Università Degli Studi Di Roma Tor Vergata

FACOLTÀ DI INGEGNERIA

Corso di Laurea Magistrale in Ingegneria Informatica

Trasferimenti di Dati tra Memoria Utente e Kernel tramite Circuiti DMA Dedicati

Relatore:
Prof. Marco Cesati

Laureando:
Antonio Papa

Correlatore:
Ing. Emiliano Betti

Anno Accademico 2012-2013

Ai miei genitori e
a mia nonna Giuseppa

Indice

Elenco delle figure

Elenco delle tabelle

Introduzione

Per molti sistemi operativi e device driver il trasferimento di dati tra memoria utente e kernel è un'operazione critica che, in determinate situazioni, può peggiorare le prestazioni dell'intero sistema. Così utilizzare un motore DMA, per effettuare le copie tra memoria kernel e utente può, e nel seguito della trattazione sarà dimostrato, apportare benefici all'intero sistema.

Il lavoro che sarà descritto in questo testo si potrebbe definire brevemente come la realizzazione di un'interfaccia generica per le operazioni di copia tra memoria kernel e utente effettuate tramite motore DMA. A tal fine è stata realizzata un'estensione del kernel che aggiunge a quest'ultimo le funzionalità e le informazioni necessarie a gestire un motore DMA che effettua trasferimenti *Memory-to-Memory*. In modo particolare il kernel scelto è quello del sistema operativo Linux[1] nella sua versione 3.9.2, mentre l'hardware dedicato alle copie è un *5000 Series Chipset DMA Engine*.
Si vedrà in seguito che l'estensione sviluppata non si limita a fornire un'interfaccia base per i trasferimenti DMA, ma mette a disposizione un insieme di politiche utilizzate per la gestione dei canali DMA. Una di queste politiche può essere utilizzata anche per gestire i trasferimenti di applicazioni real-time, infatti consente di assegnare in modo esclusivo il bus del motore DMA, per un certo intervallo di tempo, ai canali che gestiscono task ad alta priorità.

La presentazione del lavoro verrà sviluppata come segue: nel *capitolo 1* verrà inizialmente introdotto il sistema di I/O di un calcolatore ed i vari metodi di gestione delle periferiche. Successivamente si procederà alla descrizione dettagliata del meccanismo di accesso diretto alla memoria (DMA).
Nel *capitolo 2* verrà descritto motore DMA dedicato ai trasferimenti Memory to Memory e verrà illustrato il Sottosistema DMA, implementato nel kernel Linux, che espone un'interfaccia generica per le operazioni di copia. A

[1] Linux è un marchio registrato da Linus Torvalds.

seguire, nel *capitolo 3* verrà effettuata una panoramica sui principali lavori riguardanti sistemi che, tramite hardware dedicato, eseguono trasferimenti di dati in memoria.
Dopo questo sguardo più generale, nel *capitolo 4* si passerà alla descrizione specifica dell'estensione DMA-CFTU da me sviluppata, partendo dal supporto alle transazioni e trasferimenti DMA e procedendo con l'analisi delle varie politiche di gestione dei canali. Per analizzare le prestazioni delle copie in memoria, effettuate tramite l'estensione DMA-CFTU, sono stati sviluppati una serie di tool descritti nel *capitolo 5*. Infine nel *capitolo 6* verranno presentati e discussi i risultati ottenuti tramite i tool sopra menzionati.

Capitolo 1

Direct Memory Access

Nelle prime architetture PC, la CPU svolgeva il ruolo di unico *bus master* del sistema, ossia, era l'unico device hardware che gestiva il bus (dati-indirizzi) per recuperare e memorizzare i dati nelle locazioni di RAM. Con l'avvento delle più moderne architetture bus come la PCI, ogni periferica può avere il ruolo di *bus master*, se provvista della giusta circuiteria. Così al giorno d'oggi tutti i PC includono ausiliari circuiti DMA, che permettono di trasferire dati tra la RAM e l'I/O device [1].

In questo capitolo, al fine di fornire al lettore un quadro esaustivo sull'ambiente in cui opera un motore DMA, verrà inizialmente descritto il sistema di I/O di un calcolatore ed i vari metodi di gestione delle periferiche. Successivamente si procederà alla descrizione dettagliata del meccanismo di accesso diretto alla memoria (DMA).

