POLITECNICO DI MILANO
Facoltà di Ingegneria – sede di Milano Leonardo
Corso di Laurea in Ingegneria Informatica

VERSO UNA INFRASTRUTTURA DI COMPILAZIONE PER CELL BE BASATA SU LLVM

Relatore: Ing. Giovanni AGOSTA

Elaborato di Laurea di:
Daniele MARTINELLI
Matr. 740247

Anno Accademico 2011-2012

Ai miei genitori

Ringraziamenti

Un doveroso e speciale ringraziamento va al prof. Giovanni Agosta, il mio relatore, che mi ha aiutato e supportato in modo decisivo in questo elaborato di tesi e nel suo compimento. Le sue critiche costruttive, il continuo incitamento e la sua immensa disponibilità mi hanno infuso una forte carica spingendomi a credere in quello che stavo facendo.
Un ringraziamento di cuore va ai miei genitori, alla mia mamma recentemente scomparsa, che mi hanno dato l'opportunità di compiere e continuare questa fantastica esperienza universitaria, supportandomi emotivamente in modo massiccio e sincero, non smettendo mai di credere in me e consigliandomi.
Un caloroso ringraziamento anche ai compagni di corso, di pranzi, di esami, di sofferenze e di tante sane risate Mandeep, Kumar, Tomas, Franco, Max, Alle, Chiara, Sozzi, Coti, Goffre, Roma, Alessia, Ilaria, Michele, Marco, Emil, Emiliano, Spelta, Sara e Sara, Maestri.
Uno speciale ringraziamento anche ai miei fantastici amici lacustri Niki, Gio, Tommy, Ciccio, Nicolò e Campa, Chiara, Zilo, Leo, Nino, Marci, Viola, Cinzia, Amabile, Meri, Corinne e Martin, Andrea, Elisa e Francy, ai bresciani Ste, Roby e Andrea Carro e ai cremonesi Filo, Gian e Otto per l'immenso supporto che mi hanno dato in questo periodo... la loro felicità nell'apprendere la lieta notizia è stata una felicità impagabile per me!

Daniele Martinelli

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Introduzione

Questo elaborato di tesi si propone essere un miglioramento di un compilatore preesistente [1] che si basa sull'infrastruttura di compilatori *LLVM* ovvero una collezione di tecnologie che facilitano ed aiutano le fasi della compilazione e forniscono toolchain modulari e riusabili.
Il compilatore si poneva a completamento di un lavoro di sviluppo di una famiglia di frontend per LLVM scritti in linguaggio *C*, *C++*, *Objective C* e *Objective C++* con lo scopo di fornire migliori diagnostiche, migliore integrazione con gli *IDE*, una licenza compatibile con i prodotti commerciali e un compilatore leggero e veloce facile da sviluppare e mantenere.
Il compilatore originale è continuamente in sviluppo ed è considerato un prodotto di qualità per le architetture basate su *ARM* e su *X86* a 32/64 bit; è un'ottima soluzione per la *source analysis* o per tool basati su *source-to-source transformation*.
Fra le caratteristiche principali del compilatore *clang* vi sono un basso consumo di memoria, una rapida compilazione, presenza di diagnostiche espressive e una compatibilità con il compilatore *GCC*.

Principalmente lo scopo di questo lavoro di tesi è stato quello di modificare, integrare e ottimizzare la parte backend del compilatore in modo da renderlo compatibile alla compilazione di programmi destinati al processore *Cell Broadband Engine* (*Cell BE*) dell'architettura della *Play Station 3*.

[1] Stiamo parlando dell'infrastruttura di compilatori *LLVM* e del suo frontend *Clang*

Di conseguenza abbiamo reso l'applicativo più efficiente editandolo, in modo da poter rendere funzionali i programmi pensati originalmente per sfruttare le risorse del microprocessore Cell della *PS3*.
Infine abbiamo testato i miglioramenti introdotti che hanno sensibilmente incrementato l'efficienza dello strumento software in termini di memoria allocata e di tempo trascorso.

Il seguente elaborato è strutturato con un'esauriente presentazione del contesto nel quale si trova (Capitolo 2 e Capitolo 3), abbracciando pian piano i formalismi e gli strumenti utilizzati, per poi andare a toccare con mano l'applicativo (Capitolo 5) e le modifiche introdotte (Capitolo 4). Infine a corredo delle modifiche introdotte sono presentati alcuni esperimenti effettuati (Capitolo 5), evidenziando le migliorie introdotte, i colli di bottiglia ed eventuali sviluppi futuri (Capitolo 6).

Capitolo 2

Cell Broadband Engine

Il *Cell Broadband Engine* (*Cell BE*) e i processori *IBM PowerXCell 8i*, detti processori *Cell Broadband Engine Architecture* (*CBEA*), sono stati progettati da Sony, Toshiba e IBM, estendendo l'architettura *PowerPC* a 64 bit. Il *Cell BE* è la prima implementazione di una famiglia di multiprocessori conformi allo standard *CBEA*, ha un'interfaccia di memoria *double data rate 2* (*DDR2*) e performance floating point.
L'architettura di questa CPU ottiene prestazioni vicine alle massime raggiungibili. I *CBEA* hanno un singolo chip multiprocessore nel quale vi sono un'unità centrale di calcolo (*PowerPC Processor Element* o *PPE*) ed otto unità specializzate nel calcolo vettoriale (*Synergistic Processor Element* o *SPE*).
Anche se ottimizzati per specifiche operazioni, questi elementi possono scambiarsi di ruolo: la *PPE* può fare calcoli vettoriali e le *SPE* possono funzionare come una CPU, eseguendo istruzioni non vettoriali.
La logica di coordinamento è lasciata al programmatore, realizzando facilmente algoritmi che usano *SPE* per calcoli in serie e trasferiscono dati tra loro, o altri che dividono i dati e fanno calcoli in parallelo per poi raccogliere il risultato con la *PPE*.
Sebbene il *Cell BE* fu pensato per device di fascia medio-alta come game console (la *PlayStation 3* è la meno costosa tra i sistemi con *Cell BE*) e televisioni ad alta definizione, l'architettura è stata disegnata per aumentare ulteriormente le performance dei processori per supportare applicazioni commerciali e scientifiche.

Per programmare la *CBEA* bisogna essere familiari con i linguaggi *C/C++*, single-instruction multiple-data (*SIMD*), vector instruction set (l'estensione IBM *PowerPC Architecture vector/SIMD*), *AltiVec*, Intel MMX, SSE, 3DNOW!, x86 a 64 bit o instruction set VIS.
La notazione usata è quella standard IBM: bit e byte ordinati in modo ascendente da sinistra a destra (per 4 byte il bit più significativo è 0 e 31 il meno significativo).
Il registro "*memory-mapped I/O*" (*MMIO*) è un registro interno o esterno, associato con un controller I/O o un device, a cui si può accedere con istruzioni *load* e *store*. Le sue aree riservate non sono assegnate a nessuna unità funzionale e non devono essere lette o scritte altrimenti possono essere causati seri errori nel software.

2.1 Overview dei processori *CBEA*

2.1.1 Background

Motivazioni

I *CBEA* descritti sono multiprocessori single-chip con nove elementi operativi su una memoria condivisa e coerente come mostrato nel diagramma di blocco in figura 2.1.
La frequenza di funzionamento della *PPE*, delle *SPE* e della *RAM XDR* di sistema è pari a 3.2 Ghz.

I processori *CBEA* si distinguono per la suddivisione dei suoi elementi, anche se condividono la memoria, in un *PowerPC Processor Element* (*PPE*) e otto *Synergistic Processor Element* (*SPE*) in base alle loro funzioni.
La *PPE* ha un core a 64 bit basato su architettura *PowerPC* sulla quale girano sistemi operativi e applicazioni a 32/64 bit.
La *SPE* è ottimizzata per applicazioni *SIMD compute-intensive*, è indipendente dal processore ed ognuna di esse esegue programmi o thread con pieno accesso alla memoria condivisa e alla parte memory-mapped I/O.
Le *SPE* dipendono dalla *PPE* che esegue il sistema operativo e spesso il controllo dei thread di alto livello; per prestazioni migliori la *PPE* dipende dalle *SPE*.

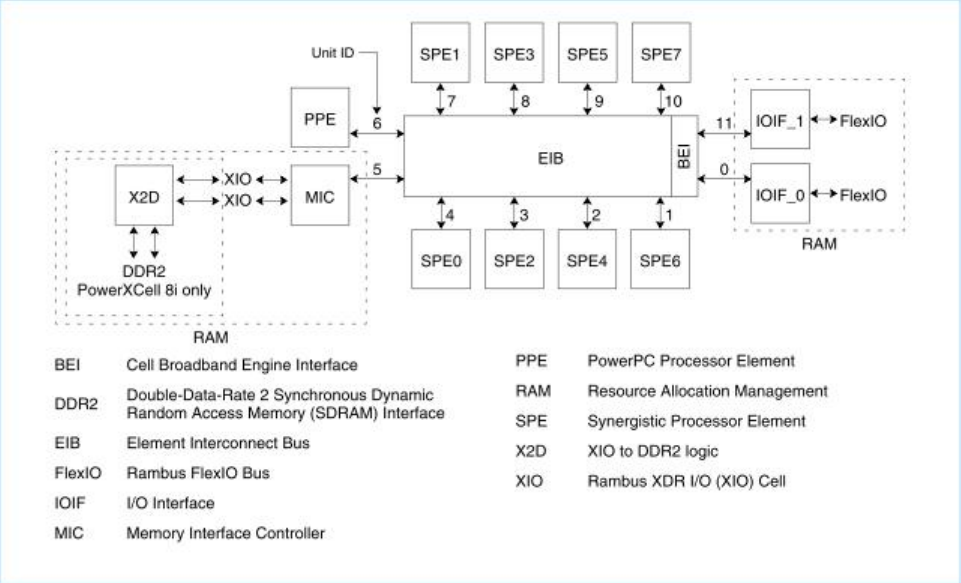

Figura 2.1: Overview dell'hardware dei processori *CBEA*

Le *SPE* sono disegnate per essere programmate in linguaggi di alto livello (*C*/*C++*) e supportano istruzioni *SIMD*; la *PPE* supporta anche istruzioni *standard PowerPC* ed estensioni *vector/SIMD*.
La *PPE* è più adatta ad attività control-intensive e un rapido task switching rispetto alle *SPE*, orientate a task compute-intensive. Nonostante gli elementi siano capaci di compiere entrambe le funzionalità, variano le prestazioni dall'area del chip all'efficienza del consumo di potenza.
Differenze sostanziali tra *PPE* e *SPE* sono le modalità di accesso alla memoria: solo la *PPE* accede direttamente alla memoria principale con load/store che spostano dati ad un registro privato, il cui contenuto può essere "cachato". Le *SPE* accedono direttamente solo alla memoria locale, per accedere alla memoria di sistema usano *tecniche DMA* spostando dati da e verso la propria regione di memoria, senza cache, detta *Local Storage* (*LS*). Le istruzioni di fetch, load e store di una *SPE* accedono solo alla *LS* privata.
Questa organizzazione a tre livelli (registri, *LS*, memoria principale), con trasferimenti DMA asincroni fra *LS* e memoria principale, è una rottura dalla classica architettura e dai modelli di programmazione parallelizzando con trasferimenti di dati e istruzioni che memorizzano i risultati nella memoria principale. Una delle cause di ciò è la latenza, in cicli di processore, che si è centuplicata dal 1980 al 2000 al punto da dare limitate prestazioni.

Quando un programma sequenziale compie una load non presente in cache (*cache miss*) l'esecuzione si ferma per qualche centinaia di cicli generando latenza. Rispetto a ciò i pochi cicli di una *SPE* che servono per un trasferimento DMA hanno un miglior trade-off, il *controllore DMA* delle *SPE* compie fino a 16 trasferimenti DMA simultaneamente. I tipici processori gestiscono solo pochi accessi indipendenti alla memoria.
Uno dei metodi di trasferimento DMA ha una lista dei trasferimenti nella memoria locale in modo che il controller DMA la processi asincronamente quando la *SPE* computa i precedenti dati trasferiti.

Potenza, Memoria e Frequenza

I *CBEA* superano tre limitazioni dei microprocessori attuali: prestazioni-consumo di potenza, utilizzo di memoria e frequenza di clock.
Le performance dei microprocessori stanno raggiungendo i limiti di dissipazione di potenza rispetto ai componenti elettronici dei circuiti integrati e l'unico modo per aumentare le prestazioni è aumentare la power efficiency a circa lo stesso ratio al quale le performance aumentano.
I processori *CBEA* fanno ciò differenziandosi fra *PPE*, ottimizzate per sistemi operativi e codice control-intensive, e le *SPE*, ottimizzate per applicazioni compute-intensive.
Negli attuali multiprocessori simmetrici e in quelli con controllori di memoria integrati, la latenza della memoria *DRAM* sta raggiungendo i 1000 cicli e lo spostamento di dati fra memoria principale e processore limita le prestazioni. I progettisti di compilatori e applicazioni dovranno gestire lo scambio di dati visto che i meccanismi hardware di caching sono gestiti da questa attività.
Nel *CBEA* le tecniche di gestione della latenza permettono di schedulare dati simultanei e trasferimenti di codice in modo da coprire tempi di latenza lunghi. A 16 trasferimenti simultanei per *SPE*, i *CBEA* supportano fino a 128 trasferimenti simultanei fra memoria locale e principale, superando la banda dei tipici processori di un fattore di scala pari a 20.
I processori attuali richiedono pipeline sempre più profonde in modo da raggiungere frequenze sempre più alte, anche se questa tecnica ha raggiunto il suo culmine.
Specializzando la *PPE* in attività control-intensive e le *SPE* in task compute-intensive, questi elementi girano ad alte frequenze senza eccessivo overhead.
La *PPE* è efficiente se esegue due thread simultaneamente piuttosto che ot-

timizzare un singolo thread, mentre ogni *SPE* è efficiente se utilizza registri che supportano istruzioni in parallelo "al volo" senza overhead da renaming di registri o computazione disordinata e trasferimenti asincroni DMA, che supportano operazioni concorrenti in memoria senza overhead.
Distinguendo e ottimizzando il controllo dai dati, i *CBEA* alleggeriscono i problemi imposti dalle limitazioni strutturali di potenza, memoria e frequenza. Il risultato è un multiprocessore che, con pari potenza, fornisce fino a 10 volte le performance di picco dei comuni processori. Perciò applicazioni compute-intensive con dati da 32 bit o meno (interi o a virgola mobile con singola precisione) sono perfetti candidati per i *CBEA*.

2.1.2 Ambiente hardware

Gli elementi del processore

La figura 2.1 mostra un diagramma a blocchi ad alto livello dell'hardware del *CBEA*: tutti gli elementi sono connessi fra loro, alla memoria on-chip e ai controller I/O da un unico bus di interconnessione di elementi (*EIB*) memory-coherent ad alta velocità ed in grado di trasferire fino a 25.6 GB/s.
La *PPE* ha un core *PowerPC Architecture RISC* dual-thread a 64 bit e supporta un sottosistema PowerPC con memoria virtuale, ha istruzioni di livello 1 (*L1*) a 32 KB, una cache per i dati ed una cache di livello 2 (*L2*) a 512 KB per istruzioni e dati.
La *PPE* è concepita per il controllo del processing, per far girare sistemi operativi e per gestire risorse e *thread SPE*. Può eseguire software PowerPC Architecture ed è adatta a eseguire codice system-control.
Le *SPE* sono otto identici elementi *SIMD* (single-instruction multiple-data) ottimizzati per operazioni data-intensive, a loro allocati dalla *PPE*, ed ogni *SPE* contiene un core RISC, local storage da 256 KB controllato dal software per istruzioni e dati ed un registro a 128 bit. Le *SPE* hanno un instruction set speciale *SIMD* (Synergistic Processor Unit Instruction Set Architecture) e un set unico di comandi per gestire i trasferimenti DMA, lo scambio di messaggi e il controllo. I *trasferimenti DMA* accedono alla memoria principale grazie agli indirizzi effettivi PowerPC: come nella *PPE*, lo scambio di indirizzi delle *SPE* è gestito dal segmento PowerPC Architecture e dalle tabelle di paginazione caricate nelle *SPE* tramite software privilegiato eseguito sulla *PPE*. Le *SPE* non sono state concepite per eseguire sistemi operativi.

Una *SPE* controlla i trasferimenti DMA e comunica per mezzo dei canali implementati e gestiti dal suo *memory flow controller* (*MFC*). La *PPE* e altri device nel sistema, incluse altre *SPE*, possono accedere allo stato del *MFC* attraverso i registri memory-mapped I/O (*MMIO*) dell'*MFC* e code, visibili al software nell'address space della memoria principale.

Element Interconnect Bus (EIB)

Il bus di interconnessione degli elementi (*EIB*) è il mezzo di communicazione fra tutti gli elementi, i controller on-chip per la memoria e l'I/O. Supporta operazioni *full memory-coherent* e *symmetric multiprocessor* (*SMP*), quindi i *CBEA* sono disegnati per essere integrati coerentemente con altri *CBEA* producendo un *cluster*.
L'*EIB* è costituito da quattro data rings di ampiezza pari a 16 byte; ogni anello trasferisce 128 bytes alla volta (una linea di cache della *PPE*). Ogni elemento del processore ha un on-ramp e un off-ramp e può inviare e ricevere dati in parallelo.
La figura 2.1 mostra i numeri di ID delle unità di ogni elemento e l'ordine in cui gli elementi sono connessi al bus *EIB*. L'ordine di connessione è importante per minimizzare la latenza dei trasferimenti sull'*EIB* che è funzione del numero di hop di connessione: trasferimenti fra elementi adiacenti hanno latenze minori, rispetto a trasferimenti fra elementi separati da più hop.
La massima banda interna è di 96 byte per ciclo di clock del processore. L'*EIB* non ha nessun comportamento di *quality of service* (*QoS*) a parte la garanzia di un progresso continuo e una funzione di *resource allocation management* (*RAM*). Il software privilegiato può usare quest'ultima per regolare il ratio con il quale i richiedenti utilizzano la memoria e le risorse I/O.

Memory Interface Controller (MIC)

Il controllore di interfaccia di memoria on-chip (*MIC*) fornisce l'interfaccia fra l'*EIB* e la memoria fisica, supportando una o due interfacce di memoria Rambus extreme data rate (*XDR*); queste insieme supportano fra i 64 MB e i 64 GB di memoria *XDR DRAM*. Il processore *PowerXCell 8i* supporta uno o due canali di memoria a 128 bit *DDR2*.

Il *MIC* ha più modalità controllate a livello software: ottimizza la latenza se le code sono vuote (*fast-path*), innalza la priorità delle letture *SPE*, inizia una lettura prima che la precedente scrittura sia terminata (*early read*), ha modalità per *power management* (speculative read e slow mode).
La memoria *XDR DRAM* è protetta da un codice di correzione degli errori (*ECC*) ed ha 4, 8 o 16 banchi di memoria.

Cell Broadband Engine Interface Unit (BEI)

L'unità di interfacciamento on-chip Cell Broadband Engine (*BEI*) supporta l'interfacciamento I/O e include un controllore di interfaccia a bus (*BIC*), un I/O controller (*IOC*) ed un controller d'interrupt interno (*IIC*). Gestisce i trasferimenti di dati fra l'*EIB* e i device I/O.
La *BEI* ha due interfacce Rambus FlexIO una delle quali (*IOIF1*) supporta solo un protocollo I/O noncoherent (*IOIF*) adatto per device I/O. L'altra (*IOIF0* o *BIF/IOIF0*) ha un protocollo selezionabile a livello software fra noncoherent e coherent attraverso la quale può essere utilizzato per estendere l'*EIB* ad un altro device memory-coherent come un altro *CBEA*.

2.1.3 Ambiente di programmazione

Un vasto set di estensioni che definiscono i tipi di dati per operazioni *SIMD* e mappano le *intrinsic* (comandi in forma di chiamate di funzioni) a una o più istruzioni assembly.

Instruction Set

Il set di istruzioni per *PPE* è una versione estesa di quello dell'architettura PowerPC con pochi cambiamenti e aggiunte multimediali vector/SIMD.
L'instruction set per *SPE*, simile all'estensione vector/SIMD per *PPE* anche se eseguiti da diversi compilatori, è il *SIMD Synergistic Processor Unit Instruction Set Architecture*, with accompanying C/C++ intrinsics e un set unico di comandi per gestire trasferimenti DMA, eventi esterni, messaggi fra processori e altre funzioni.

Storage domains e interfacce

I processori *CBEA* hanno due tipi di memoria, una principale e otto locali in ogni *SPE*, che ha un'interfaccia di canale per permettere la comunicazione

fra la sua *SPU* e il suo *MFC*.
La memoria principale, il cui spazio è interamente effective-address, può essere configurata dal software privilegiato PPE per essere condivisa da tutti gli elementi del processori e dai device memory-mapped nel sistema.
Si può accedere ad uno stato *MFC* dalla *SPU* ad esso associata attraverso un'interfaccia di canale oppure dalla *PPE* o da altri device ad esso associati (*SPE* comprese) per mezzo dei registri *MMIO* del *MFC* contenuti nello spazio della memoria principale. Si può accedere anche ad un *LS* dalla *PPE* o da altri device attraverso lo spazio della memoria principale, al quale la *PPE* accede attraverso il suo sottosistema di memoria PowerPC (PowerPC processor storage subsystem o *PPSS*).
La figura 2.2 mostra un diagramma dei tipi di storage e le relative interfacce dell'hardware del processore *CBEA*.

