |

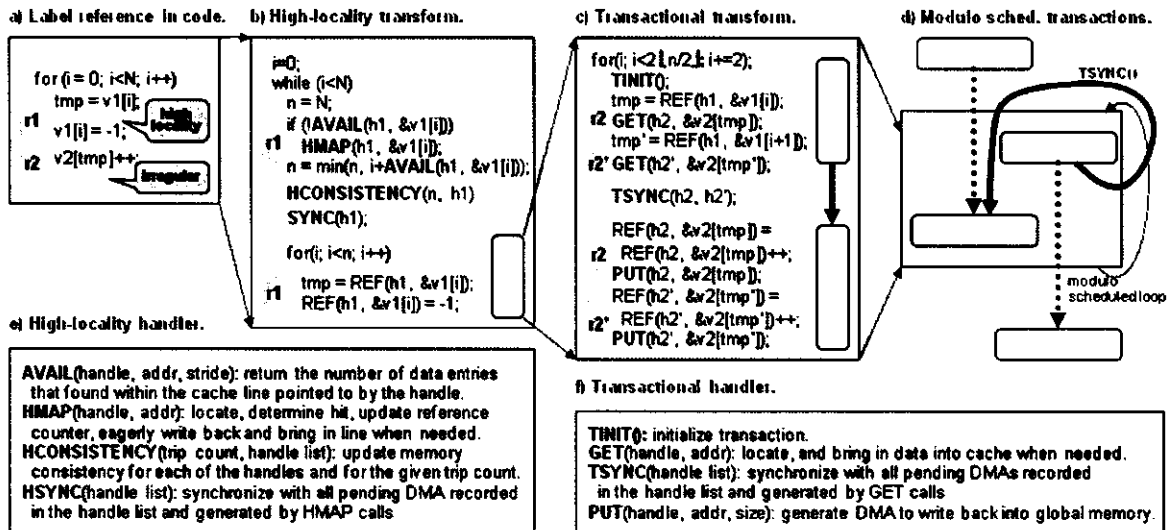
- En comptes que ens hagin d'indicar quines són les referències d'accessos d'alta localitat (regulars) i accessos irregulars detectar-les prèviament analitzant el codi.
- Aplicar les transformacions permeten bucles niats, es a dir:

```

#pragma hlc hlcached(...) tcached(...)
for (j=0; j<VectorSize; j++)
{
    // Qualsevol expressió
    #pragma hlc hlcached(...) tcached(...)
    for(i=0;j<VectorSize;i++)
    {
        // Qualsevol expressió
    }
    // Qualsevol expressió
}

```

- Reconeixement de més tipus d'expressions o la possibilitat de tenir condicions, funcions, etc en el cos del bucle i fer la generació de codi pertinent en cada cas.
- A partir de la transformació/generació de codi feta per la cache d'alta localitat i la cache transaccional, per incrementar la concurrència, es podria aplicar una tècnica coneguda com *mòdul scheduling* o *dobles buffering* com es veu en l'aparat d) de la següent figura.



ANNEX A : INSTAL·LACIÓ I CONFIGURACIÓ NANOS MERCURIUM C/C++

La instal·lació s'ha realitzat sobre una distribució linux Fedora 9. La versió de Mercurium C/C++ utilitzada és la 1.2.1.1.

Passos

- 0) Descomprimir el paquet descarregat MERCURIUM.tar.gz al path per exemple /home/sandra/PFC/MERCURIUM
- 1) Primer de tot instal·larem dos paquets que necessitem, el bison i el gpref que es localitzen a la carpeta soft.

Instal·lació del bison:

```
# cd /home/sandra/PFC/MERCURIUM/soft/bison-2.3-rofi
# ./configure
# make
# make install
```

Instal·lació del gpref:

```
# cd /home/sandra/PFC/MERCURIUM/soft/gpref-3.0.3
# ./configure
# make
# make install
```

- 2) Comprovem la versió de flex que tenim. Requeriments: flex == 2.5.4 o >=2.5.33

```
# flex -V
```

Nosaltres tenim 2.5.35 per tant estem dins els requeriments.

- 3) Instal·lem un paquet de nanos4 necessari:

```
# cd /home/sandra/PFC/MERCURIUM/nanos4-src
# autoreconf -i --force
# ../nanos4-src/configure --prefix=/home/sandra/PFC/MERCURIUM/install
# make
# make install
```

- 4) Instal·lem ara el compilador:

```
# cd /home/sandra/PFC/MERCURIUM/mcxx-build/
# ../mcxx-src/configure --disable-tl-instrumentation
--with-nanos4=/home/sandra/PFC/MERCURIUM/install
# make
# make install
```

Problemes trobats en els passos anteriors

- 1) La distribució de Fedora 9 portava la versió 4.3.0 de gcc, per tant s'ha hagut d'aplicar un "parche" que es el que segueix:

INICI FITXER patch_gcc_4.3

Index: src/frontend/cxx-exprtype.c

```
=====
--- src/frontend/cxx-exprtype.c (revision 2535)
+++ src/frontend/cxx-exprtype.c (revision 2536)
@@ -5676,7 +5676,7 @@
 {
 char c[256];
 snprintf(c, 255, "<surrogate-function-%d>", num_surrogate_functions);
- c[256] = '\0';
+ c[255] = '\0';
 surrogate_symbol->symbol_name = uniquestr(c);
 }
```