1.1 Il Sistema di I/O

Il Sistema d'ingresso/uscita detto più semplicemente I/O (Input/Output) permette la comunicazione del calcolatore con il mondo esterno. Fanno parte del sistema i dispositivi (Unità di I/O):

- per la comunicazione uomo/macchina (video, terminali, stampanti, ecc);
- per la memorizzazione permanente delle informazioni (unità a disco, nastri magnetici, ecc);
- per la rete (scheda di rete, radio, linea seriale, ecc).

Tutte le Unità di I/O sono collegate al bus di sistema (Fig 1.1) mediante dispositivi detti *Interfacce di I/O* (o anche *Controllori di I/O*) [3, 4].

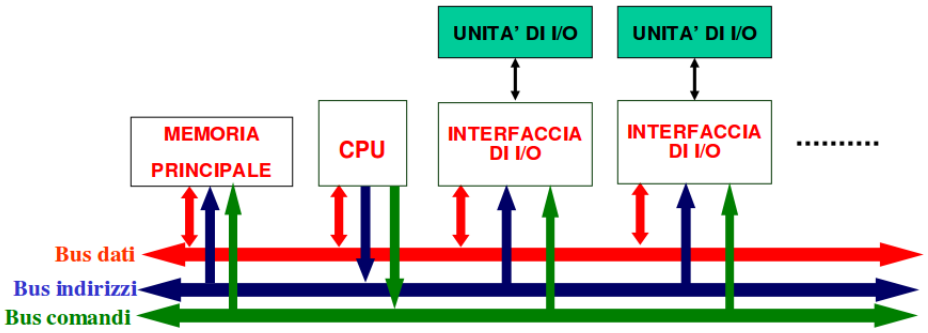

Figura 1.1: Interfacce di I/O

Tipicamente ogni periferica possiede un'interfaccia I/O e dei registri di lettura e scrittura mediante i quali è possibile controllarla. L'interfaccia solitamente espone i registri in maniera tale che possano essere acceduti tramite indirizzi, ossia identificatori numerici che consentono di selezionare senza ambiguità la periferica come sorgente o destinazione di un trasferimento di dati sul bus.

Si definisce *spazio di indirizzamento di I/O* l'insieme dei valori numerici che possono essere potenzialmente utilizzati per selezionare una periferica collegata ad un bus [5, 6]. Esistono due tipi di indirizzamento:

- *indirizzamento isolato* (*Porte di I/O*): lo spazio di indirizzamento di I/O è separato ed indipendente dallo spazio di indirizzamento della memoria centrale. Il processore è dotato di istruzioni macchina specifiche per trasferire dati con le periferiche.

- *indirizzamento mappato in memoria* (*Memory Mapped I/O*): lo spazio di indirizzamento di I/O è mappato nello spazio di indirizzamento della memoria centrale, i registri di controllo del dispositivo vengono fatti corrispondere ad indirizzi mappati. Quindi il processore può utilizzare le stesse istruzioni macchina sia per accedere alla memoria centrale che per trasferire dati con le periferiche.

L'utilizzo del Memory Mapped I/O è tipicamente preferito a quello delle Porte I/O perché:

- l'uso delle specifiche istruzioni di CPU limita le architetture utilizzabili, oppure obbliga ad operazioni di adattamento che possono produrre un degrado delle prestazioni.

- l'accesso alla memoria è in genere più efficiente ed in alcuni casi può essere usato il DMA (Direct Memory Access),
- il compilatore riesce ad ottimizzare meglio il codice [7].

Indipendentemente dalla tecnica di indirizzamento scelta, ogni dispositivo di I/O è sempre provvisto di registri per fornire il proprio stato nonché ogni informazione di controllo che risulti necessaria.