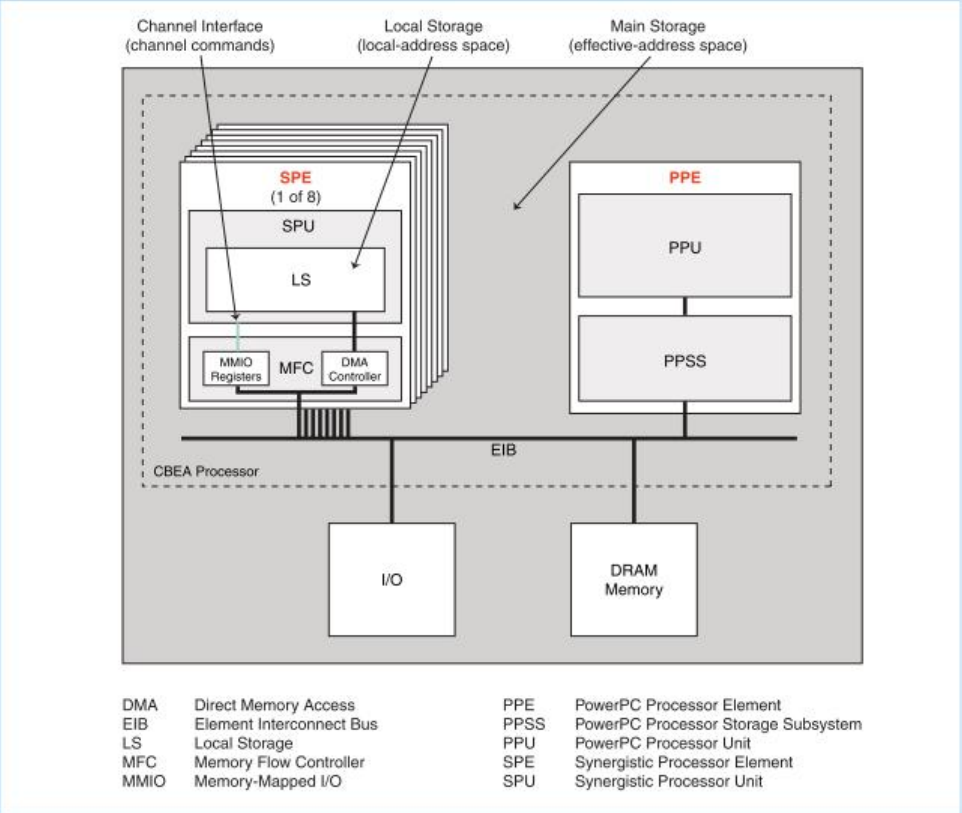

Figura 2.2: Tipi di storage e relative interfacce

Una SPU di una *SPE* può fare il fetch di istruzioni provenienti solo dalla propria *LS* e le istruzioni di load/store eseguite possono accedere solo alla *LS*. Ogni *MFC* delle *SPE* comprende un controllore DMA, le cui richieste contengono sia indirizzi della *LS* che indirizzi effettivi facilitando trasferimenti fra diversi domini di competenza delle memorie.
I trasferimenti di dati fra *LS* e memoria principale sono compiuti dalla *SPE* associata, dalla *PPE* o da un'altra *SPE*, utilizzando il controllore DMA nel *MFC* associato con la *LS*. Il software *SPE* interagisce con il *MFC* attraverso l'interfaccia di canale.
I canali supportano code di comandi di DMA, mailbox e messaggi di segnalazione o notifica. Il software eseguito su *PPE* o *SPE* può interagire con un *MFC* attraverso i registri *MMIO* visibili nello spazio della memoria principale.
Ogni *MFC* gestisce e processa due code indipendenti per i DMA e per altri comandi: una per la sua SPU e un'altra per altri device che accedono alla *SPE* attraverso lo spazio della memoria principale. Ogni *MFC* può processare più comandi DMA e può gestire autonomamente una sequenza di trasferimenti DMA in risposta a una lista di comandi DMA dalla sua *SPE*, ma non da *PPE* o altre *SPE*.
Ogni comando DMA è identificato con un ID appartenente a un gruppo di tag che permette al software di controllare o attendere il completamento dei comandi in un gruppo di tag.
Le performance di trasferimento di picco sono raggiunte se sia gli indirizzi effettivi che quelli delle *LS* sono allineati a 128 byte e la dimensione del trasferimento è multiplo di 128 byte.
Ogni *MFC* ha un'unità gestore di memoria synergistic (*SMM*) che processa informazioni di mapping degli indirizzi e permessi di accesso fornite dal sistema operativo sulla *PPE*. Per processare un indirizzo effettivo fornito da un comando DMA, il *SMM* usa fondamentalmente il medesimo meccanismo di mapping degli indirizzi e di protezione usato dall'unità dedita alla gestione della memoria (*MMU*) nel sottosistema di memorizzazione (*PPSS*) della *PPE*.

Ordinamento di byte e numerazione di bit

Lo storage dei dati e delle istruzioni nei processori *CBEA* usa l'ordinamento "Big Endian" con le seguenti caratteristiche:

- il bit più significativo è memorizzato all'indirizzo minore e il bit meno significativo è memorizzato all'indirizzo maggiore;
- la numerazione dei bit in un byte va dal bit più significativo (bit 0) al meno significativo (bit n) e differisce da altri processori "Big Endian".

La figura 2.3 è un sommario dell'ordinamento dei byte e dei bit e le convenzioni "Big Endian" dei bit dell'hardware del *CBEA*.

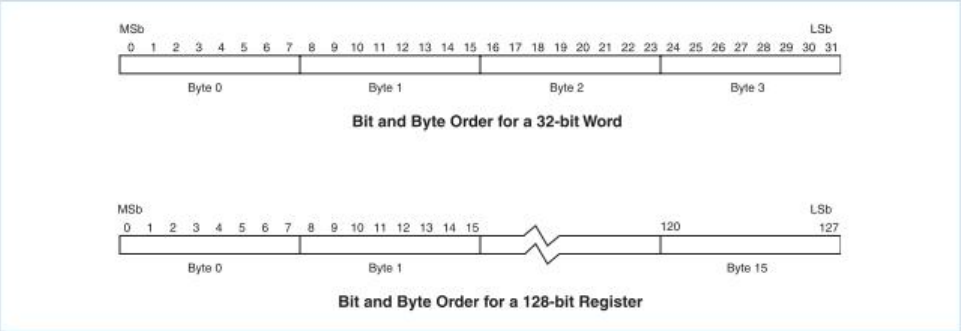

Figura 2.3: Byte "Big Endian" e ordinamento di Bit

L'ordinamente "Little Endian" non è supportato né dalla *PPE*, né dalle *SPE* inclusi i loro *MFC*. I trasferimenti DMA del *MFC* sono semplici spostamenti di byte, quindi l'orientamento è irrilevante per spostamenti di blocchi di dati. Il mapping dei byte ordinati diventa importante solo quando i dati sono caricati o interpretati da un elemento del processore o da un *MFC*.

Ambiente Runtime

La *PPE* esegue applicazioni PowerPC e sistemi operativi che includono sia istruzioni PowerPC che istruzioni derivate da estensioni vector/SIMD; per utilizzare tutte le feature del *CBEA*, la *PPE* necessita un sistema operativo in grado di supportarle, come il multiprocessing con le *SPE*, l'accesso a operazioni *PPE* vector/SIMD, il controllore di interrupt e tutte le altre funzioni.

E' cosa comune un programma principale su *PPE* che alloca thread alle *SPE*: il main thread fa lo spawning di una o più task. Una task ha uno o più main thread con i quali è associata per mezzo di alcuni *SPE thread*. Un *SPE thread* è un thread di cui è stato fatto lo spawning per essere eseguito su una SPE disponibile.
In figura 2.4 vi è un sommario delle definizioni di thread e task.

Term	Definition
Main Thread	A thread running on the PPE.
Task	A task running on the PPE and SPE. Each such task: • Has one or more main threads and some number of SPE threads. • All the main threads within the task share the task's resources, including access to the SPE threads.
SPE Thread	A thread running on an SPE. Each such thread: • Has its own 128 × 128-bit register file, program counter and MFC-DMA command queues. • Can communicate with other execution units (or with main storage through the MFC channel interface).

Figura 2.4: Definizione di Thread e Task

Un *main thread* può interagire direttamente con un *thread SPE* attraverso la *LS* della *SPE* ed indirettamente tramite lo spazio in memoria principale. Un thread può fare polling, mettersi in sleep o essere in attesa di *SPE thread*.
Il sistema operativo definisce il meccanismo e la policy di selezione fra le *SPE* disponibili, dando priorità a tutte le applicazioni nel sistema, e schedula l'esecuzione delle *SPE* indipendentemente dai main thread. Il sistema operativo è anche responsabile del runtime loading, del passaggio di parametri ai programmi SPE, delle notifiche di eventi ed errori SPE e del supporto al debugging.

2.2 PowerPC Processor Element (PPE)

L'unità *PowerPC Processor Element* (*PPE*) è un processore general-purpose, dual-threaded e RISC a 64 bit che lo conforma all'architettura PowerPC 2.02 con estensioni SIMD Altivect per le operazioni di tipo vettoriale.
La *PPE* è responsabile di tutto il controllo del sistema, esegue i sistemi operativi per tutte le applicazioni eseguite su di essa e sulle unità SPE. La *PPE* consiste in due unità principali: la *PowerPC processor unit* (*PPU*) e il *PowerPC processor storage subsystem* (*PPSS*), mostrati in figura 2.5.

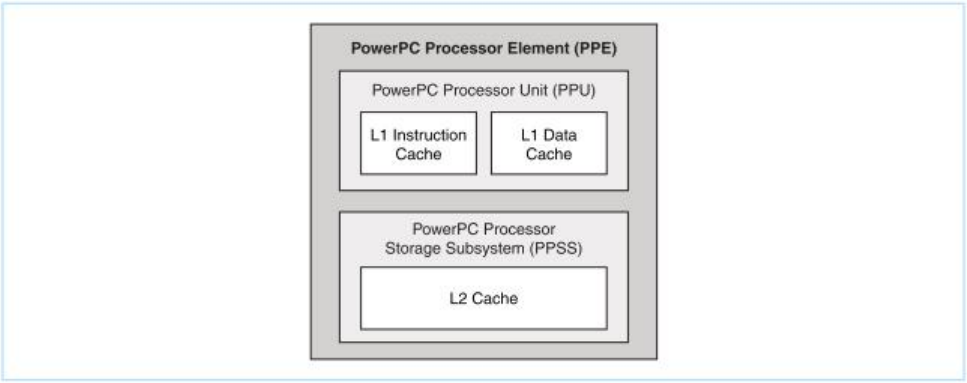

Figura 2.5: Diagramma a blocchi del *PowerPC Processor Element*

La *PPU* ha una cache per le istruzioni di livello 1 (*L1*), una cache per i dati e cinque unità funzionali: l'instruction unit (*IU*), la load and store unit (*LSU*), la vector scalar unit (*VSU*), la fixed-point unit (*FXU*) e la memory management unit (*MMU*). Può caricare 32 byte e memorizzarne 16 per ciclo di processore in modo indipendente e memory-coherent.
Il *PPSS* ha una cache unica di livello 2 (*L2*) per istruzioni e dati e gestisce le richieste di allocazione di memoria interne ed esterne verso la *PPE* dalle *SPE* o dai dispositivi di input/output.
La *PPU*, il *PPSS* e le loro unità funzionali sono mostrate in figura 2.6.

2.2.1 PowerPC Processor Unit (PPU)

La *PPU* esegue l'instruction set dell'architettura PowerPC e le istruzioni estese vector/SIMD. Ha set duplicati dei file di registri PowerPC e vector user-state (un set per ogni thread) più un set delle seguenti unità funzionali:

- Instruction Unit (*IU*): esegue fetch, decode, dispatch, issue, branch delle istruzioni e il loro completamento di esecuzione;
- Load and Store Unit (*LSU*): si occupa di tutta gli accessi ai dati, inclusa l'esecuzione di istruzioni load and store;
- Vector/Scalar Unit (*VSU*): include una floating-point unit (*FPU*) e una vector/SIMD multimedia extension unit (*VXU*) a 128 bit, che

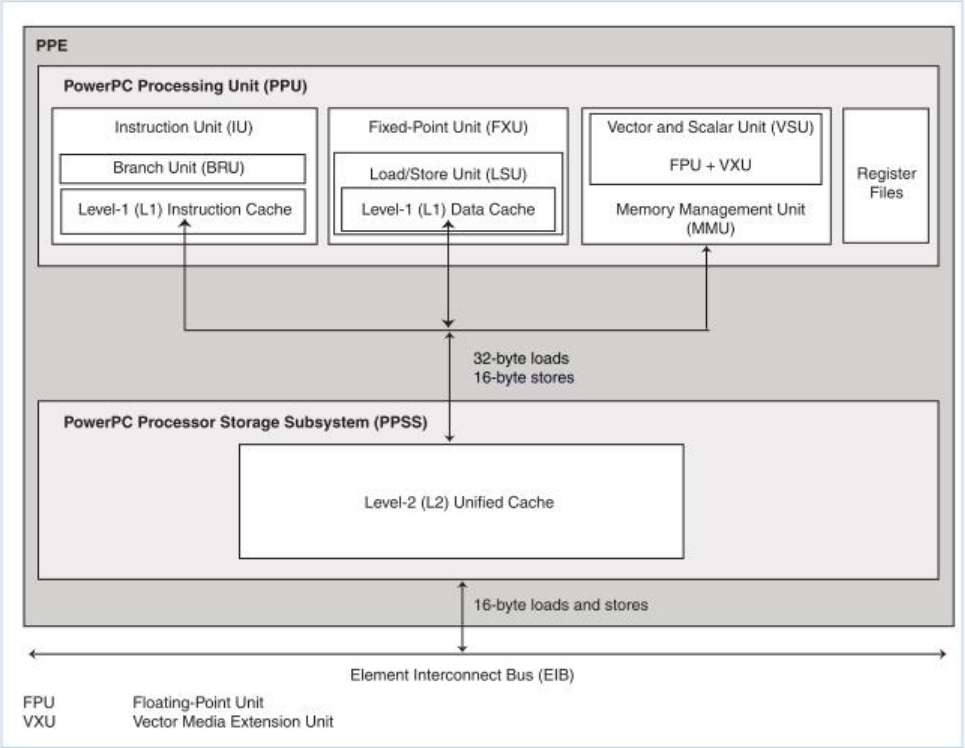

Figura 2.6: Unità funzionali del *PowerPC Processor Element*

entrambe eseguono istruzioni floating-point e vector/SIMD multimedia extension;

- Fixed-Point Unit (*FXU*): esegue operazioni fixed-point (integer) comprese somme, moltiplicazioni, divisioni, compare, shift, rotate e operazioni logiche;
- Memory Management Unit (*MMU*): gestisce scambi di indirizzi per tutti gli accessi alla memoria;

Tutte le istruzioni vector/SIMD sono create in modo che con esse si possa fare facilmente una pipeline. L'esecuzione in parallelo con le istruzioni fixed-point e floating-point è semplificata dal fatto che le vector/SIMD non generano eccezioni (eccetto interrupt data-storage nelle load e store), non suporta funzioni complesse e condivide poche risorse o path di comunicazione con altre unità di esecuzione *PPE*.

2.2.2 PowerPC Processor Storage Subsystem (PPSS)

Il PowerPC Processor Storage Subsystem (*PPSS*) gestisce tutti gli accessi alla memoria attraverso la *PPU* e operazioni memory-coherence (snooping) dall'element interconnect bus (*EIB*). Ha una cache unificata a 512 KB, set-associativa a 8 vie, write-back e con codice di correzione degli errori (*ECC*). La cache ha un'interfaccia read/write single-port alla memoria principale che supporta otto stream di dati prefetchati gestito a livello software. Include i contenuti della cache L1 dei dati ma non è detto contenga il contenuto della cache L1 per le istruzioni e fornisce un supporto fully coherent symmetric multiprocessor (*SMP*).
Il *PPSS* esegue il prefetch dei dati per la *PPU* e fa bus arbitration e fa pacing (dà il passo) nell'*EIB*. Il traffico fra la *PPU* e il *PPSS* è supportato da un port load e da un port store.

2.2.3 Registri PPE

I *registri PPE* problem-state (user) sono mostrati in figura 2.7. Tutte le istruzioni computazionali compiono operazioni sui registri e nessuna di esse modifica la memoria principale. Per utilizzare un operando che faccia uso della memoria e modifichi una posizione in memoria, il contenuto dell'operando deve essere caricato in un registro, modificato e poi memorizzato nella posizione target.

2.2.4 Istruzioni estese vector/SIMD

Le istruzioni vector/SIMD possono essere mischiate con istruzioni PowerPC: la vector/SIMD multimedia extension unit (*VXU*) opera concorrentemente con la fixed-point unit (*FXU*) a 32 o 64 bit e la floating-point unit (*FPU*) a 64 bit della *PPU*.

Vettorizzazione o SIMDizzazione

Un vector, o operando single-instruction multiple-data (SIMD), è un operando di un'istruzione che contiene un set di dati raggruppati in un array monodimensionale e i suoi elementi possono essere valori a virgola fissa o mobile.

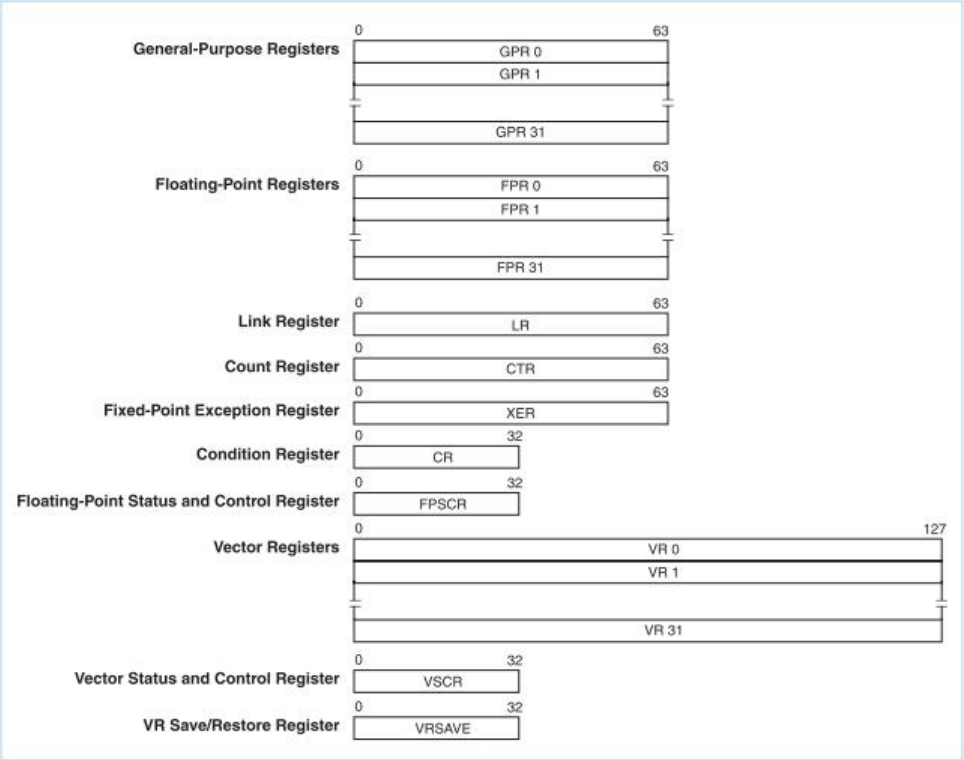

Figura 2.7: *PPE* User Register Set

L'utilizzo di SIMD fa esplodere il parallelismo a livello di dati, ovvero le operazioni richieste per trasformare un set di elementi di un vector possono essere fatte su tutti gli elementi del vector allo stesso tempo: una singola istruzione è applicata a più elementi in parallelo.
Il supporto per le operazioni SIMD è pervasivo nei processori *CBEA*: nella *PPE* e nelle *SPE* i registri vector contengono elementi di dati multipli come un singolo vector. La figura 2.8 mostra una somma.

La figura 2.9 mostra un altro esempio di operazione SIMD: un rimescolamento di byte. I byte selezionati per essere mischiati dai registri sorgenti VA e VB sono basati sugli ingressi dei byte nel vector controller VC nel quale uno 0 indica la provenienza VA e un 1 indica VB; il risultato è memorizzato nel registro VT.
L'inizializzazione per l'utilizzo su un vector processor è chiamata vettorizzazione o SIMDizzazione ed è fatta dal programmatore o da un compilatore auto-vettorizzante.

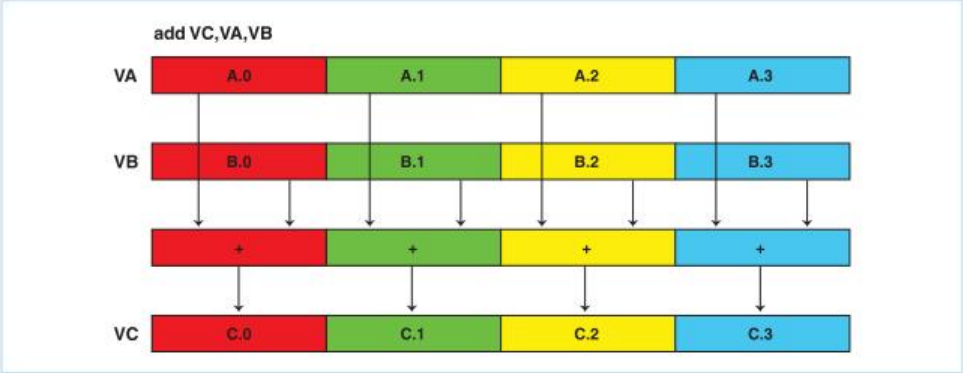

Figura 2.8: Quattro operazioni di somma fra loro concorrenti

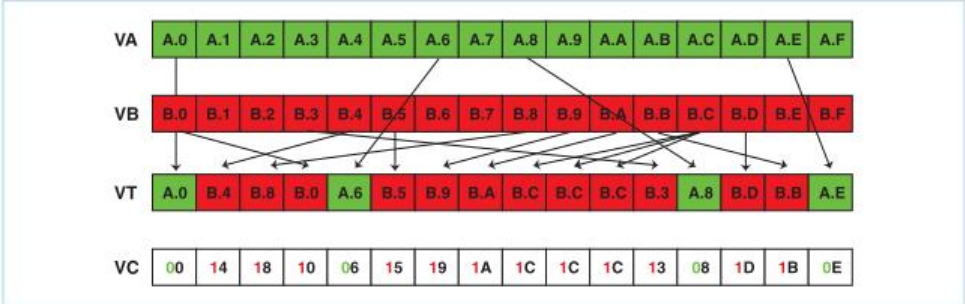

Figura 2.9: Byte-Shuffle (Permute) Operation

2.3 Synergistic Processor Elements (SPE)

Le otto *Synergistic Processor Element* (*SPE*) eseguono un nuovo instruction set SIMD detto Synergistic Processor Unit (*SPU ISA*). Ogni *SPE* è un processore RISC a 128 bit specializzato per applicazioni scalari, ricche di dati e compute-intensive SIMD e consiste in due unità principali: la synergistic processor unit (*SPU*) e il memory flow controller (*MFC*) come mostrato in figura 2.10.

Le *SPE* forniscono un ambiente operativo deterministico: non hanno cache e non ci sono cache miss che riducano le prestazioni. Le regole di scheduling con pipeline sono semplici: è facile determinare staticamente le performance del codice e generare schedule statici e di alta qualità.

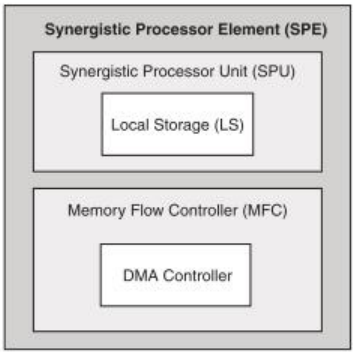

Figura 2.10: Synergistic Processor Element Block Diagram

2.3.1 Synergistic Processor Unit (SPU)

La *SPU* fa il fetch delle istruzioni dal suo local storage (*LS*) di 256 KB usato per istruzioni e dati, carica e memorizza dati fra il proprio *LS* e il suo file di registro unico per tutti i tipi di dato. La *SPU* ha quattro unità di esecuzione, un'interfaccia di direct memory access (*DMA*) e un'interfaccia di canale per comunicare con il proprio *MFC*, con la *PPE* e altri device (incluse altre *SPE*).
Ogni *SPU* è indipendente, ha il proprio program counter, ottimizzato per eseguire *programmi SPU*. Riempie il proprio *LS* chiedendo trasferimenti DMA dal *MFC*, che li implementa grazie al proprio controller DMA. Quindi la SPU fa il fetch ed esegue le istruzioni, carica e memorizza dati verso e dal *LS*.
Le principali unità funzionali della *SPU* sono mostrate in figura 2.11 e includono la synergistic execution unit (*SXU*), il *LS* e la SPU register file unit (*SRF*).

La *SPU* può fare la issue e completare fino a due istruzioni per ciclo, una in ognuna delle due pipeline in base al suo tipo di istruzione.

Local Storage (LS)

Il *LS* è una memoria a 256 KB protetta da codice di correzione di errore (*ECC*), single-ported e senza cache. Memorizza tutte le istruzioni e i dati utilizzati dalla *SPU* e supporta un accesso per ciclo sia dal *software SPE* che dai trasferimenti DMA.