Index: src/tl/tl-ast.cpp

```
=====
--- src/tl/tl-ast.cpp (revision 2535)
+++ src/tl/tl-ast.cpp (revision 2536)
@@ -27,6 +27,7 @@
#include <sstream>
#include <cstdio>
#include <cerrno>
+#include <cstring>
namespace TL
{
Index: src/tl/tl-objectlist.hpp
```

```
=====
--- src/tl/tl-objectlist.hpp (revision 2535)
+++ src/tl/tl-objectlist.hpp (revision 2536)
@@ -23,6 +23,7 @@
#include <vector>
#include <utility>
+#include <algorithm>
#include "tl-functor.hpp"
#include "tl-predicate.hpp"
#include <signal.h>
Index: src/tl/tl-source.cpp
```

```
=====
--- src/tl/tl-source.cpp (revision 2535)
+++ src/tl/tl-source.cpp (revision 2536)
@@ -24,6 +24,7 @@
#include <iostream>
```

```
#include <sstream>
#include <iomanip>
+#include <cstring>
#include "cxx-printscape.h"
#include "cxx-utils.h"
Index: src/tl/tl-object.cpp
```

```
=====
--- src/tl/tl-object.cpp (revision 2535)
+++ src/tl/tl-object.cpp (revision 2536)
@@ -24,6 +24,7 @@
#include "tl-symbol.hpp"
#include <cstdlib>
+#include <cstring>
namespace TL
{
```

FI FITXER patch_gcc_4.3

Aquest parche es degut a la versió de gcc 4.3.0 que ara són més rigorosos en l'estàndard c++ i s'han hagut d'afegir alguns headers. S'ha aplicat el parche executant:

```
# patch -p0 < patch_gcc_4.3
```

- 2) Un altre problema ha sigut la versió de libstdc++, el distribució fedora 9 porta la versió libstdc++.so.6.0.10 i en canvi tenim un error de compilació que no troba la llibreria libstdc++.so.5 a /usr/lib/

S'ha instal·lat el paquet de compatibilitat amb la llibreria estàndard del gcc 3.3.4 que també conté el libstdc++.so.5.

BIBLIOGRAFIA

- [1] Wikipedia: http://es.wikipedia.org/wiki/Pentium_D
- [2] Wikipedia: http://es.wikipedia.org/wiki/Intel_Core_2
- [3] Wikipedia: http://es.wikipedia.org/wiki/Core_2_Duo
- [4] Wikipedia: http://es.wikipedia.org/wiki/Core_i7
- [5] Wikipedia: http://es.wikipedia.org/wiki/Athlon_64_X2
- [6] Wikipedia: http://es.wikipedia.org/wiki/AMD_Turion_64_X2
- [7] Wikipedia: <http://en.wikipedia.org/wiki/Opteron>
- [8] Wikipedia: [http://en.wikipedia.org/wiki/Phenom_\(processor\)](http://en.wikipedia.org/wiki/Phenom_(processor))
- [9] Wikipedia: http://en.wikipedia.org/wiki/Phenom_II
- [10] Wikipedia: http://es.wikipedia.org/wiki/PowerPC_G5
- [11] Wikipedia: http://en.wikipedia.org/wiki/CELL_BE
- [12] E. Miraya Anamaria, "Processadores multi-núcleo", in <http://www.monografias.com>
- [13] Wikipedia: <http://en.wikipedia.org/wiki/Openmp>
- [14] Wikipedia: http://es.wikipedia.org/wiki/Interfaz_de_Paso_de_Mensajes
- [15] Wikipedia: http://es.wikipedia.org/wiki/Intel_Threading_Building_Blocks
- [16] P. Bellens, J. M. Perez, R.M. Badia and J. Labarta, "CellSs: a Programming Model for the Cell BE Architecture"
- [17] <http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html>
- [18] <http://www.trimaran.org>
- [19] <http://ipf-orc.sourceforge.net/>
- [20] C. A. Moritz et al., "FlexCache: A framework for flexible compiler generated data caching," in the Proceedings of the 2nd Workshop on Intelligent Memory Systems, 2000.
- [21] J. B. Fryman et al., "SoftCache: A Technique for Power and Area Reduction in Embedded Systems," CERCS; GIT-CERCS-03-06
- [22] J. E. Miller and A. Agarwal, "Software-based Instruction Caching for Embedded Processors," in the Proceedings of the Conference on Architectural Support for Programming Languages and Operating Systems, 2006.
- [23] M. Kruijf and K. Sankaralingam, "MapReduce for the Cell BE Architecture"
- [24] A. E. Eichenberger et al., "Using advanced compiler technology to exploit the performance of the Cell Broadband Engine architecture," IBM Systems Journal, Vol. 45, No. 1, 2006.
- [25] Official OpenMP specifications.

<http://www.openmp.org/specs/mp-documents/cspec20.pdf>.

[26] F. Blagojevic, D. S. Nikolopoulos, A. Stamatakis, and C. D. Antonopoulos. "Dynamic multigrain parallelization on the Cellbroadband engine". In PPOPP '07: Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pages 90–100, New York, NY, USA, 2007. ACM Press.

[27] http://www.blachford.info/computer/Cell/Cell0_v2.html

[28] Ganapathy Senthil, Sasikanth Gudla i Pallav Kumar Baruah, "Exploring Software Cache on the CELL BE Processor"

[29] M. González, N. Vujic, X. Martorell and E. Ayguadé, "Hybrid Access-Specific Software Cache Techniques for the CELL BE Architecture"

[30] J. Hoeflinger and B. de Supinski, "The OpenMP Memory Model," in the Proceedings of the First International Workshop on OpenMP, 2005.

[31] R. Ferrer, M. González, X. Martorell, E. Ayguadé, "Mercurium C/C++ source-to-source compiler"

[32] R. Ferrer "Mercurium C/C++ Internal Documentation Reference"