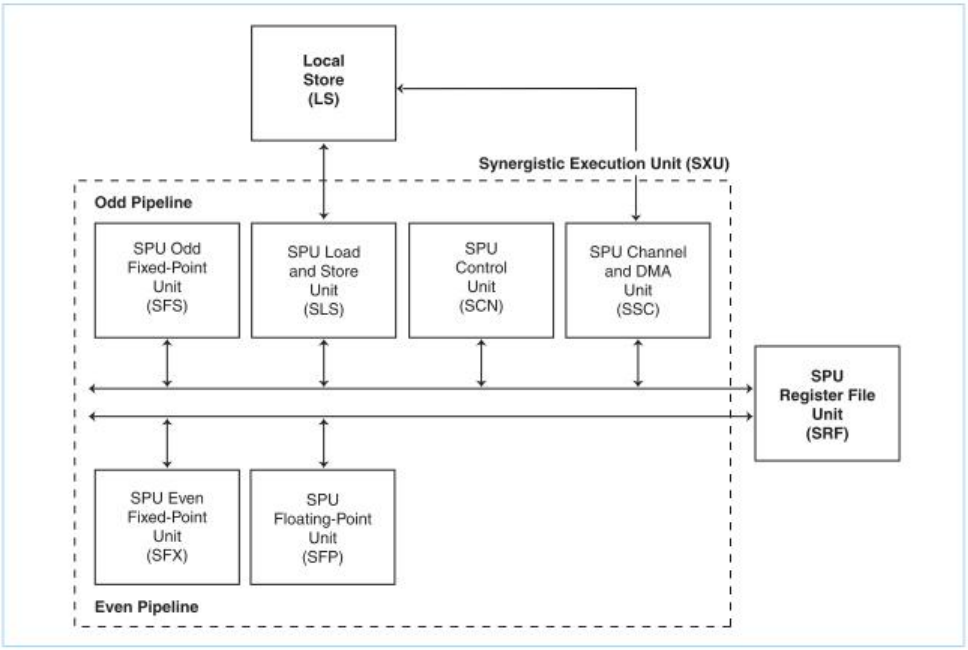

Figura 2.11: Unità funzionali della Synergistic Processor Unit

Indirizzamento e aliasing

La *SPU* accede al proprio *LS* per mezzo di istruzioni load e store, senza compiere address translation. Il software privilegiato su *PPE* può assegnare alias effective-address a un *LS* permettendo alla *PPE* e altre *SPE* di accedere al *LS* nel main-storage domain.
La *PPE* esegue questi accessi attraverso operazioni di load e store senza trasferimenti DMA, mentre le altre *SPE* devono usare trasferimenti DMA per accedere al *LS* nel *main-storage domain*. Quando l'aliasing è effettuato da software privilegiato sulla *PPE*, la *SPE* che inizializza la richiesta compie address translation, come descritto nella sezione del *MFC*.
La figura 2.12 illustra i metodi grazie ai quali una *SPU*, la *PPE*, altre *SPE* ed i device I/O accedono al *LS*, quando è stato fatto l'aliasing nel main storage domain.

2.3.2 Memory Flow Controller (MFC)

Ogni *SPU* possiede il proprio *MFC* che serve da interfaccia per la *SPU* alla memoria principale, agli altri elementi e ai dispositivi del sistema, grazie

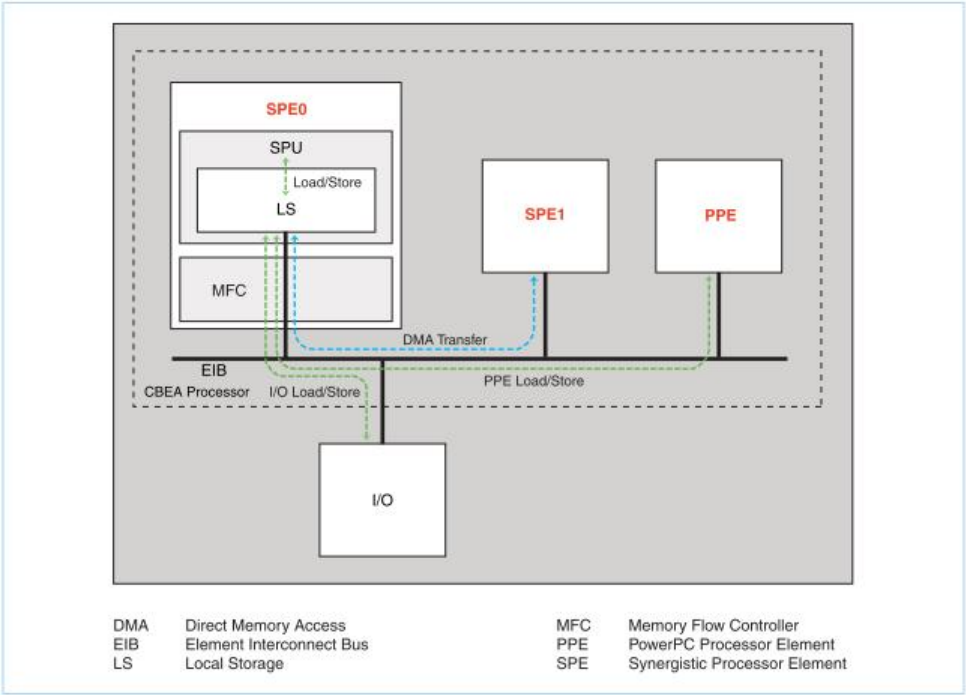

Figura 2.12: Metodi di accesso al Local Storage

all'element interconnect bus (*EIB*). La funzione principale è di interfacciare il suo *LS* con la memoria principale grazie ad un controller DMA che sposta istruzioni e dati fra il *LS* e la memoria principale. Il *MFC* supporta anche meccanismi di protezione dal lato main-storage dei suoi trasferimenti DMA, sincronizzazione fra main storage e *LS* e comunicazione con la *PPE*, altre *SPE* e device (mailbox e messaggi).

In figura 2.13 è illustrato un diagramma a blocchi del Memory Flow Controller e indica le sue funzioni principali di interfacciamento alla *SPU* e all'*EIB*:

- LS Read and Write Interface: collega il *LS* con l'*EIB*; è usato per trasferimenti DMA, atomici e richieste di snoop;
- LS DMA List Element Read Interface: collega il *LS* con il canale del *MFC* e la coda dei comandi della *SPU*;
- Channel Interface: collega il canale *SPU* della *SXU* e l'unità DMA (*SSC*) con il canale del *MFC* e la coda dei comandi della *SPU*;
- EIB Command and Data Interfaces: collega il *MFC* con il bus *EIB*.

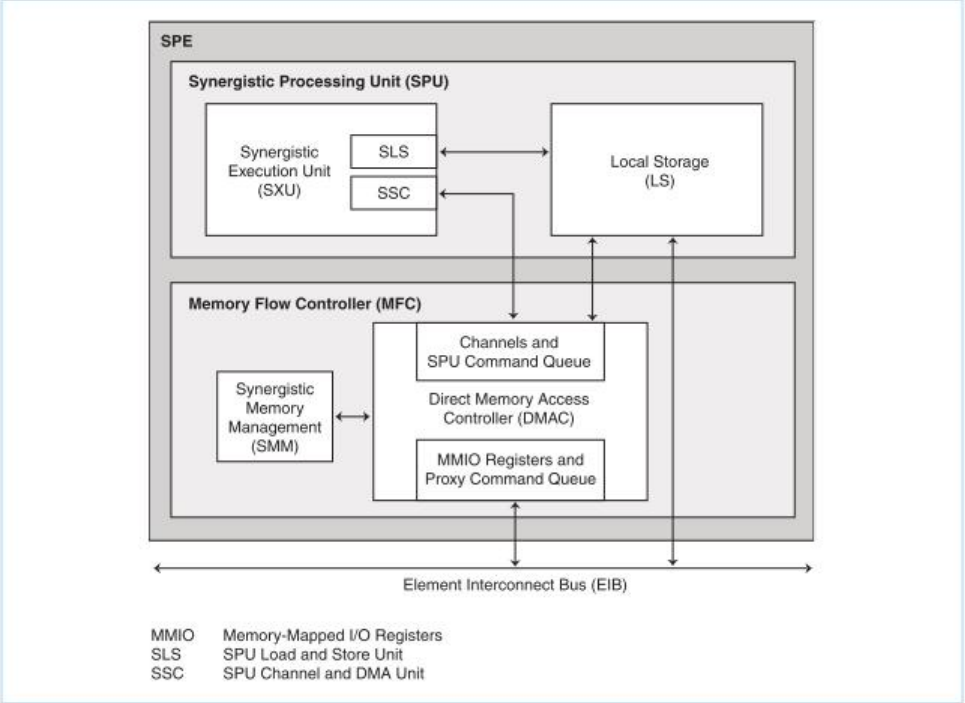

Figura 2.13: Diagramma a blocchi del Memory Flow Controller

Canali

Il *software SPE* communica con la memoria principale, la *PPE*, altre *SPE* e dispositivi attraverso i suoi canali, mostrati in figura 2.13 e nella 2.2. I canali sono interfacce unidirezionali che supportano messaggi a 32 bit e comandi; ogni *SPE* ha il proprio set di canali. Il software *SPE* accede ai canali con istruzioni speciali di lettura e scrittura che mette in coda i comandi *MFC*.
Il software della *PPE* e delle altre *SPE* e dei device può ottenere l'accesso a più interfacce di canale di una *SPE* accedendo ai registri associated problem-state (user) *MMIO* nello spazio della memoria principale.

Mailbox e segnalazione

L'interfaccia di canale in ogni *SPE* supporta tre mailbox: due per spedire messaggi dalla *SPE* a *PPE* ed un'altra per messaggi da *PPE* alla *SPE*. L'interfaccia di canale supporta anche due canali di segnalazione/notifica per messaggi verso la *SPE*.

La *PPE* è spesso utilizzata come controllore di applicazione che gestisce e distribuisce lavoro alle *SPE*; grossa parte di questa task riguarda il caricamento di dati da processare in memoria principale e la notifica ad una *SPE* per mezzo di mailbox o messaggi di segnalazione. La *SPE* può usare le sue mailbox in uscita per informare la *PPE* che ha portato a compimento la propria task.

Comandi MFC e commandi di code

Il software *SPE*, la *PPE*, altre *SPE* e dispositivi usano comandi *MFC* per inizializzare trasferimenti DMA, richiedere lo stato, sincronizzare il *MFC*, comunicare tramite mailbox e segnali/notifiche, ecc. I comandi *MFC* che implementano trasferimenti DMA fra *LS* e memoria principale sono chiamati comandi DMA.
Il *MFC* mantiene due code indipendenti di comandi separate mostrate in figura 2.13: una dei comandi *SPU* per la *SPU associata* e una dei comandi proxy da *PPE*, altre *SPE* e device.
Supporta l'esecuzione out-of-order dei commandi DMA che possono essere taggati con uno dei 32 ID del gruppo di tag: così il software determina lo stato dell'intero gruppo (come attendere il completamento dei comandi accodati in un gruppo). I comandi in un gruppo di tag possono essere sincronizzati con un fence o un'opzione di barrier. La *SPE* può anche usare comandi atomici di update che implementino aggiornamenti non interrompibili (una lettura seguita da una scrittura) of a 128-byte cache-line-size storage location.

Direct Memory Access Controller (DMAC)

Il controller DMA del *MFC* (*DMAC*) implementa trasferimenti DMA di istruzioni e dati fra *LS* e memoria principale. I programmi in esecuzione sulla *SPU associata*, sulla *PPE*, su un'altra *SPE* o su un dispositivo possono fare l'issue dei comandi DMA (la *PPE* può anche accedere a un *LS* attraverso il main-storage domain usando load e store).
Il *MFC* esegue i comandi DMA autonomamente, permettendo alla *SPU* di continuare l'esecuzione in parallelo con i trasferimenti DMA. Ogni *DMAC* può inizializzare fino a 16 trasferimenti DMA independenti fra loro da o verso il suo *LS*.

Synergistic Memory Management Unit (SMM)

L'unità synergistic memory management (*SMM*) del *MFC*, mostrata in figura 2.13, provvede all'address-translation e a funzioni di protezione definite dall'architettura PowerPC per gli accessi a memoria principale. La SMM fa ciò basandosi sull'informazione di address-translation fornita dal software privilegiato *PPE*.
La figura 2.12 mostra la modalità in cui la *PPE*, altre *SPE* e device di input/output accedono al *LS associato* quando su di esso è stato fatto l'aliasing in memoria principale. Il *SMM* supporta address translation per accessi DMA alla memoria principale fatti dalla *SPU associata*.
Nè il *SMM* nè la *SPU* hanno accesso diretto alle strutture preposte al controllo del sistema.

2.3.3 Modalità di isolamento della SPE

Una *SPE* implementa a livello hardware una modalità speciale di isolamento nella quale l'accesso al *LS* si comporta in modo diverso.
Quando si inizializza una transizione in un'ambiente isolato di esecuzione della *SPU*, mentre si sta operando in questo ambiente, un'area del *LS* è isolata. Solo la *SPU associata* può accedervi: l'accesso da tutti gli altri processori e dispositivi non è consentito.
La restante area del *LS* non è isolata ed è usata per trasferimenti di dati, controlli e communicazioni fra il resto del sistema e la *SPU* operante nell'ambiente di esecuzione isolato.
Come la figura 2.14 mostra, la regione isolata è l'area dall'indirizzo x'00000' al x'3E000' ed occupa 248 KB, i restanti 8 KB costituiscono l'area aperta. L'intero *LS* è accessibile dal codice eseguito sulla *SPU isolata*.
Se un comando DMA prova ad effettuare l'accesso a regioni isolate, l'accesso è spostato alla regione non isolata: l'hardware implementa ciò forzando i 5 bit più significativi a '11111', portando l'indirizzo più basso a x'3E000'.

Figura 2.14: Regioni del Local Storage

2.4 Programmare il Cell Broadband Engine

La programmazione del *Cell BE* può essere distinta in:

- programmazione della *PPE*;
- programmazione della *SPE*;
- scambio dati tra le varie unità (da *PPE* a *SPE* e da *SPE* a *SPE*).

La prima non presenta particolari difficoltà; la seconda invece presenta alcune peculiarità, legate ai vicoli architetturali di queste unità, primo fra tutti l'impossibilità di accedere direttamente alla memoria di sistema. La programmazione delle operazioni di scambio dati fra le varie unità che compongono il *Cell BE* è strettamente legata a tutti i vincoli architetturali del processore e per questo può risultare molto complessa. Inoltre le transazioni DMA sono la chiave di volta di tutto il sistema e se effettuate malamente possono causare un netto decadimento delle prestazioni di tutto il sistema.
Nel seguito saranno presentati esempi con complessità crescente scritti in linguaggio *C* mostrando alcune potenzialità delle *SPE* e delle tecniche di DMA, nelle quali il controllo delle unità avviene tramite la libreria libSPE2 che utilizza il concetto di *SPE logica*, un *contesto SPE*, di fatto una struttura al cui interno vengono memorizzati tutti i dati e le operazioni legate ad uno specifico *programma-SPE*. A questo contesto viene realmente associata una *SPE fisica* solo quando ne viene richiesta l'esecuzione.
Esistono diverse metodologie per l'utilizzo delle unità *SPE*: quella più comune prevede che la *PPE* funzioni come supervisore delle *SPE* mantenendone il controllo mentre queste processano dati, ma è anche possibile eseguire

del codice all'interno della *SPE* in modo del tutto isolato dal resto del sistema. Questa modalità viene sfruttata dall'Hypervisor PS3 che riserva una delle unità *SPE* per eseguire dei controlli di sicurezza non interrompibili né controllabili dalla *PPE*, nei nostri esempi ci concentreremo sul metodo principale.
Lo schema di base di un programma che esegue del codice all'interno di una *SPE* è il seguente:

- creazione di un *contesto SPE*;
- caricamento all'interno del *contesto SPE* di un eseguibile;
- esecuzione del contesto (richiesta l'esecuzione del codice su di una *SPE*);
- distruzione del contesto.

Purtroppo al momento permane l'impossibilità di installare Linux (o un qualsiasi altro sistema operativo) su tutte le *PS3* con firmware successivo alla versione 3.15.

Capitolo 3

L'infrastruttura di compilatori *LLVM*

L'infrastruttura di compilazione *Low Level Virtual Machine* (*LLVM*) è una collezione di tecnologie, scritta in *C++*, che facilitano e ottimizzano le fasi di compilazione, linking, esecuzione e non utilizzo di programmi scritti in qualsiasi linguaggio di programmazione, attraverso catene di tool (toolchain) modulari e riusabili. Al contrario del proprio nome, *LLVM* ha poco a che fare con le tradizionali macchine virtuali, sebbene provveda librerie che possono essere utilizzate per costruirle.
Attualmente supporta la compilazione di programmi scritti in C, C++, Objective C, Ada e Fortran, usando dei frontend derivati dal compilatore GNU Compiler Collection (GCC) 4.0.1. e 4.2. Un frontend alternativo per linguaggi simili al C è attualmente in sviluppo sotto il nome di *Clang*.
Usando *LLVM*, il programmatore può creare una macchina virtuale per linguaggi che la richiedono (come Java), un compilatore per un'architettura specifica e software di ottimizzazione del codice indipendenti dal tipo di linguaggio utilizzato o dalla piattaforma.

3.1 Overview di *LLVM*

L'LLVM project iniziò nel 2000, come progetto di ricerca all'Università dell'Illinois, con lo scopo di fornire una strategia di compilazione moderna basata su static single assignment (SSA) e capace di supportare la compilazione statica e dinamica di linguaggi di programmazione arbitrari.

Da allora LLVM è cresciuto al punto da diventare un progetto-ombrello che comprende diversi sottoprogetti, molti dei quali sono stati utilizzati in ambito commerciale open source e altri ampiamente diffusi nella ricerca accademica. Il codice nel progetto LLVM è rilasciato sotto licenza "UIUC" BSD-Style.
I principali sottoprogetti del LLVM sono:

1. le **librerie del LLVM Core**: forniscono un moderno ottimizzatore indipendente dal linguaggio sorgente e da quello di destinazione, con supporto alla generazione di codice per molte CPU comuni. Sono costruite attorno ad una rappresentazione di codice ben specificata, nota come LLVM intermediate representation ("`LLVM IR`"), e sono ben documentate: è molto facile inventare il proprio linguaggio o fare il porting di un compilatore preesistente ed usare LLVM come ottimizzatore e generatore di codice;

2. **Clang**: un compilatore "*LLVM native*" per linguaggi C, C++ e Objective C che mira a compilazioni molto veloci (3 volte più veloci del GCC quando compila codice in Objective C in fase di debugging), a fornire messaggi di errore e warning utili e a provvedere una piattaforma per costruire grandi tool a source level. Il suo *static analyzer* trova automaticamente bug nel codice ed è un esempio di tool che possono essere costruiti usando il frontend di Clang come una libreria per fare il parsing del codice C/C++;

3. **dragonegg** e **llvm-gcc 4.2**: integrano gli ottimizzatori LLVM e il generatore di codice con i parser di GCC 4.5 (GPL3) e GCC 4.2 (GPL2), permettendo a LLVM di compilare linguaggi come Ada, Fortran e altri supportati dai frontend del compilatore GCC, e fornisce compatibilità ad alta fedeltà con le rispettive versioni del GCC;

4. il **LLDB project**: costruisce librerie fornite da LLVM e Clang per provvedere un debugger nativo, utilizza gli AST di Clang, un parser di espressioni, il JIT e il disassembler di LLVM. E' molto veloce e più efficiente rispetto a GDB nel caricamento dei simboli in memoria;

5. la **libc++** e i **libc++ ABI projects**: provvedono un'implementazione standard ed alte prestazioni della libreria standard del C++, incluso pieno supporto per il C++'0x;

6. il **compiler-rt project**: fornisce implementazioni "altamente regolate" del generatore di codice a basso livello e supporta routine generate quando un target non ha una breve sequenza di istruzioni native per implementare un'operazione core IR;

7. il progetto **vmkit**: un'implementazione delle macchine virtuali Java e .NET costruita sulle tecnologie di LLVM;

8. il **klee project**: implementa una "virtual machine simbolica" che usa un prover di teoremi per valutare tutti i percorsi dinamici attraverso un programma, ricercando bug (e producendo un testcase nel caso ne trovi uno) e provando la validità delle proprietà delle funzioni.

Oltre a quelli ufficiali, c'è un'ampia gamma di progetti esterni che usano componenti dell'LLVM per varie attività come compilare linguaggi Ruby, Python, Haskell, Java, D, PHP, Pure, Lua e altri.
La forza maggiore di LLVM sta nella sua versatilità, flessibilità e riusabilità, grazie alle quali è usato per task diverse come compilazioni leggere just-in-time (JIT) di linguaggi embedded, come Lua, oppure per compilare codice in Fortran destinato a super computer massive.
LLVM inoltre possiede una vasta community interessata alla costruzione di tool a basso livello, iscritta al blog e alla mailing list LLVM Developer.

3.1.1 Descrizione

LLVM è composto da tre parti:

- la *suite LLVM*: contiene tutti i tool, le librerie e gli header file necessari per utilizzare l'LLVM, un assembler, un disassembler, un analizzatore bitcode ed un ottimizzatore bitcode. Contiene inoltre i test basilari di regressione per testare i tool e il frontend del GCC;

- il *frontend del GCC*: una versione di GCC che compila il codice C/C++ in bitcode LLVM. Attualmente usa il parser GCC per convertire codice in bitcode LLVM e, una volta compilato, può essere manipolato con i tool della suite LLVM;

- l'opzionale *test suite*: per testare ulteriormente le funzionalità e le prestazioni dell'LLVM.

LLVM quindi fornisce gli strati intermedi di un sistema di compilazione completo, prendendo codice intermediate form (*IF*) da un compilatore ed emettendo *IF* ottimizzato. Questo nuovo *IF* può essere poi convertito e linkato in codice assembly machine-dependent per una piattaforma target. LLVM può accettare un *IF* dalla toochain GCC, permettendole di essere utilizzato con un ampio array di compilatori scritti per questo progetto.
LLVM può anche generare codice macchina trasferibile a compile-time o in fase di linking oppure codice macchina binario a run-time.
La rappresentazione intermedia (*IR*) LLVM è indipendente sia dal linguaggio che dall'architettura: si interpone tra il codice sorgente in un dato linguaggio e un generatore di codice per una specifica architettura.
LLVM ha un proprio set di istruzioni indipendente dai linguaggi di programmazione la maggior parte di cui ha forma simile al Three address code. Ciascuna istruzione è nella forma static single assignment (*SSA*) quindi è strutturata da assegnare un valore ad una determinata variabile (detta registro tipizzato) una sola volta, semplificando l'analisi delle dipendenze tra variabili. Ogni cambiamento del tipo di una variabile o di un oggetto è effettuata attraverso l'uso dell'istruzione `cast`.
LLVM supporta l'ottimizzazione del codice inter-procedurale e permette al codice di essere compilato staticamente, come con GCC, o lasciato per late-compiling da un *IF* a codice macchina in un compilatore JIT, come in Java.
LLVM fa uso dei tipi essenziali (interi) e di 5 tipi derivati (pointer, array, vettori, strutture e funzioni) che possono rappresentare costrutti più complessi appartenenti a linguaggi di alto livello, ad esempio una classe in C++ è rappresentabile da una combinazione di strutture, funzioni e array di puntatori di funzioni.
Il compilatore JIT LLVM è in grado di ottimizzare salti statici non necessari a runtime: è quindi utile per una valutazione parziale dove un programma abbia molte opzioni, la maggior parte delle quali facilmente etichettate come non necessarie in un ambiente specifico. Questa funzionalità è usata nella pipeline OpenGL del Mac OS X 10.5 Leopard per fornire alcune funzionalità non presenti a livello hardware.

3.1.2 Frontend

All'inizio LLVM doveva sostituire il generatore di codice nella stack GCC e quindi molti frontend GCC sono stati modificati, poi l'interesse crescente ha permesso di sviluppare interi frontend ex-novo per molti linguaggi di cui *Clang*, inizialmente supportato da Apple, ha ricevuto la maggior attenzione. Clang mirava a rimpiazzare il compilatore C/Objective C nel GCC con un sistema moderno e facilmente integrato con gli IDE (Integrated Development Environment) supportando ampiamente il multithreading.
Lo sviluppo di Objective C sotto GCC era stagnante e i cambi di Apple al linguaggio sono stati supportati in una sede separata, quindi la creazione del compilatore ha permesso loro di indirizzare molti problemi che LLVM indirizzava per l'integrazione con gli IDE e altre feature moderne.
Il compilatore *Utrecht Haskell* genera codice per LLVM più efficiente rispetto al generatore di codice del C, sebbene sia nelle prime fasi di sviluppo.
Il compilatore *Glasgow Haskell* (*GHC*) ha un backend per LLVM che raggiunge uno speed-up del 30% del codice compilato, rispetto al codice nativo compilato con *GHC* o generatore di codice C, che è sprovvisto di solo una delle tecniche di ottimizzazione implementate nel *GHC*.
LLVM ha tanti altri componenti in diverse fasi di sviluppo, incluso un frontend per Java bytecode, uno per Common Intermediate Language (CIL), uno per CPython, vari per Standard ML, l'implementazione MacRuby di Ruby 1.9 e un nuovo allocatore di registri per la colorazione dei grafi.

3.1.3 Funzionalità di LLVM

Il sistema di compilazione LLVM per C e C++ include:

- un frontend per C, C++, Objective C e Fortran, basato sui parser di GCC 4.2, che supporta lo standard ANSI per C/C++ come GCC;
- un'implementazione stabile dell'instruction set di LLVM, che serve come rappresentazione del codice online e offline, insieme a lettori e scrittori di assembly (ASCII) e bytecode (binary) ed un verifier;
- un sistema di gestione di pass che li mette automaticamente in sequenza (ne fa l'analisi, la trasformazione e genera codice) in base alle loro dipendenze e ne fa delle pipeline per avere maggior efficienza;
- un ampio range di ottimizzazioni "global scalar";

- un framework di ottimizzazione inter-procedurale in fase di linking con un set di analisi e transformazioni, comprese analisi di puntatori, costruzione di grafi di chiamata e ottimizzazioni "profile-guided";
- un generatore di codice che può cambiare facilmente il target e supporta X86 a 32 e 64 bit, PowerPC a 32 e 64 bit, ARM, Thumb, SPARC, Alpha, CellSPU, MIPS, MSP430, SystemZ e XCore;
- un sistema di generazione di codice Just-In-Time (JIT) che supporta X86 a 32 e 64 bit e PowerPC a 32 e 64 bit;
- supporto alla generazione di informazioni di debugging DWARF;
- un backend in C per testare e generare codice "native" su altri target;
- un sistema di profilazione simile a gprof;
- un framework di testing con codici e applicazioni di benchmark;
- API e tool di debugging per semplificare lo sviluppo dei componenti.

3.1.4 Punti di forza di LLVM

I punti di forza che LLVM possiede sono:

1. un semplice linguaggio di basso livello con semantiche rigorose;
2. un frontend per C/C++, in sviluppo per Java, Scheme e altri linguaggi;
3. un ottimizzatore "aggressivo" che fa ottimizzazioni scalari, interprocedurali, profile-driven e alcune semplici per loop;
4. un modello di compilazione a lungo termine e ottimizzazioni a link-time, install-time, run-time e offline;
5. una garbage collection accurata;
6. la facilità nel cambio di target, descritto da un potente linguaggio, da parte del generatore di codice;
7. una documentazione estensiva;
8. ospita molti progetti di vario genere;

9. lavorarci e sviluppare è facile grazie ai suoi tool;
10. è sotto sviluppo, costantemente esteso, accresciuto e migliorato;
11. è gratuito sotto licenza "three-clause BSD" approvata dall'OSI;
12. molte entità commerciali lo usano, fornendo nuove estensioni e feature.

3.1.5 Il framework LLVM Pass

Il framework *LLVM Pass* è una parte importante del sistema LLVM, perché i *passes LLVM* sono fra le parti più interessanti del compilatore che esistano. I *Passes* effettuano le trasformazioni e le ottimizzazioni che compongono il compilatore, costruiscono i risultati dell'analisi usati da queste trasformazioni e sono, soprattutto, una tecnica di strutturazione per il codice sorgente del compilatore stesso.
Tutti i *Passes LLVM* sono sottoclassi della classe *Pass* ed implementano funzionalità sovrascrivendo metodi virtuali ereditati da *Pass*. In base a come il *Pass* funziona, vi è la possibilità di ereditare dalle classi:

- `ModulePass`,
- `CallGraphSCCPass`,
- `FunctionPass`,
- `LoopPass`,
- `RegionPass`,
- `BasicBlockPass`

che danno al sistema informazioni più specifiche su quel che fa il *Pass* e come può essere combinato con altri *Passes*. Una delle funzionalità principali del framework *LLVM Pass* è che schedula i *Passes* in modo che vengano eseguiti in modo efficiente in base ai vincoli del *Pass* stesso (la sua sopraclasse).

3.1.6 Utenti di LLVM

LLVM è usato in progetti di diversa natura e gli utenti possono essere:

- ricercatori in compilatori (trasformazioni compile-time, link-time e run-time di programmi C/C++);
- sviluppatori di macchine virtuali (instruction set portabile e language-independent, un framework di compilazione);
- ricercatori in architetture (tecniche per compilare a livello hardware);
- ricercatori in sicurezza (analisi statica o strumentazione);
- sviluppatori (sistema per prototipazione veloce di trasformazioni);
- utenti finali (migliori prestazioni indipendenti dal codice).

3.2 Clang: una famiglia di frontend per LLVM

Il compilatore si pone a completamento dello sviluppo di una famiglia di frontend per LLVM scritti in C, C++, Objective C e Objective C++ con lo scopo di fornire migliori diagnostiche, migliore integrazione con gli IDE, una licenza compatibile con prodotti commerciali e un compilatore leggero, veloce e facile da sviluppare e da mantenere.
Il compilatore originale è costantemente in sviluppo, è un prodotto di qualità per le architetture ARM e X86 a 32/64 bit ed è un'ottima soluzione per la source analysis o per tool basati su trasformazione "source-to-source".
Fra le caratteristiche principali del compilatore *Clang* vi sono un basso consumo di memoria, una rapida compilazione, presenza di diagnostiche espressive e una compatibilità con il compilatore GCC.
Clang è stato rilasciato come parte delle regolari release di LLVM a partire dalla versione 2.6.

3.2.1 Funzionalità e obiettivi

Alcuni degli obiettivi del progetto includono i seguenti:

- funzionalità per gli utenti finali:
  - focus su velocità, leggerezza, scalabilità (approfondito in seguito);
  - messaggi di diagnostica espressivi.
    Clang è user friendly e genera diagnostiche utili localizzando cosa non va, evidenziando e scrivendo messaggi chiari;
  - compatibilità con GCC;
- utility e applicazioni:
  - architettura modulare basata su librerie (approfondito in seguito);
  - client diversi (refactoring, analisi statica, generazione di codice);
  - stretta integrazione con gli IDE (approfondito in seguito);
  - licenza LLVM 'BSD';
- design interno e implementazione:
  - compilatore di qualità (alte prestazioni, robusto, bug-free, diffuso)
  - codice semplice, modificabile ed estendibile;
  - un parser unico per C, Objective C, C++ e Objective C++;
  - conforme con C/C++/ObjC e con le loro varianti.

Focus su velocità, leggerezza, scalabilità

L'architettura basata su librerie compila velocemente, profila il costo di ogni strato di stack e il driver ha opzioni per l'analisi di performance. Il frontend di Clang è più veloce di GCC e usa meno memoria, ad esempio compilando "`Carbon.h`" su Mac OS/X è 2.5 volte più veloce di GCC (figura 3.1).
`Carbon.h` è un mostro: include transitivamente 558 file, 12.3 M di codice, dichiara 10.000 funzioni, 2.000 definizioni di struct, 8.000 field, 20.000 costanti enum, ecc. E' incluso in quasi tutti i file C di applicazioni GUI su Mac rendendo il suo compile time molto importante.
Possiamo misurare il tempo di preprocessing, di parsing e di costruzione degli AST per il codice (GCC non permette di distingere quest'ultimi).

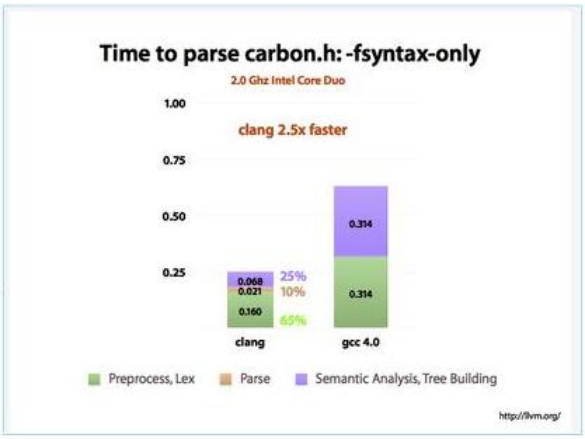

Figura 3.1: Tempo di parsing del file "Carbon.h"

Figura 3.2: Spazio di parsing del file "Carbon.h"

Il preprocessore di Clang è più veloce di GCC del 40%, mentre parsing e costruzione AST sono 4 volte più veloci di GCC. Se i sorgenti non dipendono pesantemente dal preprocessore (o si usano header precompilati) lo speedup di Clang è ancora maggiore.
Il compile time è importante ma spesso il consumo di memoria è maggiore (quando si usa Clang come API): con meno memoria il codice occupa, più codice ci sta in memoria per unità di tempo (per analisi di interi programmi). Un enorme vantaggio di Clang è che i suoi AST consumano memoria 5 volte meno degli alberi di sintassi del GCC, anche se gli AST di Clang catturano più informazione source-level rispetto agli alberi di GCC (figura 3.2). Si

può fare ciò con API disegnate appositamente e rappresentazione efficiente. Inoltre quando si confronta con GCC in modalità batch, la sua architettura a librerie lo fa adattare e permette di costruire nuovi tool, applicando un approccio "out-of-the-box" e nuove tecniche per ottimizzare la compilazione. In figura 3.3 il preprocessore del Clang parallelizza "distcc" in modo 3 volte più scalabile del preprocessore GCC (il collo di bottiglia di "distcc" è il preprocessore quindi ne serve uno più veloce). Dalle prime due barre di ogni gruppo si nota che un preprocessore circa 40% più veloce può ridurre il preprocessing time di pesanti applicazioni C++ del 40%.
La terza barra è più interessante e mostra come un caching di accessi al file system, attraverso invocazioni del preprocessore, permette a Clang di ridurre il tempo speso nel kernel di 10 volte, rendendo "distcc" 3 volte più scalabile. Il design pulito basato su framework di Clang permette cose molto difficili su altri sistemi come la compilazione incrementale, multithreading, intelligent caching, ecc.

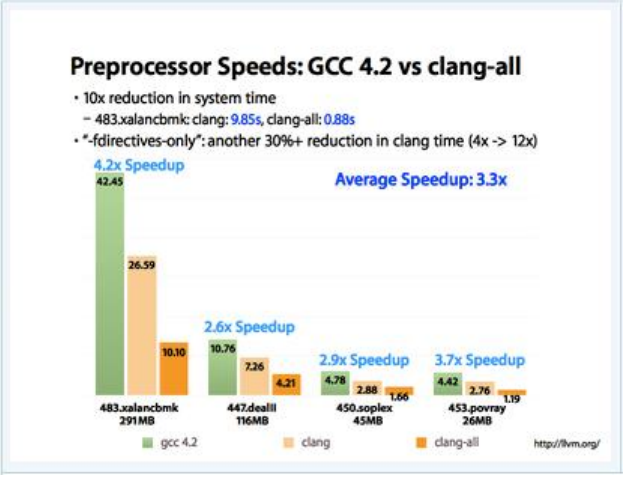

Figura 3.3: Velocità del preprocessore

Architettura modulare basata su librerie

Diverse parti del frontend possono essere divise in librerie separate e mischiate per necessità diverse. Attualmente, Clang è diviso in librerie e tool:

- *libsupport*: libreria di supporto base da LLVM;
- *libsystem*: libreria di astrazione di sistema da LLVM;

- *libbasic*: diagnostiche, SourceLocations, astrazione SourceBuffer, caching di file system per file di input;
- *libast*: classi per rappresentare gli AST C, il sistema di tipizzazione C, funzioni builtin e vari modi per analizzare e manipolare gli AST;
- *liblex*: lexing e preprocessing, hash table per identificatori, gestione di pragma, token e macro;
- *libparse*: parsing e invoca azioni a grana grossa richieste dall'utente;
- *libsema*: analisi semantica e fornisce un set di azioni del parser per costruire un AST standardizzato;
- *libcodegen*: riduce l'AST a *IR* per ottimizzare e generare codice;
- *librewrite*: modifica di buffer di testo;
- *libanalysis*: supporto all'analisi statica;
- *clang*: un programma driver, client delle librerie a vari livelli.

Se si vuole costruire un preprocessore, bastano le librerie base e del lexer.
Se si vuole un indexer, bastano le precedenti, la libreria del parser e alcune azioni per l'indexing.
Se si vuole un refactoring, un'analisi statica o un tool di compilazione "source-to-source", bastano le precedenti e le librerie di costruzione degli AST e dell'analizzatore semantico.

Permettere stretta integrazione con gli IDE

Unendo pezzi diversi, gli IDE hanno visibilità sull'intero progetto e sono a lungo termine, invece tool di compilatori stand-alone sono invocati su ogni file individuale e hanno "scope" limitato. Condividere un address space fra diversi file in un progetto permette di applicare il caching e altre tecniche per ridurre il tempo di analisi e compilazione, aumentando l'efficienza.
Un'altra differenza sono i diversi requisiti nel frontend: alte performance ed esperienza "brillante" richiedono compilazione incrementale, "fuzzy parsing", ecc.

3.2.2 Overview di Clang

Il compilatore open-source Clang è rivolto alla famiglia di linguaggi C e vuole implementare le loro classi nel modo migliore. Clang è costruito sull'ottimizzatore LLVM e sul generatore di codice che gli permettono ottimizzazione di altà qualità e generazione di codice per molti target.
Clang supporta la famiglia di linguaggi di programmazione in C che include:

- *C*: K&R C, ANSI C89, ISO C90, ISO C94, ISO C99;
- *Objective C*: ObjC 1, ObjC 2, ObjC 2.1 e alcune varianti;
- *C++*;
- *Objective C++*.

Inoltre ha funzionalità dipendenti dall'architettura o dal sistema operativo e supporta estensioni di linguaggi compatibili con GCC, Microsoft e compilatori comuni, in modo da facilitare la migrazione a Clang.

3.2.3 Linguaggio e funzionalità Target-Independent

Controllare errori e warning

Clang controlla quali costrutti di codice lo costringano ad emettere errori e warning e come siano stampati a video.
Clang controlla a grana fine la diagnostica emessa formata da:

1. indicatore di file/linea/colonna: mostra dove la diagnostica si è presentata nel codice;
2. categoria del tipo di diagnostica: nota, warning, errore o fatal;
3. stringa di testo: descrive qual è il problema;
4. un'opzione che indica se stampare il nome della diagnostica;
5. un'opzione per come controllare la diagnostica (per quelle supportate);
6. categoria di alto livello: raggruppa diagnostiche (se supportate);
7. linea di sorgente della issue: con range che indicano i posti importanti;
8. informazione "FixIt": come sistemare il problema (se Clang ne è certo);

9. rappresentazione machine-parsable dei range coinvolti (off di default).

Tutte le diagnostiche sono mappate in una delle seguenti classi:

1. ignored;
2. nota;
3. warning;
4. errore;
5. fatal error.

Le diagnostiche si possono associare ognuna con una categoria di alto livello, per dividere "*build*" con errori o warning raggruppati. Le categorie possono essere visibili: se settate su "`name`" è stampato il testo, se settate su "`id`" è stampato un numero di categoria (il mapping id-nomi categorie è ottenibile). Clang controlla le diagnostiche con opzioni e flag da linea di comando o controlla quali sono abilitate con i *pragma* nel codice sorgente (disabilitano warning specifici in una parte del sorgente). Clang supporta i *pragma* di GCC per essere compatibile con i sorgenti preesistenti ed alcune estensioni. Il pragma controlla tutti i warning settabili a ignored, warning, errore o fatal error. Il seguente esempio ignora i warning "`-Wall`".

```
#pragma GCC diagnostic ignored "-Wall"
```

Oltre alle funzionalità del pragma di GCC, Clang permette il push e il pop dello stato attuale dei warning, utile quando si scrive un file di header compilato da altre persone (difficile sapere quali flag di warning l'hanno costruito). Nell'esempio seguente "`-Wmultichar`" è ignorato per una singola riga di codice, dopo la quale le diagnostiche tornano allo stato precedente.

```
#pragma clang diagnostic push
#pragma clang diagnostic ignored "-Wmultichar"
char b = 'df'; // no warning.
#pragma clang diagnostic pop
```

I pragma push e pop salvano e ripristinano lo stato delle diagnostiche, indipendentemente da com'erano settate; è possibile usarli su diagnostiche compatibili con GCC, mentre GCC ignora push e pop. Clang supporta il pragma GCC ma, non avendo lo stesso set di warning, per pragma compatibili non è garantito lo stesso comportamento.
Sebbene non strettamente parte di Clang, le diagnostiche dell'analizzatore statico possono essere gestite dall'utente attraverso cambi al sorgente con:

- *annotazioni*: l'analizzatore statico riconosce attributi "stile GCC" che sopprimono i suoi warning o gli insegnano invarianti per trovare più bug. Molti attributi sono GCC standard, altri vanno aggiunti;
- `__clang_analyzer__`: quando l'analizzatore statico usa Clang per parsare i sorgenti, definisce implicitamente questa macro del preprocessore. Anche se sconsigliato, preferendo i bug, il codice la usa per escludere codice esaminato dall'analizzatore come nell'esempio seguente.

```
#ifndef __clang_analyzer__
// Code not to be analyzed
#endif
```

Header precompilati

Riducono il tempo di compilazione, grazie al caching del preprocessing di pesanti file di header inclusi da più sorgenti (su sistemi come Mac OS/X).

Estensioni GCC e Microsoft

Clang è compatibile con GCC il più possibile, ma alcune estensioni non sono implementate e altre non sono supportate.
Alcune estensioni Microsoft Visual C++ sono supportate in parte, poichè abilitarle fa cadere certi costrutti e statement "Microsoft-style" asm.

3.2.4 Funzionalità Target-Specific delle CPU e limitazioni

Alcune piattaforme non sono completamente supportate, sebbene aggiungere il minimo supporto per il parsing e l'analisi semantica su una nuova architettura sia facile e sufficiente a convertire in *IR*. Per contro la generazione di assembly richiede un backend LLVM adatto.

Capitolo 4

Problemi risolti nel compilatore Clang

Nel corso del nostro lavoro con *Clang* ci siamo imbattuti e scontrati con diversi problemi e mancate implementazioni che abbiamo risolto e provveduto a integrare.

1. Tra tutti spicca la mancata accettazione di parametri del `main` di diverso tipo dai classici "`argc`" e "`argv`", rispettivamente di tipo "`int`" e "`char *`", piuttosto che da un numero superiore di parametri ove necessario.
   La costruzione base dei programmi da far eseguire sull'elemento *SPE* del *Cell BE* infatti prevede che i tre parametri del main siano ciascuno di tipo "`unsigned long long`" e perciò questo problema era cruciale da risolvere inizialmente.

2. Altra difficoltà da affrontare è stato il mancato riconoscimento della parola chiave `__vector` ridefinizione della preesistente parola chiave `vector` che indica la struttura dati dei *vettori*.
   Questa ridefinizione, `__vector`, è stata introdotta per semplicità di utilizzo in alcune librerie del *Cell BE* e come tale andava risolta nel *Clang*.

3. Sempre inerente alla *SPE*, lo scambio di dati inviati dalla, e verso la, *PPE* non funzionava correttamente e il problema era dovuto principalmente al mancato funzionamento delle `__builtin` relative alla lettura e scrittura su canale (`__builtin_rdch` e `__builtin_wrch`).

Questo era un altro problema cruciale perché impediva lo sfruttamento degli elementi *SPE* che, senza poter ricevere e inviare dati alla *PPE*, potevano effettuare solo pochissime funzioni e operazioni su dati che non potevano essere scambiati con l'elemento principale.

4. Nella *SPE* si è reso necessario anche l'utilizzo di un'altra `__builtin` relativa all'estrazione di un determinato valore da un `__vector`, la funzione `spu_extract`.
   Questo problema poteva senza ombra di dubbio essere aggirato in altro modo, ma per comodità di utilizzo si è preferito provvedere alla sua risoluzione.

4.1 Parametri del `main` di tipo diverso da "`int`" e "`char *`"

Inizialmente il compilatore *Clang* dell'*LLVM* non prevedeva parametri nella funzione `main` di tipo "`unsigned long long`" (tipici delle funzioni dei programmi eseguiti sulle *SPE*) bensì i classici "`argc`" e "`argv`", rispettivamente di tipo "`int`" e "`char *`".
Per chiarificare le differenze sui tipi di parametri nelle successive figure **??** e **??** sono presentati due esempi di "`main`" che svolgono la medesima funzione ma possiedono parametri diversi.
Nella figura 4.1 è rappresentato un esempio giocattolo di codice del "`main`", con i tipici parametri "`argc`" di tipo "`int`" e "`argv`" di tipo "`char *`", il cui compito è stampare una semplice stringa "Hello World!".

```
#include <stdio.h>

int main(int argc, char * argv[])
{
  printf("Hello World!\n");
  return 0;
}
```

Figura 4.1: Classici tipi di parametri nei programmi main

Mentre in figura 4.2 è rappresentato il medesimo esempio che costituisce il listato di "`hello_spe.c`" con tre parametri di tipo "`unsigned long long`", utilizzati per contenere indirizzi di memoria a 64 bit. Il primo parametro contiene il *contesto SPE* al cui interno viene eseguito il programma mentre il secondo ed il terzo parametro, che fanno le veci dei classici puntatori agli argomenti e alle stringhe di ambiente, contengono i valori che nel codice "`hello_ppe.c`" sono stati passati alla funzione "`spe_context_run()`".

```
#include <stdio.h>

int main(unsigned long long spe, unsigned long long argp, unsigned long long envp)
{
  printf("Hello World!\n");
  return 0;
}
```

Figura 4.2: Tipi di parametri nei programmi main eseguiti sulle SPE

Si è resa quindi necessaria una modifica del codice sorgente del compilatore nella sua parte semantica per permettere l'uso di parametri di questo tipo.
La porzione di codice della funzione semantica dedita al controllo della tipologia di parametri è contenuta nella procedura "`void Sema::CheckMain()`" del file "`SemaDecl.cpp`" della libreria del *Clang*.
Trovato il punto in cui intervenire abbiamo modificato il seguente spezzone di codice preesistente.

```
QualType CharPP = Context.getPointerType(Context.getPointerType(Context.CharTy));
QualType Expected[] = { Context.IntTy, CharPP, CharPP, CharPP };
for (unsigned i = 0; i < nparams; ++i) {
  QualType AT = FTP->getArgType(i);
  bool mismatch = true;
  if (Context.hasSameUnqualifiedType(AT, Expected[i]))
    mismatch = false;
  else if (Expected[i] == CharPP) {
    // As an extension, the following forms are okay:
    //   char const **
    //   char const * const *
    //   char * const *
    QualifierCollector qs;
```

```
      const PointerType* PT;
      if ((PT = qs.strip(AT)->getAs<PointerType>())
      && (PT = qs.strip(PT->getPointeeType())->getAs<PointerType>())
      && (QualType(qs.strip(PT->getPointeeType()), 0) == Context.CharTy)) {
        qs.removeConst();
        mismatch = !qs.empty();
      }
    }
    if (mismatch) {
      Diag(FD->getLocation(), diag::err_main_arg_wrong) << i << Expected[i];
      // TODO: suggest replacing given type with expected type
      FD->setInvalidDecl(true);
    }
  }
```

Ci siamo quindi limitati a controllare che il primo parametro da verificare fosse il classico "`argc`", quindi di tipo intero, ed i seguenti fossero "`argv`", quindi di tipo "`char *`", mentre in caso negativo l'unica alternativa possibile è che tutti i parametri fossero di tipo "`unsigned long long`".
Abbiamo perciò provveduto a creare il relativo controllo dei parametri di tipo "`unsigned long long`" in base alle nostre esigenze di averne da uno a tre nel `main` dei file relativi alle funzioni da eseguire negli elementi *SPE* del *Cell BE*. Ciò è stato possibile dichiarando ed inizializzando l'array "`UllExp`" di tipo "`QualType`" con cui confrontare i tipi dei parametri del "`main`" nel seguente modo:

```
QualType UllExp[] = { Context.UnsignedLongLongTy, Context.UnsignedLongLongTy,
                      Context.UnsignedLongLongTy };
```

Si è reso inoltre necessario l'utilizzo di un altro *flag*, da noi chiamato "`ullmismatch`", in grado di verificare se i parametri *non* siano di tipo "`unsigned long long`" ed in caso positivo di produrre il messaggio di diagnostica inerente all'errato tipo dei parametri.
Queste modifiche introdotte devono però essere integrate con lo stato precedente della semantica di *Clang* in modo tale che il relativo controllo sui nuovi tipi "`unsigned long long`" dei parametri non influisca sul controllo degli abituali tipi "`int`" e "`char *`" e viceversa.

A ciò abbiamo facilmente ovviato innestando il controllo dei parametri di tipo "`unsigned long long`" nella parte di verifica dei parametri "`argc`" e "`argv`", all'interno del medesimo ciclo "`for`", come mostrato nel codice in seguito. La verifica dei relativi flag "`mismatch`" e "`ullmismatch`" e la conseguente emissione del messaggio di diagnostica, in caso uno dei due abbia valore pari a "`true`", è compiuta al termine di ogni controllo delle condizioni per la parte "`argc`" e "`argv`" e per quella "`unsigned long long`".

```
if (ullmismatch) {
  Diag(FD->getLocation(), diag::err_main_arg_wrong) << i << UllExp[i];
  FD->setInvalidDecl(true);
}
/* ...*/
if (mismatch) {
  Diag(FD->getLocation(), diag::err_main_arg_wrong) << i << Expected[i];
  FD->setInvalidDecl(true);
}
```

La nuova porzione di codice è diventata conseguentemente quella contenuta nel listato successivo.

```
QualType CharPP = Context.getPointerType(Context.getPointerType(Context.CharTy));
QualType Expected[] = { Context.IntTy, CharPP, CharPP, CharPP };

/* ADDED FROM HERE */

QualType UllPP = Context.getPointerType(Context.getPointerType
  (Context.UnsignedLongLongTy));
QualType UllExp[] = { Context.UnsignedLongLongTy, Context.UnsignedLongLongTy,
  Context.UnsignedLongLongTy };

/* TO HERE */

for (unsigned i = 0; i < nparams; ++i) {
  QualType AT = FTP->getArgType(i);
  bool mismatch = true;
  bool ullmismatch = true;
  if (Context.hasSameUnqualifiedType(AT, Expected[i])) {
    printf("Context.hSUT(AT,Expected) is TRUE, i = %d\n", i);
    mismatch = false;
```

```
    /* ADDED FROM HERE */

  } else if (Context.hasSameUnqualifiedType(AT, UllExp[i])) {
    ullmismatch = false;
  } else if (UllExp[i] == UllPP) {
    // As an extension, the following forms are okay:
    //   unsigned long long const **
    //   unsigned long long const * const *
    //   unsigned long long * const *
    QualifierCollector uqs;
    const PointerType* uPT;
    if ((uPT = uqs.strip(AT)->getAs<PointerType>())
    && (uPT = uqs.strip(uPT->getPointeeType())->getAs<PointerType>())
    && (QualType(uqs.strip(uPT->getPointeeType()), 0) == Context.UnsignedLongLongTy)) {
      uqs.removeConst();
      ullmismatch = !uqs.empty();
    }
    if (ullmismatch) {
      Diag(FD->getLocation(), diag::err_main_arg_wrong) << i << UllExp[i];
      // TODO: suggest replacing given type with expected type
      FD->setInvalidDecl(true);
    }

    /* TO HERE */

  } else if (Expected[i] == CharPP) {
    // As an extension, the following forms are okay:
    //   char const **
    //   char const * const *
    //   char * const *
    QualifierCollector qs;
    const PointerType* PT;
    if ((PT = qs.strip(AT)->getAs<PointerType>())
    && (PT = qs.strip(PT->getPointeeType())->getAs<PointerType>())
    && (QualType(qs.strip(PT->getPointeeType()), 0) == Context.CharTy)) {
      qs.removeConst();
      mismatch = !qs.empty();
    }
    if (mismatch) {
      Diag(FD->getLocation(), diag::err_main_arg_wrong) << i << Expected[i];
      // TODO: suggest replacing given type with expected type
```

```
        FD->setInvalidDecl(true);
      }
    }
  }
```

Per porre in maggior rilievo le modifiche introdotte, nel codice sono stati aggiunti dei commenti che delimitano le porzioni di listato aggiunte.

4.2 Inserimento della parola chiave `__vector`

Clang non accettava la parola chiave `__vector` ridefinizione della preesistente parola chiave `vector` che indica la struttura dati dei *vettori*, ovvero una lista di elementi dello stesso tipo simili per concezione agli *array* di dati. Quindi, ad esempio, un `__vector unsigned short` contiene all'interno di sé uno o più dati di tipo `unsigned short`.
Questa ridefinizione, `__vector`, è stata introdotta per semplicità di utilizzo nella libreria "`spu_internals.h`" del *Cell BE* che contiene anche altre definizioni di funzioni, cosiddette "target-dependent", e nelle librerie "`spu_intrinsics.h`" che contiene un'istruzione

```
#include <spu_internals.h>
```

e "`spu_mfcio.h`" ad essa collegate e tutte utilizzate prettamente dalle unità *SPU* dello stesso.
Abbiamo per ciò provato inizialmente a sostituire il preesistente `#define`

```
#define qword __vector signed char
```

con il seguente `typedef`, sempre in "`spu_internals.h`"

```
typedef signed char qword __attribute__((ext_vector_type (16)));
```

dove il valore è `16` poiché la "`qword`" (quadword, quindi parola di lunghezza pari a 4) è un vettore di tipo "`signed char`", che restituiva però un errore "*invalid conversion between ext-vector type 'qword' and 'vec_float4'*". Per ciò abbiamo modificato la stringa come segue:

```
typedef signed char qword __attribute__((vector_size (16)));
```

che ci restituiva un altro errore differente "*invalid conversion between vector type 'qword' and 'vec_float4' of different size*" a cui infine abbiamo posto soluzione andando a modificare le seguenti `#define` di "`spu_intrisics.h`":

```
#define vec_uchar16             __vector unsigned char
#define vec_char16              __vector   signed char
#define vec_ushort8             __vector unsigned short
#define vec_short8              __vector   signed short
#define vec_uint4               __vector unsigned int
#define vec_int4                __vector   signed int
#define vec_ullong2             __vector unsigned long long
#define vec_llong2              __vector   signed long long
#define vec_float4              __vector          float
#define vec_double2             __vector          double
```

con le relative versioni modificate che indicano esplicitatamente che sono vettori allineati, ovvero sostituendole con:

```
#define vector __vector
typedef unsigned char      vec_uchar16 __attribute__ ((vector_size (16)));
typedef          char      vec_char16  __attribute__ ((vector_size (16)));
typedef unsigned short     vec_ushort8 __attribute__ ((vector_size (16)));
typedef          short     vec_short8  __attribute__ ((vector_size (16)));
typedef unsigned int       vec_uint4   __attribute__ ((vector_size (16)));
typedef          int       vec_int4    __attribute__ ((vector_size (16)));
typedef unsigned long long vec_ullong2 __attribute__ ((vector_size (16)));
typedef          long long vec_llong2  __attribute__ ((vector_size (16)));
typedef          float     vec_float4  __attribute__ ((vector_size (16)));
typedef          double    vec_double2 __attribute__ ((vector_size (16)));
```

Ciò quindi ci ha permesso di risolvere il problema della ridefinizione del tipo di dati "`vector`" in "`__vector`" a livello di compilatore "*spugcc*", utilizzato per creare eseguibili per l'elemento *SPE* del *Cell BE*, ma al nostro compilatore *Clang* non andava bene.
Quindi abbiamo iniziato il processo di modifica del compilatore *Clang* cercando e prendendo come modello di riferimento le funzioni dedite a questa ridefinizione per l'*Altivec* ovvero i tipi di dati vettoriali per l'elemento *PPE* del *Cell BE*, andando a inserire funzioni e porzioni di codice nella parte semantica, di parsing e di stampa del compilatore.
Il vero problema era il mancato settaggio dell'architettura per la quale stavamo compilando, il *CellSPU*, nonostante avessimo inserito il relativo flag da linea di comando all'atto della compilazione. Abbiamo perciò verificato la presenza di un flag per il *CellSPU* nella libreria "`LangOptions.h`" per poi

concentrare la nostra attenzione sulla funzione `void CompilerInvocation::CreateFromArgs(.)` contenuto in "*CompilerInvocation.cpp*" dove siamo giunti alla decisione di aggiungere il seguente brandello di codice

```
size_t found;
found = Res.getTargetOpts().Triple.find("spu");
if (found != -1)
  Res.getLangOpts().CellSpu = 1;  // setto il linguaggio CellSpu
```

nel quale se nelle opzioni della stringa di comando passata in ingresso al compilatore è contenuto "`spu`" (quindi valido anche per il flag `-fcellspu`) allora forzo il settaggio dell'architettura a `CellSPU`.
In questo modo abbiamo posto termine al problema della ridefinizione del tipo di dati *vector* anche per *Clang* che ora lo accetta senza problemi.

4.2.1 Creazione di mini-programmi di testing su operazioni matematiche per ogni tipo di dato

Risolto il problema della parola chiave `__vector` si è conseguentemente resa necessaria una verifica della funzionalità della struttura dati ridefinita, svolta per mezzo di testing con "programmi-giocattolo" che svolgessero semplici operazioni aritmetiche (quali somme, sottrazioni, moltiplicazioni e divisioni) fra due numeri ricevuti in ingresso da linea di comando e confrontando che il risultato ottenuto fosse il medesimo per l'operazione compiuta sia da elementi *PPE* che da elementi SPE.
Naturalmente per fare ciò ci siamo avvalsi di tipi di dati `__vector` all'interno dei quali stipare i numeri in ingresso e il risultato dell'operazione aritmetica, ma per ottenere il risultato dall'unità funzionale *SPE* si è reso necessario l'utilizzo delle funzioni che permettessero la lettura e la scrittura di dati sul canale, ovvero sul bus di intercomunicazione *EIB* per lo scambio di informazioni fra *PPE* e *SPE* e viceversa.
Questi mini-programmini ci hanno permesso anche di comprendere più a fondo e di toccare con mano le regole per rendere i nostri programmi *SIMD* (Single Instruction Multiple Data), condizione necessaria e sufficiente per sfruttare al meglio le capacità del *Cell BE* in termini di spazio occupato in memoria e tempo di esecuzione. Infatti il processore *Cell BE*, come detto in precedenza, sfrutta tipi di dati *SIMD* con una moltitudine di dati per eseguire in parallelo su di essi operazioni aritmetiche piuttosto che logiche,

sia nell'elemento *PPE* che nelle singole *SPU*.
Qua sotto il listato del lato *SPE* (il lato *PPE* è visibile nell'Appendice B) di uno dei programmi che confronta il risultato di una somma algebrica di due numeri, ricevuti in input da riga di comando, effettuata sulla *PPE* e sulla *SPE*.

Codice SPE di `a_d_vec_spu.c`:

```
#include <stdio.h>
#include <spu_intrinsics.h>
#include <spu_mfcio.h>

float in1_spe[1] __attribute__((aligned(16)));
float in2_spe[1] __attribute__((aligned(16)));
float out_spe[1] __attribute__((aligned(16)));

typedef struct {
    unsigned long long  ea_in1;
    unsigned long long  ea_in2;
    unsigned long long  ea_out;
    unsigned int        size;
    int                 pad[ 1];
} abs_params_t;
abs_params_t abs_params __attribute__((aligned(16))) ;

int main(unsigned long long spe, unsigned long long argp, unsigned long long envp)
{
   int s, i, tag = 1;
   __vector float *vin1 = (__vector float *) in1_spe;
   __vector float *vin2 = (__vector float *) in2_spe;
   __vector float *vout = (__vector float *) out_spe;
   __vector float vsum  = (__vector float) { 0 };
            float *sum = (float *) &vsum ;

   // Input parameter parm is a pointer to the context.
   // Fetch the context, waiting for it to complete.
   spu_mfcdma64(&abs_params, mfc_ea2h(argp), mfc_ea2l(argp),
                sizeof(abs_params), tag, MFC_GET_CMD);
   spu_writech(MFC_WrTagMask, 1 << tag);
   spu_mfcstat(MFC_TAG_UPDATE_ALL);
```

```
   // Fetch the data. Wait for DMA to complete before performing computation
   spu_mfcdma64((void *)vin1, mfc_ea2h(abs_params.ea_in1), mfc_ea2l(abs_params.ea_in1),
                abs_params.size, tag, MFC_GET_CMD);
   spu_writech(MFC_WrTagMask, 1 << tag) ;
   spu_mfcdma64((void *)vin2, mfc_ea2h(abs_params.ea_in2), mfc_ea2l(abs_params.ea_in2),
                abs_params.size, tag, MFC_GET_CMD);
   spu_writech(MFC_WrTagMask, 1 << tag) ;
   spu_mfcstat(MFC_TAG_UPDATE_ALL);

   // Compute the two numbers
   vsum = *vin1 + *vin2;
   /* send computed sum back to master */
   *vout = vsum ;
   printf("Final Result SPE #= %f\n", *vout);

   // Put data back into main storage
   spu_mfcdma64(vout, mfc_ea2h(abs_params.ea_out), mfc_ea2l(abs_params.ea_out),
                 sizeof(abs_params.size), tag, MFC_PUT_CMD) ;
   spu_writech( MFC_WrTagMask, 1 << tag) ;
   // Wait DMA to complete before terminating SPE thread
   spu_mfcstat(MFC_TAG_UPDATE_ALL);
   return 0;
}
```

4.3 Mancato funzionamento di `__builtin` per lettura e scrittura su canale

Mentre ci accingevamo a testare e compilare i mini-programmini di cui sopra ci siamo imbattuti nel mancato funzionamento delle cosiddette *Macro*, fatte con `__builtin`, relative alla lettura e scrittura su canale, ovvero principalmente di `__builtin_rdch` e `__builtin_wrch`, oltre ad altre tre `__builtin` ad esse collegate, che generavano errori " *undefined reference to '__builtin_si_wrch'* " e " *undefined reference to '__builtin_si_rdch'* ". Quindi ci siamo ritrovati dinnanzi ad un bivio:

- scrivere le *Macro* mancanti (o meglio non funzionanti per *Clang*) andando a modificare il compilatore *LLVM* in modo che permangano;

- implementare queste funzionalità in *Clang* facendo una libreria a parte da importare ogni volta nei nostri programmi.

Dando un'occhiata alla libreria "`spu_internals.h`" le uniche parti che definivano le nostre funzioni `rdch`, `wrch` e le altre tre ad esse correlate, grazie alle quali si possono utilizzare le funzioni `spu_mfcdma32(.)`, `spu_writech(.)` e `spu_mfcstat(.)` che le contengono, erano infatti:

```
#define si_rdch(imm)          __builtin_si_rdch(imm)
#define si_wrch(imm,ra)       __builtin_si_wrch(imm,ra)
#define si_from_uint(scalar) __builtin_si_from_uint(scalar)
#define si_from_ptr(scalar)  __builtin_si_from_ptr(scalar)
#define si_to_uint(ra)        __builtin_si_to_uint(ra)
```

Dopo aver provato ad intraprendere l'ardua via della modifica del compilatore fino alla consapevolezza dell'eccessiva complessità nella risoluzione del problema, siamo tornati sui nostri passi ed abbiamo optato per la creazione di una ben più veloce e intuitiva libreria che abbiamo chiamato "`rdch_wrch.h`". Al suo interno abbiamo provveduto a inserire le istruzioni direttamente in *Assembler*, indicate nel *Data Sheet* del *Cell BE* come in figura 4.3, che le `__builtin` avrebbero restituito quando le *Macro* fossero chiamate dal compilatore.

```
- (void) spu_writech ( (enumeration) channel, (int/unsigned int) a )
- (void) wrch ( (enumeration) channel, (uint32_t) a )

- (void) si_wrch(channel, si_from_int(a) )
  (void) si_wrch(channel, si_from_uint(a))
```

```
- (unsigned int) spu_readch ( (enumeration) channel )
- (uint32_t) rdch ( (enumeration) channel )

- (unsigned int) si_to_uint ( (qword) a )
  (unsigned int) d = si_to_uint(si_rdch(channel))
```

Figura 4.3: Datasheet riguardante le funzioni `rdch` e `wrch`

Naturalmente abbiamo precedentemente acquisito dimestichezza con l'*Assembler* leggendo il codice macchina restituito dai programmi precedentemente menzionati e da altri "programmi-esempio" e "programmi-tutorial" di cui il *Cell BE* è provvisto.
Il risultato quindi è una nuova libreria "`rdch_wrch.h`", da inserire attraverso istruzioni di

```
#include <rdch_wrch.h>
```

in "`spu_internals.h`", il cui contenuto parziale è visibile nel listato qua sotto (il codice completo è riportato in Appendice C).

```
unsigned int __builtin_si_rdch (unsigned int chan)
{
  register unsigned int dest;
  if (chan == SPU_RdEventStat)  // 0
    __asm__ __volatile__( "\trdch %0, $ch0" : "=r" (dest) : );
  if (chan == SPU_WrEventMask)  // 1
    __asm__ __volatile__( "\trdch %0, $ch1" : "=r" (dest) : );
  /* ... */
  if (chan == SPU_RdInMbox)  // 29
    __asm__ __volatile__( "\trdch %0, $ch29" : "=r" (dest) : );
  if (chan == SPU_WrOutIntrMbox)  // 30
    __asm__ __volatile__( "\trdch %0, $ch30" : "=r" (dest) : );
  return dest;
}

void __builtin_si_wrch (unsigned int chan, unsigned int cont)
{
  if (chan == SPU_RdEventStat)  // 0
    __asm__ __volatile__( "\twrch $ch0, %0" : : "r" (cont) );
  if (chan == SPU_WrEventMask)  // 1
    __asm__ __volatile__( "\twrch $ch1, %0" : : "r" (cont) );
  /* ... */
  if (chan == SPU_RdInMbox)  // 29
    __asm__ __volatile__( "\twrch $ch29, %0" : : "r" (cont) );
  if (chan == SPU_WrOutIntrMbox)  // 30
    __asm__ __volatile__( "\twrch $ch30, %0" : : "r" (cont) );
}
```

4.4 Mancato funzionamento di `__builtin` per estrazione di valori da un `__vector`

Nello sviluppo di alcuni programmi di *analisi d'immagine* si è reso necessario l'utilizzo di una funzione di `extract` specifica e "target-dependent" per le *SPU*, la chiamata a `spu_extract` che per un determinato `vector` di un certo tipo di dati restituisce l'elemento del vettore voluto.
Quindi non essendo una singola funzione, ma una funzione per ogni tipo di dati differente seppur avente lo stesso nome `spu_extract`, una soluzione simile a quella adoperata per il precedente problema ovvero l'implementazione in una libreria non era realizzabile in modo completo ma parzialmente per risolvere l'errore " *use of unknown builtin '__builtin_spu_extract'* " generato. Questo perchè l'*overriding*, ovvero l'implementazione di funzioni con lo stesso nome per parametri differenti in numero o tipo, non è accettato da *Clang*.

```
#include "llvm/Pass.h"
#include "llvm/Function.h"
#include "llvm/Support/raw_ostream.h"

using namespace llvm;

namespace {
  struct Hello : public FunctionPass {

    static char ID;
    Hello() : FunctionPass(ID) {}

    virtual bool runOnFunction(Function &F) {
      errs() << "Hello: ";
      errs().write_escaped(F.getName()) << '\n';
      return false;
    }

  };
}

char Hello::ID = 0;
static RegisterPass<Hello> X("hello", "Hello World Pass", false, false);
```

Figura 4.4: Pass di `Hello World`

Per queste ragioni si è reso necessario l'ausilio di un *pass*, un artificio software, che permettesse di sostituire a *run-time* una funzione "`spu_extract()`" vuota che non fa nulla, quindi fittizia, con la relativa e specifica `spu_extract_TYPE(.)` dove `TYPE` sta a indicare la determinata `spu_extract` che ricevuto

un `vector` il cui tipo di dati è, per l'appunto, `TYPE` restituisce l'elemento voluto dal vettore che è di tipo `TYPE`. Quindi nella nostra libreria avremo l'implementazione della fittizia `spu_extract()` e delle funzioni da sostituire ad essa quali `spu_extract_signed_char(.)`, `spu_extract_longlong(.)`, `spu_extract_double(.)` e così via.

```
Function* targetFunc = ...;

class OurFunctionPass : public FunctionPass {
  public:
    OurFunctionPass(): callCounter(0) { }

    virtual runOnFunction(Function& F) {
      for (Function::iterator b = F.begin(), be = F.end(); b != be; ++b) {
        for (BasicBlock::iterator i = b->begin(), ie = b->end(); i != ie; ++i) {
          if (CallInst* callInst = dyn_cast<CallInst>(&*i)) {
            // We know we've encountered a call instruction, so we
            // need to determine if it's a call to the
            // function pointed to by m_func or not.
            if (callInst->getCalledFunction() == targetFunc)
              ++callCounter;
          }
        }
      }
    }

  private:
    unsigned callCounter;
};
```

Figura 4.5: Pass per trovare le `Call Sites`

Utilizzando come modello il classico esempio "*hello world*" di un pass, come mostrato in figura 4.4, e utilizzando un esempio simile al nostro (visibile in figura 4.5) il cui pseudo-codice è il seguente:

```
initialize callCounter to zero
for each Function f in the Module
  for each BasicBlock b in f
    for each Instruction i in b
      if (i is a CallInst and calls the given function)
        increment callCounter
```

siamo riusciti a costruire il nostro nuovo pass. Di grazie aiuto sono stati anche altri esempi di *Pass* funzionanti presenti sempre in `Transform` come "`InstructionCombining`".

Abbiamo realizzato un *pass* che per l'appunto abbiamo chiamato "*extract*" la cui implementazione è definita nel codice sorgente `Extract.cpp`, il cui contenuto parziale (il codice completo è riportato in Appendice D) è visibile qua sotto e contenuto nelle librerie `Transform` dell'*LLVM*.

```
/* ... */
namespace {
  // Extract - The implementation
  Function* targetFunc,    * uc_targetFunc, * c_targetFunc, * us_targetFunc,
          * s_targetFunc,  * ui_targetFunc, * i_targetFunc, * ull_targetFunc,
          * ll_targetFunc, *f_targetFunc,   * d_targetFunc;

struct Extract : public FunctionPass {

    Extract() : FunctionPass(ID), extractFlag(true) {}
    virtual bool doInitialization(Module &M) {
      //  VectorTyID   14: SIMD 'packed' format, or other vector type
      const Type *BP1Ty = Type::getPrimitiveType(M.getContext(), (Type::TypeID) 14);
      const Type *BP2Ty = Type::getInt32Ty(M.getContext());

      f_targetFunc  = (Function *) M.getOrInsertFunction("__builtin_spu_extract_float",
                        Type::getFloatTy(M.getContext()),  BP1Ty, BP2Ty, (Type *)0);
      d_targetFunc  = (Function *) M.getOrInsertFunction("__builtin_spu_extract_double",
                        Type::getDoubleTy(M.getContext()), BP1Ty, BP2Ty, (Type *)0);
      /* ... */
      ll_targetFunc = (Function *) M.getOrInsertFunction("__builtin_spu_extract_longlong",
                        Type::getInt64Ty(M.getContext()), BP1Ty, BP2Ty, (Type *)0);
    }
    virtual bool doInitialization(Function &F) {
      if (strcmp(F.getName().str().c_str(), "__builtin_spu_extract") == 0)
        targetFunc = &F;
      if (strcmp(F.getName().str().c_str(), "__builtin_spu_extract_float") == 0)
        f_targetFunc = &F;
      if (strcmp(F.getName().str().c_str(), "__builtin_spu_extract_double") == 0)
        d_targetFunc = &F;
      /* ... */
      if (strcmp(F.getName().str().c_str(), "__builtin_spu_extract_longlong") == 0)
        ll_targetFunc = &F;
    }
    virtual bool runOnFunction(Function &F) {
      for (Function::iterator b = F.begin(), be = F.end(); b != be; ++b) {
```

```
        BasicBlock::InstListType &BBIL = b->getInstList();
        for (BasicBlock::iterator i = b->begin(), ie = b->end(); i != ie; ++i) {
          if (CallInst* callInst = dyn_cast<CallInst>(&*i)) {
            // We know we've encountered a call instruction, so we
            // need to determine if it's a call to the
            // function pointed to by spu_extract or not.
            if (callInst->getCalledFunction() == targetFunc) {
              if(callInst->getArgOperand(0)->getType()->isVectorTy()) {
                if(callInst->getArgOperand(0)->getType()->isFPOrFPVectorTy()) {
                  if(callInst->getArgOperand(0)->getType()
                                ->getScalarType()->isFloatTy()) {
                    // insert a call to a f_targetFunction
                    CallInst *new_call=CallInst::Create((Constant *)f_targetFunc,"", i);
                    new_call->setTailCall();
                    callInst->replaceAllUsesWith(new_call);
                    // Delete old instruction
                    i = --BBIL.erase(i);
                    Changed = true;
                    ++ExtractCounter;
                  } // end if isFloatTy() / float
     /* ... */
                  if(callInst->getArgOperand(0)->getType()
                                ->getScalarType()->isIntegerTy(64)) {
                    // insert a call to a ll_targetFunction
                    CallInst *new_call=CallInst::Create((Constant *)ll_targetFunc,"", i);
                    new_call->setTailCall();
                    callInst->replaceAllUsesWith(new_call);
                    // Delete old instruction
                    i = --BBIL.erase(i);
                    Changed = true;
                    ++ExtractCounter;
                  } // end if isIntegerTy(64) / long long
              } // end if
            } // end if
          } // end for each BasicBlock::iterator i
        } // end for each Function::iterator b
      }
    };  // end of struct Extract
}   // end of anonymous namespace
```

Grazie a questo *Pass* di nome `extract` e ad altre modiche necessarie al funzionamento abbiamo potuto sostituire la funzione vuota di `__builtin_spu_extract` con le funzioni definite per ogni tipo di dato, inserite nella libreria "`rdch_wrch.h`" precedentemente creata, il cui codice è per l'appunto una semplice stampa di codice assembler preso dal *Datasheet* del *Cell BE* contenente tutte le varie `Intrisics`.
Alcuni esempi di altri *Pass* contenuti in `Transform` ci hanno permesso di poter eseguire il nostro *Pass*, tramite inserimento da riga di comando dell'opzione "`-extract`", non solo su `opt` come indicava l'esempio di *Hello World*, ma anche sul nostro compilatore *Clang*.
Di seguito una parte aggiunte riportate alla libreria "`rdch_wrch.h`" (presente in Appendice B):

```
char __builtin_spu_extract_char (__vector char vec, int elem) {
  register char dest;
  __asm__ __volatile__( "\trotqbyi %0, %1, (%2*1-3)%%16"
                        : "=r" (dest) : "r" (vec), "r" (elem) );
  return dest;
}
short  __builtin_spu_extract_short (__vector short vec, int elem) {
  register unsigned short dest;
  __asm__ __volatile__( "\trotqbyi %0, %1, ((%2-1)*2+0)%%16"
                        : "=r" (dest) : "r" (vec), "r" (elem) );
  return dest;
}
/* ... */
float __builtin_spu_extract_float (__vector float vec, int elem) {
  register float dest;
  __asm__ __volatile__( "\trotqbyi %0, %1, (%2*4+0)%%16"
                        : "=r" (dest) : "r" (vec), "r" (elem) );
  return dest;
}
double __builtin_spu_extract_double (__vector double vec, int elem) {
  register double dest;
  __asm__ __volatile__( "\trotqbyi %0, %1, (%2*8+0)%%16"
                        : "=r" (dest) : "r" (vec), "r" (elem) );
  return dest;
}
```

A corredo e in parallelo alla costruzione del *Pass extract*, in modo da avere feedback, come buona prassi abbiamo verificato che la sostituzione venisse fatta correttamente (leggendo e facendo generare anche direttamente il codice *assembler*) con l'ausilio di diverse versioni di un programma giocattolo di test denominato per l'appunto "`test_extract.c`", il cui listato della versione più estesa del lato *SPE* è riportato sotto.

```
#include <stdio.h>
#include <spu_intrinsics.h>
#include <spu_mfcio.h>

float vec_spe[1]  __attribute__((aligned(16)));
float scal_spe[1] __attribute__((aligned(16)));
typedef struct {
    unsigned long long  vec;
    unsigned long long  scal;
    unsigned int        size;
    int                 pad[ 3];
} abs_params_t;
abs_params_t abs_params __attribute__((aligned(16)));

int main(unsigned long long spe, unsigned long long argp, unsigned long long envp)
{
   int   tag = 1;
   int   s;
   float a;
   __vector float *vvec  = (__vector float *) vec_spe;
   __vector float *vscal = (__vector float *) scal_spe;
   __vector float vsum   = (__vector float) { 0 };
            float *sum = (float *) &vsum ;

  spu_mfcdma64(&abs_params, mfc_ea2h(argp), mfc_ea2l(argp),
                  sizeof(abs_params_t), tag, MFC_GET_CMD);
  spu_writech(MFC_WrTagMask, 1 << tag);
  spu_mfcstat(MFC_TAG_UPDATE_ALL);

  if(abs_params.size % 16 != 0)
        // the array size must have size multiple of 16
  // Fetch the data. Wait for DMA to complete before performing computation
  spu_mfcdma64(vvec, mfc_ea2h(abs_params.vec), mfc_ea2l(abs_params.vec),
                  abs_params.size, tag, MFC_GET_CMD);
```

```
  spu_writech(MFC_WrTagMask, 1 << tag);
  spu_mfcstat(MFC_TAG_UPDATE_ALL);

  a=spu_extract(*vvec, 1);

  // Compute the two numbers
  vsum = *vvec;
  /* send computed sum back to master */
  *vscal = vsum ;

  // Put data back into main storage
  spu_mfcdma64(vscal, mfc_ea2h(abs_params.scal), mfc_ea2l(abs_params.scal),
                   sizeof(abs_params.size), tag, MFC_PUT_CMD) ;
  spu_writech( MFC_WrTagMask, 1 << tag) ;
  // Wait DMA to complete before terminating SPE thread
  spu_mfcstat(MFC_TAG_UPDATE_ALL);

  return 0;
}
```

Capitolo 5

Analisi di immagine

L'*analisi d'immagine* o *image analysis* è l'estrazione di informazioni significative da immagini, principalmente immagini digitali, per mezzo di tecniche di elaborazione di immagini (*digital image processing*). Le operazioni svolte possono variare da lettura di `tag` a task più complesse come il riconoscimento di una persona dal suo volto.
L'analisi d'immagine mediante computer possono fare anche un uso massivo e intensivo di *pattern recognition*, geometria digitale e elaborazione di segnali.
Il *processing* di immagini compie:

- letture/scritture dell'immagine da disco,
- la manipola, processandola, e
- restituisce il risultato in output.

Un'immagine è un array bi-dimensionale di numeri, ognuno dei quali stabilisce univocamente il colore o il livello di grigio di ogni elemento dell'immagine (*pixel*), perciò in un'immagine in bianco e nero ogni *pixel* assume valore pari a 0 o 1.
Se l'immagine possiede gradazioni di grigio, ogni valore dell'*array* assume un valore compreso fra zero e il numero delle gradazioni, in base alla risoluzione dell'apparecchio che ha generato l'immagine. L'uomo può distinguere 40 livelli di grigio perciò un numero di gradazioni pari almeno a 256 rende l'immagine come una fotografia.
Il colore in un'immagine la rende simile ai livelli di grigio nel precedente esempio con la differenza che vi sono tre canali, tre bande principali corrispondenti al rosso, al verde e al blu. Con ciò ogni pixel assume tre valori

uno per ogni colore.
I formati d'immagine su cui ci siamo concentrati sono `TIFF` ("Tagged Image File Format") e BMP ("Windows bit mapped"). Il formato del file specifica come memorizzare l'immagine e le informazioni ad essa correlate.

5.1 Formato *TIFF*

Dalla collaborazione di molte aziende produttrici di computer e scanner fu creato uno standard industriale per lo scambio di dati basati su immagini digitali di tipo *raster*, la specifica *TIFF* ("Tagged Image File Format" o "Tag Image File Format") sviluppata da Aldus nella metà degli anni '80, supportato da molti produttori di scanner che diventò la base dell'*image processing*.
Gli obiettivi della specifica *TIFF* sono l'estendibilità, la portabilità e la modificabilità in modo tale che i sistemi operativi ne permettessero la modifica, l'elaborazione e il cambio dei file *TIFF*.
Le specifiche del formato *TIFF* permettono quindi una notevole flessibilità, vantaggio di per sé, che rende però difficile scrivere un interprete pienamente conforme alle specifiche, comportando perciò che una stessa immagine possa essere visualizzata con colori differenti a seconda dell'interprete che si utilizza.
Largamente utilizzato per lo scambio di immagini *raster* fra stampanti e scanner perché permette di specificare numerose indicazioni aggiuntive come tabelle di gamut o informazioni sulla calibratura del colore, il *TIFF* è quindi usato per far comunicare più macchine all'interno dello stesso studio fotografico o di editing che hanno la stessa calibratura.

Permette di rappresentare immagini con diversi spazi di colore:

- Scale di grigio (o *halftoning*),
- *RGB* (modello di colori di tipo additivo e basato su rosso, verde e blu),
- *CMYK* (modello di colori in quadricromia basato su ciano, magenta, giallo e *key black*),
- *CIE L*a*b*.

```
SubfileType
     Tag = 255 (FF)     Type = short       N = 1
     Indicates the kind of data in the subfile.

ImageWidth
     Tag = 256 (100)    Type = short       N = 1
     The width (x or horizontal) of the image in pixels.

ImageLength
     Tag = 257 (101)    Type = short       N = 1
     The length (y or height or vertical) of the image in pixels.

RowsPerStrip
     Tag = 278 (116)    Type = long        N = 1
     The number of rows per strip.
     The default is the entire image in one strip.

StripOffsets
     Tag = 273 (111)    Type = short or long  N = strips per image
     The byte offset for each strip.

StripByteCounts
     Tag = 279 (117)    Type = long        N = 1
     The number of bytes in each strip.

SamplesPerPixel
     Tag = 277 (115)    Type = short       N = 1
     The number of samples per pixel
     (1 for monochrome data, 3 for color).

BitsPerSample
     Tag = 258 (102)    Type = short       N = SamplesPerPixel
     The number of bits per pixel.  2**BitsPerSample = # of gray levels.
```

Figura 5.1: Tag standard di un file TIFF

Sono possibili anche diversi formati di compressione:

- *LZW* e *ZIP* sono di tipo lossless, senza perdita di informazione;
- *JPEG* di tipo "lossy", con perdita di informazione.

Le immagini possono essere memorizzate, oltre che come linee di scansione, anche in riquadri, permettendo di avere un rapido accesso a immagini di grosse dimensioni.
Attualmente è un formato diffuso per immagini in *JPEG* e *PNG* dall'elevato numero di bpp (bit per pixel, bit dedicati al colore del singolo pixel).
Il nome "tag" in Tag Image File Format si riferisce alla struttura base del file in cui un *tag TIFF* fornisce le informazioni sull'immagine come l'altezza (width o ampiezza), la larghezza (length o height) e il numero di pixel.

I tag sono organizzati in *tag directories* in cui i puntatori li portano da una *directory* ad un'altra, in modo tale che il risultato sia un formato di file flessibile, modificabile e riusabile.
In figura 5.1 sono mostrati i tag standard, mentre in figura 5.2 è riportata la struttura di un TIFF in cui i primi 8 bit, standard per ogni file TIFF, costituiscono l'*header*. I seguenti bit sono diversi da immagine a immagine. L'*Image File Directory* (o IFD) contiene il numero di elementi della directory e gli elementi stessi, la cui struttura è mostrata nella colonna a destra della figura **??**. Ogni elemento contiene un tag che indica il tipo di informazione del file, il tipo, la lunghezza e un puntatore all'informazione stessa.

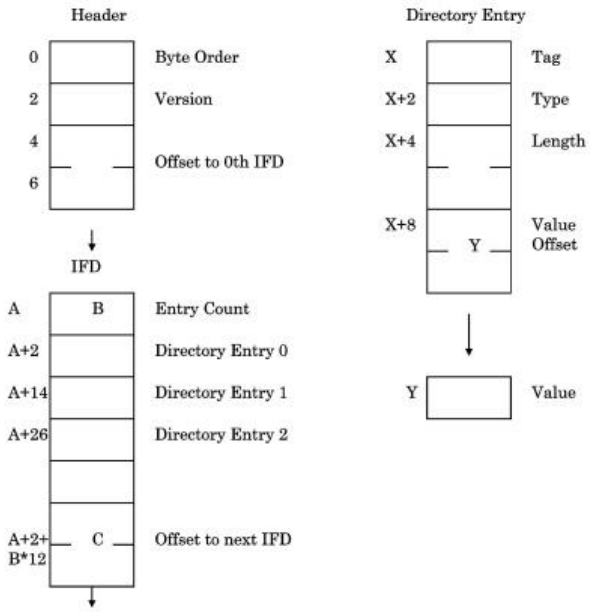

Figura 5.2: Struttura di un file TIFF

5.2 Formato *BMP*

Il formato di file "Windows bit mapped" (*BMP* o Windows bitmap), introdotto con Windows 3.0 nel 1990, è quello più diffuso per i sistemi Microsoft Windows ed ha il pregio di essere più semplice, più facile da leggere e scrivere e meno pesante del fomato di dati *TIFF*, sebbene tecnicamente non sia migliore del *TIFF*.

Essendo il *BMP* un formato di dati nativo di Windows la maggior parte delle applicazioni basate su di esso lo supportano. Inoltre essendo stato creato solo per i processori Intel, i bit meno significativi vengono prima rispetto al formato *TIFF* in cui il bit più significativo può essere a destra o a sinistra. Ci sono cinque versioni dei file *BMP* sebbene quella più diffusa sia la terza con 8 bit per pixel, livelli di grigio e senza compressione.
I file *BMP* hanno:

- un file di header,
- un bit map header,
- una tabella dei colori e
- i dati dell'immagine.

L'header del file mostrato in figura 5.3 occupa i primi 14 byte di ogni file *BMP* dei quali i primi due sono il tipo del file (sempre uguale al valore decimale 4D42 o "BM"), i seguenti quattro rappresentano la dimensione dell'immagine, i due successivi sono riservati (sempre uguali a zero) e gli ultimi quattro byte danno l'offset dei dati dell'immagine.

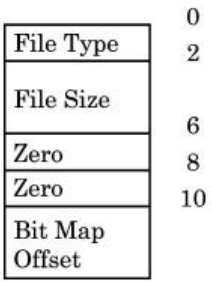

Figura 5.3: File header del file BMP

I successivi 40 byte, mostrati in figura 5.4, costituiscono l'header *Bit Map* unico per file *BMP* versione 3.x.
La dimensione dell'header è fissata sempre a 40, mentre i bit successivi definiscono l'altezza e la larghezza dell'immagine ovvero il numero di colonne e righe (se la larghezza è negativa l'immagine è memorizzata al contrario).

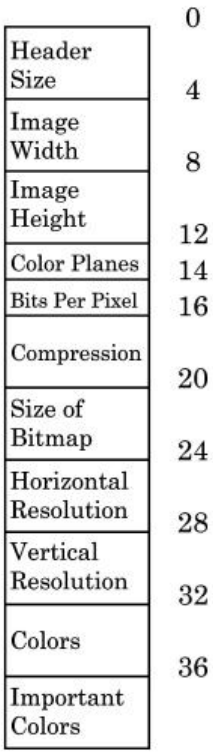

Figura 5.4: L'header Bit Map

Il numero di *color planes* è fissato a 1, mentre è preferibile che i bit per pixel siano pari a otto in modo da fornire 256 livelli di grigio. I successivi campi riguardano la compressione dell'immagine, la dimensione del campo bitmap (dimensione dell'immagine compressa), la risoluzione, i colori e i livelli di grigio dell'immagine.
Dopo i due header è presente la tabella dei colori che assegna un livello di grigio o un colore a un numero presente nei dati dell'immagine. Se il valore 12 è presente nei dati dell'immagine non significa che quel pixel è al dodicesimo livello di grigio, ma che il pixel ha il livello di grigio contenuto nel dodicesimo posto nella tabella dei colori. La tabella dei colori ha quattro byte in ogni elemento della stessa ovvero uno per il blu, il verde, il rosso e uno di riempimento sempre pari a zero. Per un'immagine a 256 livelli di grigio, la tabella dei colori è lunga 4x256 byte.
La parte finale del file *BMP* è costituita dai dati dell'immagine memorizzati riga per riga con un riempimento (padding) al termine di ogni riga che assicura le righe siano multipli di quattro.

5.3 Implementazione

Le *routine* di lettura/scrittura dell'immagine da disco hanno bisogno di leggere e scrivere su file in modo tale che il programmatore non si preoccupi dei dettagli, nascondendo le operazioni più a basso livello. Un frammento di codice con le funzioni ad alto livello è il seguente:

```
0   char   *in_name,  *out_name;
1   short  **the_image;
2   long   height,  width;
3   create_image_file(in_name,  out_name);
4   get_image_size(in_name,  &height,  &width);
5   the_image  =  allocate_image_array(height,  width);
6   read_image_array(in_name,  the_image);
7      call  an  image  processing  routine
8   write_image_array(out_name,  the_image);
9   free_image_array(the_image,  height);
```

Dopo aver dichiarato le variabili di base necessarie (path dell'immagine in input e in output, un array/vettore per l'immagine e risoluzioni) creiamo l'immagine in uscita (stesso tipo e dimensione) con `create_image_file`, leggiamo la dimensione dell'immagine in input ottenendo la risoluzione necessaria per allocare l'array/vettore dell'immagine (`get_image_size` e `allocate_image_array`), leggiamo l'array/vettore dell'immagine dal file in ingresso (di tipo TIFF o BMP) con `read_image_array`, manipoliamo l'immagine e con `write_image_array` scriviamo l'array/vettore dell'immagine manipolata nel file di output (liberando l'array di memoria precedentemente allocato con `free_image_array`).
Questa struttura di funzioni separa tutti gli accessi ai rispettivi file di input/output dalle routine di analisi d'immagine che ricevono un array/vettore di numeri e la dimensione dell'array, aumentandone la portabilità e permettendo di aggiungere facilmente altri formati senza intaccare la maggior parte del codice nel sistema di *processing* dell'immagine.
Ad esempio la funzione `read_image_array` è ad alto livello, non fa che chiamare sotto-procedure che determinano il formato del file in ingresso, settando e leggendo gli header, e lo leggono (`write_image_array` è ovviamente il duale).

5.3.1 Halftoning

L'*halftoning* è una tecnica di riproduzione che simula la continuità di tonalità attraverso l'uso di punti variabili in dimensioni, forma oppure posizionamento. In pratica il processo di halftone ("mezza colorazione") riduce la riproduzione visuale di un'immagine contenente molti più livelli di grigio in un'immagine binaria stampata solo con un colore d'inchiostro (il nero di solito). Questa riproduzione binaria si affida un'illusione ottica di base nella quale questi punti danno l'impressione all'occhio umano che l'immagine abbia una tonalità più soffusa e sbiandita, "smooth", rispetto all'originale.
Qua sotto è riportato l'algoritmo di haltoning in pseudo-codice che converte un'immagine I con R righe e C colonne con vari livelli di grigio in una che contiene solo 1 o 0 per mezzo di una tecnica di diffusione dell'errore (quando supera una certa soglia viene settato a 1 o a 0).

```
I(R,C)  - input image with R rows and C columns
Ep(m,n) - sum of the errors propogated to position (m,n) due to prior assignments
Eg(m,n) - the total error generated at position (m,n).
C(i,j)  - the error distribution function with I rows and J columns

1. Set Ep(m,n) = Eg(m,n) = 0 for R rows and C columns
2. loop m = 1,R
   3. loop n = 1,C
      4. Calculate the total propogated error at (m,n) due to prior assignments
      5. Sum the current pixel value and the total propogated error: T = I(m,n) + Ep(m,n)
      6. IF T > threshold
            THEN do steps 7. and 8.
            ELSE do steps 9. and 10.
         7. Set pixel (m,n) on
         8. Calculate error generated at current location Eg(m,n) = T - 2*threshold
         9. Set pixel (m,n) off
         10. Calculate error generated at current location Eg(m,n) = threshold
   3. end loop over n
2. end loop over m
```

$E_p(m,n)$ è la somma degli errori propagati alla posizione (m,n) da precedenti assegnamenti pari a 1 o 0, mentre $E_g(m,n)$ è l'errore globale generato alla posizione (m,n). $C(i,j)$ è la funzione di distribuzione dell'errore, la cui dimensione e i cui valori sono stati settati sperimentalmente in modo da ottenere i risultati migliori.

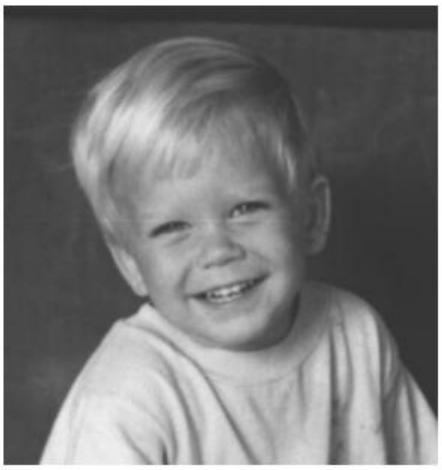

Figura 5.5: Immagine in ingresso

Figura 5.6: Immagine in uscita dopo il processo di halftoning

L'autore originario [1] ha settato C come una matrice 2x3 con i valori mostrati nell'equazione seguente:

$$C_{ij} = \begin{matrix} 0.0 & 0.2 & 0.0 \\ 0.6 & 0.1 & 0.1 \end{matrix}$$

Un esempio di risultato di processo di halftoning dell'immagine in Figura 5.5 è quello mostrato in Figura 5.6, dove la soglia (threshold) è stata settata a 128 (metà di 256). Il risultato ottenuto dall'applicazione dell'*image processing* attraverso halftoning è facilmente riconoscibile rispetto all'immagine in ingresso.

[1] Ci stiamo riferendo a John A. Saghri, Hsieh S. Hou e Andrew F. Tescher, autori del testo *Personal Computer Based Image Processing with Halftoning* edito da Optical Engineering

Il codice che implementa l'algoritmo utilizzato per l'operazione di halftoning è riportato qua sotto, mentre in Appendice E è risportato il codice modificato e SIMDizzato: le operazioni di lettura e scrittura sono affidate al lato *PPE* mentre la *SPE* si occupa del vero e proprio halftoning.

```
#include "cips.h"

half_tone(short **in_image, short **out_image, short threshold,
          short one, short zero, long rows, long cols)
{
  float **eg, **ep;
  float c[2][3],
  sum_p,
  t;
  int i, j, m, n, xx, yy;

  c[0][0] = 0.0;
  c[0][1] = 0.2;
  c[0][2] = 0.0;
  c[1][0] = 0.6;
  c[1][1] = 0.1;
  c[1][2] = 0.1;

  // Calculate the total propogated error at location (m,n) due to prior assignment.
  // Go through the input image. If the output should be one then
  // display that pixel as such. If the output should be zero then display it that way.
  // Also set the pixels in the input image array to 1's and 0's in case
  // the print option was chosen.

  eg = malloc(rows * sizeof(float *));
  for(i=0; i<rows; i++){
    eg[i] = malloc(cols * sizeof(float ));
    if(eg[i] == '\0')
      printf("\n\tmalloc of eg[%d] failed", i);
  } /* ends loop over i */
  ep = malloc(rows * sizeof(float *));
  for(i=0; i<rows; i++){
    ep[i] = malloc(cols * sizeof(float ));
    if(ep[i] == '\0')
      printf("\n\tmalloc of ep[%d] failed", i);
  } /* ends loop over i */
```

```
  for(i=0; i<rows; i++){
    for(j=0; j<cols; j++){
      eg[i][j] = 0.0;
      ep[i][j] = 0.0;
    }
  }
  for(m=0; m<rows; m++){
    for(n=0; n<cols; n++){
      sum_p = 0.0;
      for(i=0; i<2; i++){
        for(j=0; j<3; j++){
          xx = m-i+1;
          yy = n-j+1;
          if(xx < 0) xx = 0;
          if(xx >= rows) xx = rows-1;
          if(yy < 0) yy = 0;
          if(yy >= cols) yy = cols-1;
          sum_p = sum_p + c[i][j] * eg[xx][yy];
        } /* ends loop over j */
      } /* ends loop over i */
      ep[m][n] = sum_p;
      t = in_image[m][n] + ep[m][n];
      // Here set the point [m][n]=one
      if(t > threshold){
        eg[m][n] = t - threshold*2;
        out_image[m][n] = one;
      } /* ends if t > threshold */
      // Here set the point [m][n]=zero
      else{ /* t <= threshold */
        eg[m][n] = t;
        out_image[m][n] = zero;
      } /* ends else t <= threshold */
    } /* ends loop over n columns */
  } /* ends loop over m rows */
  for(i=0; i<rows; i++){
    free(eg[i]);
    free(ep[i]);
  }
} /* ends half_tone */
```

Capitolo 6

Conclusione e sviluppi futuri

Durante questo lavoro di tesi le soluzioni migliori che abbiamo trovato per risolvere i problemi presentati in precedenza sono quelle esposte sopra, ciò non toglie che esista una soluzione più ottimizzata di quella presentata che diverrebbe un subottimo.
Infatti sarebbe interessante poter rendere questo compilatore ancora più completo ed efficiente, riducendo notevolmente i tempi di generazione del codice oppure escogitando un algoritmo o un differente scheduling in grado di innalzare sensibilmente le prestazioni in termini di memoria allocata piuttosto che implementare una miglioria per entrambi.
La priorità attuale però resta la necessità di aumentare la compatibilità del compilatore *Clang* con l'architettura del processore *Cell BE*, implementando nello specifico `__builitin` necessarie all'utilizzo di altre funzionalità principali e secondarie dell'elemento *SPE* che attualmente mancano, sebbene ciò possa comportare uno sforzo tutt'altro che indifferente.
Infine non meno interessante sarebbe valutare se il costo necessario per incrementare la funzionalità del suddetto applicativo, giustificasse i risultati ottenuti in termini sia economici che di tempo, piuttosto che adottare l'opzione di mantenere la soluzione attuale, accontentandosi così di una soluzione pur sempre subottima.

Appendice A

Tabella dei programmi testati

In tabella A.4 trovate una breve descrizione dei programmi testati con relative descrizioni.

Nome programma	Righe di codice	Descrizione
`hello_ppe.c`	53	Eseguito dalla PPE, carica ed esegue all'interno di una singola SPE il programma `hello_spe`.
`hello_spe.c`	7	Eseguito dalla SPE, stampa la stringa "Hello world".
`sum_master.c`	117	Eseguito nella PPE, passa a `sum_slave` informazioni su una serie di numeri per poi stamparne la somma dei parziali.
`sum_slave.c`	64	Eseguito in una SPE, calcola quattro somme parziali di una serie di numeri e copia i risultati nella memoria di sistema.
`adv_master.c`	247	Eseguito nella PPE, suddivide, coordina e parallelizza calcoli su più SPE per poi stamparne la somma totale dei parziali.
`adv_slave.c`	99	Eseguito nelle SPE, somma parte di valori in virgola mobile utilizzando istruzioni SIMD per più calcoli in parallelo e copia i risultati nella memoria di sistema.
`simple.c`	114	Eseguito dalla PPE, crea 8 thread SPE `simple_spu` identici a turno, al termine stampa un messaggio "Esecuzione corretta".
`simple_spu.c`	48	Embeddato in un eseguibile PPE, il programma thread SPE stampa una stringa "Hello Cell" seguita dal suo ID.
`euler _scalar.c`	100	Eseguito nella PPE, simula un sistema a particelle con Euler integration e mostra come può essere fatto il porting e l'esecuzione parallela di una funzione scalare semplificata.

Tabella A.1: Tabella dei programmi testati

Nome programma	Righe di codice	Descrizione
STEP 1a: `euler_simd _aos.c`	89	Embeddato in eseguibile PPE, il programma SPE SIMDizza il codice per l'esecuzione su sistemi a particelle del tipo array di strutture (STEP 1a).
STEP 1b: `euler_simd _soa.c`	98	Embeddato in eseguibile PPE, il programma SPE SIMDizza il codice per l'esecuzione su sistemi a particelle del tipo strutture di array (STEP 1b).
STEP 2: `euler_spe.c`	136	Embeddato in un eseguibile PPE, fa il porting del codice per l'esecuzione sulla SPE (STEP 2).
`euler _multi_spe.c`	149	Embeddato in un eseguibile PPE, parallelizza il codice in più SPU (STEP 3).
`euler_tuned _multi_spe.c`	149	Embeddato in un eseguibile PPE, ottimizza e regola i parametri per le performance (STEP 4).
`simple_dma.c`	108	Scrive semplici operazioni DMA.
`single _buffer.c`	132	Trasferisce grandi array di dati da memoria principale al LS, li processa e scrive il risultato in memoria principale con tecniche di single buffer (il processing può sovrascrivere l'input buffer).
`single_buffer _in_out.c`	150	Trasferisce grandi array di dati da memoria principale al LS, li processa e scrive il risultato in memoria principale con tecniche di single buffer (l'output non può sovrascrivere l'input buffer).
`single_buffer _list.c`	172	Trasferisce dati scattered da memoria principale al LS, li processa e scrive il risultato in memoria principale con liste DMA (il processing può sovrascrivere l'input buffer).
`single _buffer_list _in_out.c`	177	Trasferisce dati scattered da memoria principale al LS, li processa e scrive il risultato in memoria principale con liste DMA (usa un input buffer ed un output buffer, i dati sono trasferiti e processati in sequenza senza tecniche multi-buffering).
`double _buffer.c`	179	Trasferisce grandi array di dati da memoria principale al LS, li processa e scrive il risultato in memoria principale con tecniche di double buffering (con le quali la latenza di trasferimento può essere nascosta).
`double _buffer.c _in_out.c`	191	Trasferisce grandi array di dati da memoria principale al LS, li processa e scrive il risultato in memoria principale con tecniche di double buffering (con le quali la latenza di trasferimento può essere nascosta). L'output non può sovrascrivere l'input buffer, perciò sono allocati 2 input buffer e 2 output buffer.
`double _buffer.c _list.c`	239	Trasferisce dati scattered da memoria principale al LS, li processa e scrive il risultato in memoria principale usando liste DMA e tecniche di double buffering (l'output del processing può sovrascrivere l'input buffer). La latenza è nascosta dall'overlapping del trasferimento dei dati con la loro computazione.

Tabella A.2: Tabella dei programmi testati

Nome programma	Righe di codice	Descrizione
`double_buffer_list_in_out.c`	241	Trasferisce dati scattered da memoria principale al LS, li processa e scrive il risultato in memoria principale con liste DMA e tecniche di double buffering. L'output del processing non può sovrascrivere l'input buffer, perciò sono allocati 2 input buffer e 2 output buffer. Inoltre, 4 array di liste DMA sono allocati per memorizzare la lista DMA in input e quella in output (la latenza è nascosta dall'overlapping del trasferimento dei dati con la loro computazione).
`triple_buffer.c`	184	Trasferisce grandi array di dati da memoria principale al LS, li processa e scrive il risultato in memoria principale con tecniche di triple buffering (con le quali la latenza di trasferimento può essere nascosta).
`triple_buffer_in_out.c`	227	Trasferisce grandi array di dati da memoria principale al LS, li processa e scrive il risultato in memoria principale con tecniche di triple buffering (con le quali la latenza di trasferimento può essere nascosta). L'output non può sovrascrivere l'input buffer, perciò sono stati allocati 3 buffer nella LS. Il codice alterna l'uso di ogni buffer sia come input che come output.
`triple_buffer_list_in_out.c`	308	Trasferisce dati scattered da memoria principale al LS, li proessa e scrive il risultato in memoria principale con liste DMA e teniche di triple buffering. L'output del processing non può sovrascrivere l'input buffer, perciò sono stati allocati 3 local buffer. Il codice alterna l'uso di ogni buffer sia come input che come output, inoltre 4 array di liste DMA sono allocati per memorizzare la lista DMA in input e quella in ouput (la latenza è nascosta dal l'overlapping del trasferimento dei dati con la loro computazione).
`destructive_dma_list.c`	189	Mostra l'uso di liste DMA distruttive (l'array della lista è costruito alla fine dell'input buffer e sovrascritto dal trasferimento).
`mutex_example.c`	237	Come funziona un mutex. La PPU inizializza N thread SPE, tenta di ottenere il lock del buffer per aggiornarlo e poi rilasciare il lock.
`mutex_spu_example.c`	101	Ogni SPE proverà a ottenere il lock del buffer per aggiornarlo e poi rilasciare il lock.
`cond_example.c`	407	Contiene il lato PPU di come funziona una conditional variable. Setta i thread SPU e fa `cond_wait` o setta i thread e il buffer.
`cond_signal_spu_example.c`	148	Contiene il lato SPU, i thread SPU computano ed il primo che raggiunge il limite chiama `cond_signal` per svegliare la PPU.
`cond_wait_spu_example.c`	126	Contiene il lato SPU, le SPU entrano nella `cond_wait` dopo essere state avviate. Si svegliano e finiscono i loro lavori, quando la PPU chiama `cond_broadcast` dopo aver settato il buffer.

Tabella A.3: Tabella dei programmi testati

Nome programma	Righe di codice	Descrizione
atomic_op _example.c	1084	Come funzionano operazioni atomiche, con un test per ognuna (tranne atomic_read, chiamata da altri test). La PPE crea una variabile atomic e setta il numero di SPE, le SPE e la PPE aggiornano questa variabile in LOOP_COUNT iterazioni e alla fine la PPE controlla che il valore sia settato a quanto atteso.
do_atomic _add_test.c	65	Inizializzato a 0, aggiunge 1 ad ogni ciclo. Include solo le SPE oppure sia le SPE che la PPE.
do_atomic _add_return _test.c	93	Inizializzato a 0, aggiunge 1 ad ogni ciclo. Usa i return da atomic_add_return e atomic_read per confermare che stanno dando l'output corretto alla fine di ogni ciclo.
do_atomic _set_test.c	65	Ogni ciclo setta atomic al loop counter, alla fine dovrebbe essere settato a LOOP_COUNT - 1.
do_atomic _inc_test.c	65	Inizializzato a 0, è incrementato di 1 ad ogni ciclo.
do_atomic _inc_return _test.c	93	Inizializzato a 0, è incrementato di 1 ad ogni ciclo. Usa i return da atomic_inc_return e atomic_read per confermare che stanno dando l'output corretto alla fine di ogni ciclo.
do_atomic _sub_test.c	65	Inizializzato a num_spus * LOOP_COUNT, sottrae 1 ad ogni ciclo. Include solo le SPE oppure sia le SPE che la PPE.
do_atomic _sub_return _test.c	65	Inizializzato a (num_spus+1) * LOOP_COUNT, sottrae 1 ad ogni ciclo. Usa i return da atomic_sub_return e atomic_read per confermare l'output corretto alla fine di ogni ciclo.
do_atomic _sub_and _test_test.c	71	Inizializzato a (num_spus+1) * LOOP_COUNT, sottrae 1 ad ogni ciclo. Usa i return da atomic_sub_and_test per confermare che ritorni correttamente 1 solo quando atomic è settato a 0.
do_atomic _dec_test.c	65	Inizializzato a (num_spus+1) * LOOP_COUNT, è decrementato di 1 ad ogni ciclo.
do_atomic _dec_return _test.c	95	Inizializzato a (num_spus+1) * LOOP_COUNT, è decrementato di 1 ad ogni ciclo. Usa i return da atomic_dec_return e atomic_read per confermare l'output corretto alla fine di ogni ciclo.
do_atomic _dec_and _test_test.c	71	Inizializzato a (num_spus+1) * LOOP_COUNT, è decrementato di 1 ad ogni ciclo. Usa i return da atomic_dec_and_test per confermare ritorni 1 solo quando atomic è settato a 0.
do_atomic _dec_if _positive _test.c	65	Inizializzato a (num_spus-1) * LOOP_COUNT-1, è decrementato di 1 ad ogni ciclo. Conferma che alla fine, anche se dovrebbe essere decrementato sotto 0, il valore è ancora 0.

Tabella A.4: Tabella dei programmi testati

Appendice B

Listato del lato *PPE* dei mini-programmi della sezione 4.2.1

Di seguito il codice completo eseguio sulla *PPE* dal mini-programma di test `a_d_vec.c`:

```
#include <stdio.h>
#include <stdlib.h>
#include <altivec.h>
#include <libspe2.h>

float in1[1] __attribute__((aligned(16)));
float in2[1] __attribute__((aligned(16)));
float out[1] __attribute__((aligned(16)));

typedef struct {
    unsigned long long  ea_in1;
    unsigned long long  ea_in2;
    unsigned long long  ea_out;
    unsigned int        size;
    int                 pad[ 1];
} abs_params_t;
abs_params_t abs_params __attribute__((aligned(16)));
```

```
int main(int argc, char **argv)
{
   int paddedDataSize ;
   spe_program_handle_t * prog;
   spe_context_ptr_t spe;
   unsigned int entry;
   spe_stop_info_t  stop_info ;
   int  i, ret;
   double b = 0, c = 0, a2 = 0;

   if(argc < 1 || argc > 4) {
      printf("\nusage : d_vec int_number int_number");
      exit(0);
   }
   b = atof(argv[1]);  c = atof(argv[2]);
   in1[0] = 0.0f;   in2[0] = 0.0f;
   in1[0] = b;      in2[0] = c;     out[0] = 0.0f;
   paddedDataSize = sizeof(double) + 15 - (sizeof(double) + 15) % 16;
   /* Open the SPE program */
   prog = spe_image_open("a_d_vec_spu");
   if(!prog) {
      perror("spe_image_open");
      exit(1);
   }
   /* Initialize the SPE and load the SPE program */
   spe = spe_context_create(0, NULL);
   if(!spe) {
      perror("spe_context_create");
      exit(1);
   }
   ret = spe_program_load(spe, prog);
   if(ret) {
      perror("spe_program_load");
      exit(1);
   }
   /* Create and start worker tasks */
   /* Start the SPE program */
   abs_params.ea_in1 = (unsigned long) in1;
   abs_params.ea_in2 = (unsigned long) in2;
   abs_params.ea_out = (unsigned long) out;
   abs_params.size   = paddedDataSize ;
```

```
   entry = SPE_DEFAULT_ENTRY;
   ret = spe_context_run(spe, &entry, 0, &abs_params, NULL, &stop_info);
   if(ret < 0) {
      perror("spe_context_run") ;
      exit(1) ;
   }
   /* Release the SPE context and program */
   ret = spe_context_destroy(spe);
   if(ret) {
      perror("spe_context_destroy");
      exit(1);
   }
   ret = spe_image_close(prog);
   if(ret) {
      perror("spe_image_close");
      exit(1);
   }
   // Compute the two numbers
   a2 = b + c;
   printf("b ( %f) + c ( %f)  = a2 ( %f)\n", b, c, a2);
   printf("out = %d, *out = %f\n", out, *out);
   if (*out == a2)
      return 0;
   return 1;
}
```

Appendice C

Listato della libreria `rdch_wrch.h`

Di seguito il codice completo della libreria `rdch_wrch.h`:

```
unsigned int __builtin_si_to_uint (unsigned int reg) {
  register unsigned int res = (unsigned int)reg;
  return res;
}
unsigned int __builtin_si_rdch (unsigned int chan) s{
  register unsigned int dest;
  if (chan == SPU_RdEventStat)   // 0
    __asm__ __volatile__( "\trdch %0, $ch0" : "=r" (dest) : );
  if (chan == SPU_WrEventMask)   // 1
    __asm__ __volatile__( "\trdch %0, $ch1" : "=r" (dest) : );
  if (chan == SPU_WrEventAck)    // 2
    __asm__ __volatile__( "\trdch %0, $ch2" : "=r" (dest) : );
  if (chan == SPU_RdSigNotify1)  // 3
    __asm__ __volatile__( "\trdch %0, $ch3" : "=r" (dest) : );
  if (chan == SPU_RdSigNotify2)  // 4
    __asm__ __volatile__( "\trdch %0, $ch4" : "=r" (dest) : );
  if (chan == SPU_WrDec)         // 7
    __asm__ __volatile__( "\trdch %0, $ch7" : "=r" (dest) : );
  if (chan == SPU_RdDec)         // 8
    __asm__ __volatile__( "\trdch %0, $ch8" : "=r" (dest) : );
  if (chan == MFC_WrMSSyncReq)   // 9
    __asm__ __volatile__( "\trdch %0, $ch9" : "=r" (dest) : );
  if (chan == SPU_RdEventMask)   // 11
    __asm__ __volatile__( "\trdch %0, $ch11" : "=r" (dest) : );
```

```
  if (chan == MFC_RdTagMask)     // 12
    __asm__ __volatile__( "\trdch %0, $ch12" : "=r" (dest) : );
  if (chan == SPU_RdMachStat)    // 13
    __asm__ __volatile__( "\trdch %0, $ch13" : "=r" (dest) : );
  if (chan == SPU_WrSRR0)        // 14
    __asm__ __volatile__( "\trdch %0, $ch14" : "=r" (dest) : );
  if (chan == SPU_RdSRR0)        // 15
    __asm__ __volatile__( "\trdch %0, $ch15" : "=r" (dest) : );
  if (chan == MFC_LSA)           // 16
    __asm__ __volatile__( "\trdch %0, $ch16" : "=r" (dest) : );
  if (chan == MFC_EAH)           // 17
    __asm__ __volatile__( "\trdch %0, $ch17" : "=r" (dest) : );
  if (chan == MFC_EAL)           // 18
    __asm__ __volatile__( "\trdch %0, $ch18" : "=r" (dest) : );
  if (chan == MFC_Size)          // 19
    __asm__ __volatile__( "\trdch %0, $ch19" : "=r" (dest) : );
  if (chan == MFC_TagID)         // 20
    __asm__ __volatile__( "\trdch %0, $ch20" : "=r" (dest) : );
  if (chan == MFC_Cmd)           // 21
    __asm__ __volatile__( "\trdch %0, $ch21" : "=r" (dest) : );
  if (chan == MFC_WrTagMask)     // 22
    __asm__ __volatile__( "\trdch %0, $ch22" : "=r" (dest) : );
  if (chan == MFC_WrTagUpdate)   // 23
    __asm__ __volatile__( "\trdch %0, $ch23" : "=r" (dest) : );
  if (chan == MFC_RdTagStat)     // 24
    __asm__ __volatile__( "\trdch %0, $ch24" : "=r" (dest) : );
  if (chan == MFC_RdListStallStat)  // 25
    __asm__ __volatile__( "\trdch %0, $ch25" : "=r" (dest) : );
  if (chan == MFC_WrListStallAck)   // 26
    __asm__ __volatile__( "\trdch %0, $ch26" : "=r" (dest) : );
  if (chan == MFC_RdAtomicStat)    // 27
    __asm__ __volatile__( "\trdch %0, $ch27" : "=r" (dest) : );
  if (chan == SPU_WrOutMbox)       // 28
    __asm__ __volatile__( "\trdch %0, $ch28" : "=r" (dest) : );
  if (chan == SPU_RdInMbox)        // 29
    __asm__ __volatile__( "\trdch %0, $ch29" : "=r" (dest) : );
  if (chan == SPU_WrOutIntrMbox)   // 30
    __asm__ __volatile__( "\trdch %0, $ch30" : "=r" (dest) : );
  return dest;
}
```

```
unsigned int __builtin_si_from_uint (unsigned int reg) {
  register unsigned int res = (unsigned int)reg;
  return res;
}
void __builtin_si_wrch (unsigned int chan, unsigned int cont) {
  if (chan == SPU_RdEventStat)   // 0
    __asm__ __volatile__( "\twrch $ch0, %0" : : "r" (cont) );
  if (chan == SPU_WrEventMask)   // 1
    __asm__ __volatile__( "\twrch $ch1, %0" : : "r" (cont) );
  if (chan == SPU_WrEventAck)    // 2
    __asm__ __volatile__( "\twrch $ch2, %0" : : "r" (cont) );
  if (chan == SPU_RdSigNotify1)  // 3
    __asm__ __volatile__( "\twrch $ch3, %0" : : "r" (cont) );
  if (chan == SPU_RdSigNotify2)  // 4
    __asm__ __volatile__( "\twrch $ch4, %0" : : "r" (cont) );
  if (chan == SPU_WrDec)         // 7
    __asm__ __volatile__( "\twrch $ch7, %0" : : "r" (cont) );
  if (chan == SPU_RdDec)         // 8
    __asm__ __volatile__( "\twrch $ch8, %0" : : "r" (cont) );
  if (chan == MFC_WrMSSyncReq)   // 9
    __asm__ __volatile__( "\twrch $ch9, %0" : : "r" (cont) );
  if (chan == SPU_RdEventMask)   // 11
    __asm__ __volatile__( "\twrch $ch11, %0" : : "r" (cont) );
  if (chan == MFC_RdTagMask)     // 12
    __asm__ __volatile__( "\twrch $ch12, %0" : : "r" (cont) );
  if (chan == SPU_RdMachStat)    // 13
     __asm__ __volatile__( "\twrch $ch13, %0" : : "r" (cont) );
  if (chan == SPU_WrSRR0)        // 14
    __asm__ __volatile__( "\twrch $ch14, %0" : : "r" (cont) );
  if (chan == SPU_RdSRR0)        // 15
    __asm__ __volatile__( "\twrch $ch15, %0" : : "r" (cont) );
  if (chan == MFC_LSA)           // 16
    __asm__ __volatile__( "\twrch $ch16, %0" : : "r" (cont) );
  if (chan == MFC_EAH)           // 17
    __asm__ __volatile__( "\twrch $ch17, %0" : : "r" (cont) );
  if (chan == MFC_EAL)           // 18
    __asm__ __volatile__( "\twrch $ch18, %0" : : "r" (cont) );
  if (chan == MFC_Size)          // 19
    __asm__ __volatile__( "\twrch $ch19, %0" : : "r" (cont) );
  if (chan == MFC_TagID)         // 20
    __asm__ __volatile__( "\twrch $ch20, %0" : : "r" (cont) );
```

```
  if (chan == MFC_Cmd)           // 21
    __asm__ __volatile__( "\twrch $ch21, %0" : : "r" (cont) );
  if (chan == MFC_WrTagMask)     // 22
    __asm__ __volatile__( "\twrch $ch22, %0" : : "r" (cont) );
  if (chan == MFC_WrTagUpdate)   // 23
    __asm__ __volatile__( "\twrch $ch23, %0" : : "r" (cont) );
  if (chan == MFC_RdTagStat)     // 24
    __asm__ __volatile__( "\twrch $ch24, %0" : : "r" (cont) );
  if (chan == MFC_RdListStallStat)   // 25
    __asm__ __volatile__( "\twrch $ch25, %0" : : "r" (cont) );
  if (chan == MFC_WrListStallAck)   // 26
    __asm__ __volatile__( "\twrch $ch26, %0" : : "r" (cont) );
  if (chan == MFC_RdAtomicStat)     // 27
    __asm__ __volatile__( "\twrch $ch27, %0" : : "r" (cont) );
  if (chan == SPU_WrOutMbox)        // 28
    __asm__ __volatile__( "\twrch $ch28, %0" : : "r" (cont) );
  if (chan == SPU_RdInMbox)         // 29
    __asm__ __volatile__( "\twrch $ch29, %0" : : "r" (cont) );
  if (chan == SPU_WrOutIntrMbox)   // 30
    __asm__ __volatile__( "\twrch $ch30, %0" : : "r" (cont) );
}
unsigned int __builtin_si_from_ptr (void* reg) {
  unsigned int res = (unsigned int)reg;
  return res;
}

// aggiunte in seguito le sottostanti

char __builtin_spu_extract_char (__vector char vec, int elem) {
  register char dest;
  __asm__ __volatile__( "\trotqbyi %0, %1, (%2*1-3)%%16"
                        : "=r" (dest) : "r" (vec), "r" (elem) );
  return dest;
}
short  __builtin_spu_extract_short (__vector short vec, int elem) {
  register unsigned short dest;
  __asm__ __volatile__( "\trotqbyi %0, %1, ((%2-1)*2+0)%%16"
                        : "=r" (dest) : "r" (vec), "r" (elem) );
  return dest;
}
```

```
int __builtin_spu_extract_int (__vector int vec, int elem) {
  register unsigned int dest;
  __asm__ __volatile__( "\trotqbyi %0, %1, (%2*4+0)%%16"
                        : "=r" (dest) : "r" (vec), "r" (elem) );
  return dest;
}
long long __builtin_spu_extract_longlong (__vector long long vec, int elem) {
  register long long dest;
  __asm__ __volatile__( "\trotqbyi %0, %1, (%2*8+0)%%16"
                        : "=r" (dest) : "r" (vec), "r" (elem) );
  return dest;
}
float __builtin_spu_extract_float (__vector float vec, int elem) {
  register float dest;
  __asm__ __volatile__( "\trotqbyi %0, %1, (%2*4+0)%%16"
                        : "=r" (dest) : "r" (vec), "r" (elem) );
  return dest;
}
double __builtin_spu_extract_double (__vector double vec, int elem) {
  register double dest;
  __asm__ __volatile__( "\trotqbyi %0, %1, (%2*8+0)%%16"
                        : "=r" (dest) : "r" (vec), "r" (elem) );
  return dest;
}
```

Appendice D

Listato del pass *Extract* contenuto nel file `Extract.cpp`

Di seguito il codice completo del pass *Extract* contenuto nel seguente file `Extract.cpp`:

```
//===- Extract.cpp -----------------===//
//
//                     The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//===----------------------------------------------------------------------===//
// This file implements a version of the LLVM "Extract" pass needed
// for "spu_extract" instruction for Cell BE.
//===----------------------------------------------------------------------===//

#define DEBUG_TYPE "extract"
#include "llvm/Pass.h"
#include "llvm/Function.h"
#include "llvm/Support/raw_ostream.h"
#include "llvm/ADT/Statistic.h"
#include <llvm/Instructions.h>
#include <llvm/Module.h>
#include <llvm/DerivedTypes.h>
#include <llvm/LLVMContext.h>
#include <llvm/Transforms/Scalar.h>
#include <string.h>
```

```
using namespace llvm;

STATISTIC(callCounter, "Counts number of called functions");
STATISTIC(ExtractCounter, "Counts number of spu_extract functions modified");

namespace {
  // Extract - The implementation
  Function* targetFunc,    * uc_targetFunc, * c_targetFunc, * us_targetFunc,
          * s_targetFunc,  * ui_targetFunc, * i_targetFunc, * ull_targetFunc,
          * ll_targetFunc, *f_targetFunc,   * d_targetFunc;

struct Extract : public FunctionPass {

    static char ID; // Pass identification, replacement for typeid
    bool extractFlag;
    Extract() : FunctionPass(ID), extractFlag(true) {}

    Extract(bool extractFlag) : FunctionPass(ID), extractFlag(extractFlag) {}

    virtual const char *getPassName() const {
      return "Spu_extract Pass";
    }

    virtual bool doInitialization(Module &M) {

      //  VectorTyID   14: SIMD 'packed' format, or other vector type
      const Type *BP1Ty = Type::getPrimitiveType(M.getContext(), (Type::TypeID) 14);
      const Type *BP2Ty = Type::getInt32Ty(M.getContext());

      f_targetFunc  = (Function *) M.getOrInsertFunction("__builtin_spu_extract_float",
                        Type::getFloatTy(M.getContext()),  BP1Ty, BP2Ty, (Type *)0);
      d_targetFunc  = (Function *) M.getOrInsertFunction("__builtin_spu_extract_double",
                        Type::getDoubleTy(M.getContext()), BP1Ty, BP2Ty, (Type *)0);
      uc_targetFunc = (Function *) M.getOrInsertFunction("__builtin_spu_extract_uchar",
                        Type::getInt8Ty(M.getContext()),   BP1Ty, BP2Ty, (Type *)0);
      c_targetFunc  = (Function *) M.getOrInsertFunction("__builtin_spu_extract_char",
                        Type::getInt8Ty(M.getContext()),   BP1Ty, BP2Ty, (Type *)0);
      us_targetFunc = (Function *) M.getOrInsertFunction("__builtin_spu_extract_ushort",
                        Type::getInt16Ty(M.getContext()),  BP1Ty, BP2Ty, (Type *)0);
      s_targetFunc  = (Function *) M.getOrInsertFunction("__builtin_spu_extract_short",
                        Type::getInt16Ty(M.getContext()),  BP1Ty, BP2Ty, (Type *)0);
```

```
  ui_targetFunc = (Function *) M.getOrInsertFunction("__builtin_spu_extract_uint",
                    Type::getInt32Ty(M.getContext()),  BP1Ty, BP2Ty, (Type *)0);
  i_targetFunc  = (Function *) M.getOrInsertFunction("__builtin_spu_extract_int",
                    Type::getInt32Ty(M.getContext()),  BP1Ty, BP2Ty, (Type *)0);
  ull_targetFunc= (Function *) M.getOrInsertFunction("__builtin_spu_extract_ulonglong",
                    Type::getInt64Ty(M.getContext()), BP1Ty, BP2Ty, (Type *)0);
  ll_targetFunc = (Function *) M.getOrInsertFunction("__builtin_spu_extract_longlong",
                    Type::getInt64Ty(M.getContext()), BP1Ty, BP2Ty, (Type *)0);

  // return true if the function is modified
  return true;
}

virtual bool doInitialization(Function &F) {

  if (strcmp(F.getName().str().c_str(), "__builtin_spu_extract") == 0)
    targetFunc = &F;
  if (strcmp(F.getName().str().c_str(), "__builtin_spu_extract_float") == 0)
    f_targetFunc = &F;
  if (strcmp(F.getName().str().c_str(), "__builtin_spu_extract_double") == 0)
    d_targetFunc = &F;
  if (strcmp(F.getName().str().c_str(), "__builtin_spu_extract_uchar") == 0)
    uc_targetFunc = &F;
  if (strcmp(F.getName().str().c_str(), "__builtin_spu_extract_char") == 0)
    c_targetFunc = &F;
  if (strcmp(F.getName().str().c_str(), "__builtin_spu_extract_ushort") == 0)
    us_targetFunc = &F;
  if (strcmp(F.getName().str().c_str(), "__builtin_spu_extract_short") == 0)
    s_targetFunc = &F;
  if (strcmp(F.getName().str().c_str(), "__builtin_spu_extract_uint") == 0)
    ui_targetFunc = &F;
  if (strcmp(F.getName().str().c_str(), "__builtin_spu_extract_int") == 0)
    i_targetFunc = &F;
  if (strcmp(F.getName().str().c_str(), "__builtin_spu_extract_ulonglong") == 0)
    ull_targetFunc = &F;
  if (strcmp(F.getName().str().c_str(), "__builtin_spu_extract_longlong") == 0)
    ll_targetFunc = &F;

  // return true if the function is modified
  return false;
}
```

```
virtual bool runOnFunction(Function &F) {

  for (Function::iterator b = F.begin(), be = F.end(); b != be; ++b) {
    bool Changed = false;
    BasicBlock::InstListType &BBIL = b->getInstList();
    for (BasicBlock::iterator i = b->begin(), ie = b->end(); i != ie; ++i) {
      if (CallInst* callInst = dyn_cast<CallInst>(&*i)) {
        // We know we've encountered a call instruction, so we
        // need to determine if it's a call to the
        // function pointed to by spu_extract or not.
        if (callInst->getCalledFunction() == targetFunc) {
          if(callInst->getArgOperand(0)->getType()->isVectorTy()) {
            if(callInst->getArgOperand(0)->getType()->isFPOrFPVectorTy()) {
              if(callInst->getArgOperand(0)->getType()
                          ->getScalarType()->isFloatTy()) {
                // insert a call to a f_targetFunction
                CallInst *new_call=CallInst::Create((Constant *)f_targetFunc,"", i);
                new_call->setTailCall();
                callInst->replaceAllUsesWith(new_call);
                // Delete old instruction
                i = --BBIL.erase(i);
                Changed = true;
                ++ExtractCounter;
              } // end if isFloatTy() / float
              if(callInst->getArgOperand(0)->getType()
                          ->getScalarType()->isDoubleTy()) {
                // insert a call to a d_targetFunction
                CallInst *new_call=CallInst::Create((Constant *)d_targetFunc,"",i);
                new_call->setTailCall();
                callInst->replaceAllUsesWith(new_call);
                // Delete old instruction
                i = --BBIL.erase(i);
                Changed = true;
                ++ExtractCounter;
              } // end if isDoubleTy() / double
              if(callInst->getArgOperand(0)->getType()
                          ->getScalarType()->isIntegerTy(8)) {
                // insert a call to a c_targetFunction
                CallInst *new_call=CallInst::Create((Constant *)c_targetFunc,"", i);
                new_call->setTailCall();
```

```
            callInst->replaceAllUsesWith(new_call);
            // Delete old instruction
            i = --BBIL.erase(i);
            Changed = true;
            ++ExtractCounter;
          } // end if isIntegerTy(8) / char
          if(callInst->getArgOperand(0)->getType()
                      ->getScalarType()->isIntegerTy(16)) {
            // insert a call to a s_targetFunction
            CallInst *new_call=CallInst::Create((Constant *)s_targetFunc,"", i);
            new_call->setTailCall();
            callInst->replaceAllUsesWith(new_call);
            // Delete old instruction
            i = --BBIL.erase(i);
            Changed = true;
            ++ExtractCounter;
          } // end if isIntegerTy(16) / short
          if(callInst->getArgOperand(0)->getType()
                      ->getScalarType()->isIntegerTy(32)) {
            // insert a call to a i_targetFunction
            CallInst *new_call=CallInst::Create((Constant *)i_targetFunc,"", i);
            new_call->setTailCall();
            callInst->replaceAllUsesWith(new_call);
            // Delete old instruction
            i = --BBIL.erase(i);
            Changed = true;
            ++ExtractCounter;
          } // end if isIntegerTy(32) / int
          if(callInst->getArgOperand(0)->getType()
                      ->getScalarType()->isIntegerTy(64)) {
            // insert a call to a ll_targetFunction
            CallInst *new_call=CallInst::Create((Constant *)ll_targetFunc,"", i);
            new_call->setTailCall();
            callInst->replaceAllUsesWith(new_call);
            // Delete old instruction
            i = --BBIL.erase(i);
            Changed = true;
            ++ExtractCounter;
          } // end if isIntegerTy(64) / long long

        ++callCounter;
```

```
            } // end if
          } // end if
        } // end for each BasicBlock::iterator i
      } // end for each Function::iterator b

      // return true if the function is modified
      if (Changed)
        return true;
      return false;
    }
  };  // end of struct Extract
}   // end of anonymous namespace


char Extract::ID = 0;
static RegisterPass<Extract> X("extract", "Extract Pass (needed for 'spu_extract' usage)");


FunctionPass *llvm::createExtractPass(bool extractFlag) {
  return new Extract(extractFlag);
}
```

Appendice E

Listato del programma di halftoning

Di seguito il codice completo del lisitato del programma di halftoning diviso in lato *PPE* e lato *SPE*.
Codice del prgramma di halftoning lato *SPE*:

```
#include <stdio.h>
#include <spu_intrinsics.h>
#include <spu_mfcio.h>

float thresh_spe[1] __attribute__((aligned(16)));
float one_spe[1]    __attribute__((aligned(16)));
float zero_spe[1]   __attribute__((aligned(16)));
float rows_spe[1]   __attribute__((aligned(16)));
float cols_spe[1]   __attribute__((aligned(16)));

typedef struct {
    unsigned long long  ea_thresh;
    unsigned long long  ea_one;
    unsigned long long  ea_zero;
    unsigned long long  ea_rows;
    unsigned long long  ea_cols;
    unsigned long long  ea_in_image;
    unsigned long long  ea_out_image;
    unsigned int        size;
    int                 pad[ 1];
} abs_params_t;
```

```
abs_params_t abs_params __attribute__((aligned(16)));

int main(unsigned long long spe, unsigned long long argp, unsigned long long envp)
{
   int   tag = 1;
   float val_t, val_r, val_c;
   float **eg, **ep;
   float c[2][3], sum_p, t;
   int   i, j, xx, yy;

   c[0][0]  = 0.0;
   c[0][1]  = 0.2;
   c[0][2]  = 0.0;
   c[1][0]  = 0.6;
   c[1][1]  = 0.1;
   c[1][2]  = 0.1;

   __vector float *vthreshold = (__vector float *) thresh_spe;
   __vector float *vone       = (__vector float *) one_spe;
   __vector float *vzero      = (__vector float *) zero_spe;
   __vector float *vrows      = (__vector float *) rows_spe;
   __vector float *vcols      = (__vector float *) cols_spe;

   // Input parameter parm is a pointer to the context.
   // Fetch the context, waiting for it to omplete.
   spu_mfcdma64(&abs_params, mfc_ea2h(argp), mfc_ea2l(argp),
                sizeof(abs_params_t), tag, MFC_GET_CMD);
   spu_writech(MFC_WrTagMask, 1 << tag);
   spu_mfcstat(MFC_TAG_UPDATE_ALL);

   if(abs_params.size % 16 != 0)
      // the array size must have size multiple of 16
   // Fetch the data. Wait for DMA to complete before performing computation
   spu_mfcdma64(vthreshold, mfc_ea2h(abs_params.ea_thresh), mfc_ea2l(abs_params.ea_thresh),
                abs_params.size, tag, MFC_GET_CMD);
   spu_writech(MFC_WrTagMask, 1 << tag);

   spu_mfcdma64(vone, mfc_ea2h(abs_params.ea_one), mfc_ea2l(abs_params.ea_one),
                abs_params.size, tag, MFC_GET_CMD);
   spu_writech(MFC_WrTagMask, 1 << tag);
```

```
spu_mfcdma64(vzero, mfc_ea2h(abs_params.ea_zero), mfc_ea2l(abs_params.ea_zero),
             abs_params.size, tag, MFC_GET_CMD);
spu_writech(MFC_WrTagMask, 1 << tag);

spu_mfcdma64(vrows, mfc_ea2h(abs_params.ea_rows), mfc_ea2l(abs_params.ea_rows),
             abs_params.size, tag, MFC_GET_CMD);
spu_writech(MFC_WrTagMask, 1 << tag) ;

spu_mfcdma64(vcols, mfc_ea2h(abs_params.ea_cols), mfc_ea2l(abs_params.ea_cols),
             abs_params.size, tag, MFC_GET_CMD);
spu_writech(MFC_WrTagMask, 1 << tag) ;

spu_mfcstat(MFC_TAG_UPDATE_ALL);

val_r = spu_extract(*vrows, 1);
val_c = spu_extract(*vcols, 1);

short in_image_spe[(int)val_r][(int)val_c]  __attribute__((aligned(16)));
short out_image_spe[(int)val_r][(int)val_c] __attribute__((aligned(16)));
__vector short *vin_image  = (__vector short *) in_image_spe;
__vector short *vout_image = (__vector short *) out_image_spe;

spu_mfcdma64(vin_image, mfc_ea2h(abs_params.ea_in_image), mfc_ea2l(abs_params.ea_in_image
             abs_params.size, tag, MFC_GET_CMD);
spu_writech(MFC_WrTagMask, 1 << tag) ;

spu_mfcdma64(vout_image, mfc_ea2h(abs_params.ea_out_image), mfc_ea2l(abs_params.ea_out_im
             abs_params.size, tag, MFC_GET_CMD);
spu_writech(MFC_WrTagMask, 1 << tag) ;

spu_mfcstat(MFC_TAG_UPDATE_ALL);

// Half Toning Computation

  /*  Calculate the total propogated error
  *   at location(m,n) due to prior assignment.
  *
  *   Go through the input image. If the output
  *   should be one then display that pixel as such.
  *   If the output should be zero then display it that way.
  *
```

```
    *   Also set the pixels in the input image array
    *   to 1's and 0's in case the print option was chosen.
    */

//__vector float vval = (__vector float) { 0 };
//     float *val = (float *) &vval ;


  val_t = spu_extract(*vthreshold, 1);

  i = val_r * sizeof(float *);
  i = val_c * sizeof(float );

  eg = malloc( val_r * sizeof(float *) );
  for(i=0; i<val_r; i++){
     eg[i] = malloc( val_c * sizeof(float ) );
     if(eg[i] == '\0'){
        printf("\n\tmalloc of eg[%d] failed", i);
     } /* ends if */
  } /* ends loop over i */
  ep = malloc( val_r * sizeof(float *) );
  for(i=0; i<val_r; i++){
     ep[i] = malloc( val_c * sizeof(float ) );
     if(ep[i] == '\0'){
        printf("\n\tmalloc of ep[%d] failed", i);
     } /* ends if */
  }  /* ends loop over i */

  for(i=0; i<val_r; i++){
     for(j=0; j<val_c; j++){
        eg[i][j] = 0.0;
        ep[i][j] = 0.0;
     }
  }

  for(m=0; m<val_r; m++){
     for(n=0; n<val_c; n++){
        sum_p = 0.0;
        for(i=0; i<2; i++){
           for(j=0; j<3; j++){
              xx = m-i+1;
```

```
                yy = n-j+1;
                if(xx < 0)     xx = 0;
                if(xx >= val_r) xx = val_r-1;
                if(yy < 0)     yy = 0;
                if(yy >= val_c) yy = val_c-1;
                sum_p = sum_p + c[i][j] * eg[xx][yy];
             }  /* ends loop over j */
          }     /* ends loop over i */
          ep[m][n] = sum_p;
        //t = ep[m][n] + in_image_spe[m][n] ;
          t = ep[m][n] + vin_image[m][n];
        //  Here set the point [m][n]=one
          if(t > val_t){
             eg[m][n] = t - val_t * 2;
//  --->>   out_image[m][n] = *vone;
           //out_image_spe[m][n] = one_spe[0];
          } /* ends if t > threshold */
        //  Here set the point [m][n]=zero
          else{ /* t <= threshold */
             eg[m][n]        = t;
//  --->>   out_image[m][n] = *vzero;
           //out_image_spe[m][n] = zero_spe[0];
          } /* ends else t <= threshold   */
       }   /* ends loop over n columns */
   }          /* ends loop over m rows */

   for(i=0; i<val_r; i++){
      free(eg[i]);
      free(ep[i]);
   }
/* send computed sum back to master */
   spu_mfcdma64(vout_image, mfc_ea2h(abs_params.ea_out_image), mfc_ea2l(abs_params.ea_out_im
                  sizeof(abs_params.size), tag, MFC_PUT_CMD) ;
   spu_writech( MFC_WrTagMask, 1 << tag) ;
   // Wait DMA to complete before terminating SPE thread
   spu_mfcstat(MFC_TAG_UPDATE_ALL);

   return 0;
}
```

Bibliografia

[1] A. Fausto, *Programmare il Cell Broadband Engine della Playstation 3*, di recente pubblicazione in Proc. of CSDUML03, Oct. 2003

[2] International Business Machines Corporation (IBM), Sony Computer Entertainment Inc., Toshiba Corporation, *Cell Broadband Engine - Programming Handbook including the PowerXCell 8i Processor (Version 1.12)*, 3 Apr. 2009

[3] llvm.org, *http://llvm.org/*

[4] it.wikipedia.org, *http://it.wikipedia.org/wiki/LLVM*

[5] en.wikipedia.org, *http://en.wikipedia.org/wiki/Low_ Level_ Virtual_ Machine*

[6] llvm.org, *http://llvm.org/Features.html*

[7] clang.llvm.org, *http://clang.llvm.org/*

[8] en.wikipedia.org, *http://en.wikipedia.org/wiki/Clang*

[9] clang.llvm.org, *http://clang.llvm.org/docs/features.html*

[10] clang.llvm.org, *http://clang.llvm.org/docs/UsersManual.html*

[11] llvm.org, *http://llvm.org/WritingAnLLVMPass.html*

[12] llvm.org, *http://llvm.org/ProgrammersManual.html*

[13] llvm.org, *http://llvm.org/docs/LangRef.html*

[14] llvm.org, *http://llvm.org/Doxygen*

[15] llvm.org, *http://llvm.org/docs/Doxygen*

[16] gcc.gnu.org,
http://gcc.gnu.org/onlinedocs/gcc4.4.2/gcc/C-Extensions.html

[17] gcc.gnu.org,
http://gcc.gnu.org/onlinedocs/gcc4.4.2/gcc/Function-Names.html

[18] gcc.gnu.org,
http://gcc.gnu.org/onlinedocs/gcc4.4.2/gcc/Extended-Asm.html

[19] gcc.gnu.org,
http://gcc.gnu.org/onlinedocs/gcc4.4.2/gcc/Modifiers.html

[20] lists.cs.uiuc.edu, *http://lists.cs.uiuc.edu/pipermail/llvmdev*

[21] old.nabble.com, *http://old.nabble.com*

[22] stackoverflow.com, *http://stackoverflow.com*

[23] en.wikipedia.org, *http://en.wikipedia.org/wiki/Image_analysis*

[24] AA.VV., *The C/C++ Users Journal*, R & D Pubblications, Miller & Freeman, Inc, CMP Media, Inc, 1990-1998

[25] Dwayne Phillips, *Image Processing in C - Second Edition*, R & D Pubblications, Apr. 2000

[26] John A. Saghri, Hsieh S. Hou, Andrew F. Tescher *Personal Computer Based Image Processing with Halftoning*, Optical Engineering, vol. 25, no. 3, Mar. 1986

[27] John Milano, *Compressed Image File Formats*, Addison - Wesley Professional, 1999

[28] en.wikipedia.org, *http://en.wikipedia.org/wiki/Tagged_Image_File_Format*

[29] en.wikipedia.org, *http://en.wikipedia.org/wiki/Windows_bitmap*

[30] en.wikipedia.org, *http://en.wikipedia.org/wiki/Halftone*