I dispositivi I/O spesso devono gestire grandi quantità di dati a velocità elevate. Ci sono principalmente 3 meccanismi per l'acquisizione di dati, che si differenziano per il diverso livello di coinvolgimento della CPU nella gestione dei dispositivi di I/O:

1. **I/O Programmato - Polling**
2. **Interrupt driven I/O**
3. **Accesso diretto alla memoria** (DMA, *Direct Memory Access*)

La Tabella 1.1 suddivide i seguenti meccanismi rispetto all'uso dell'interrupt e al coinvolgimento della CPU nei trasferimenti periferica-memoria:

	Senza interrupt	Con interrupt
Trasferimento I/O tramite processore	I/O programmato	Interrupt
Trasferimento I/O tramite DMA		DMA

Tabella 1.1: Metodi di gestione delle periferiche

1.1.1 I/O Programmato - Polling

Nell'I/O Programmato la gestione dei dispositivi è totalmente demandata alla CPU. Ogni dato viene prima trasferito dal buffer della periferica ad un registro interno della CPU, e poi immagazzinato in memoria (o viceversa). Lo spostamento di ciascun dato implica l'esecuzione di almeno un'istruzione da parte della CPU. Quindi il Polling è basato sul controllo periodico da parte del processore di un flag, locato in un registro di stato della periferica,

che segnala la disponibilità di un dato o la possibilità di eseguire una nuova operazione.

Per esempio immaginiamo di gestire una tastiera tramite Polling, nello specifico vediamo come richiedere il codice del prossimo tasto premuto sulla tastiera:

```
wait:
...                 # altre operazioni
inb    $0x64        # %al <- [registro di stato]
andb   $0b10,%al    # controlla il flag IBF
jz     wait         # salta se IBF e' zero
inb    $0x60        # %al <- [registro dati]
```

Il flag IBF (Input Buffer Full) in seconda posizione del registro di stato vale 1 se un nuovo codice tasto è disponibile nel registro dati; la periferica pone a 0 il flag automaticamente dopo una lettura del registro dati [6].
Quindi è possibile schematizzare il meccanismo del Polling in questo modo:

1. La CPU richiede un'operazione di I/O
2. Il controller di I/O effettua l'operazione di I/O
3. Il controller di I/O attiva un bit nello "status register"
    - Il controller di I/O non informa direttamente la CPU
    - Il controller di I/O non interrompe la CPU
4. La CPU guarda periodicamente il bit nello "status register"

Ne consegue che la modalità di gestione delle periferiche tramite Polling è semplice da realizzare in hardware, ma costosa in termini di organizzazione del sistema operativo e di efficienza complessiva. Infatti ogni dispositivo è vincolato alla CPU per essere servito, quindi si hanno suddetti limiti:

- il tempo, che nel caso peggiore il dispositivo deve attendere, prima che la CPU esegua il trasferimento richiesto può essere alto, e dipende da quanti dispositivi di I/O sono connessi;
- tutti i dati devono passare attraverso la CPU, e non esiste connessione diretta tra dispositivo e memoria;
- la CPU dedica una parte del suo tempo ad eseguire banali operazioni di test e trasferimento dati.

Come è possibile notare il meccanismo di Polling, in particolare nel caso della gestione del disco, risulta inadeguato perché troppo costoso. Prestazioni migliori si potrebbero ottenere se una periferica potesse *informare* il processore che una certa condizione si è verificata.

1.1.2 Interrupt driven I/O

Si definisce *interruzione* (*interrupt*) un segnale inviato da una periferica al processore per indicare il verificarsi di un evento (ad esempio, il completamento di una operazione o la disponibilità di nuove informazioni). Le interruzioni possono essere generate dal processore stesso ed in questo caso sono di tipo *sincrono* (negli x86 vengono chiamate *eccezioni*), oppure possono partire da dispositivi esterni, risultando quindi *asincrone*. La differenza principale è che le prime, essendo generate internamente dal processore, si originano sempre al termine di un'istruzione, mentre quelle asincrone sono dette tali proprio perché, venendo dall'esterno non è in nessun modo prevedibile l'istante in cui possono arrivare [7].
Inoltre le interruzioni possono essere *mascherabili*, ovvero la CPU può continuare a lavorare come se non sia stato ancora ricevuto il segnale di interruzione. In alcuni processori (8088 compreso) esiste un pin per le interruzioni *non mascherabili* (NMI) utilizzato ad esempio dalla CPU per avviare un programma che permetta di salvare lo stato della macchina, se le periferiche segnalano un calo della tensione di alimentazione [9].

Quando una periferica deve inviare dati alla CPU o vuole ricevere dati da essa effettua una richiesta di interruzione. Il segnale inviato dal dispositivo di I/O (IRQ, *Interrupt Request*) prende il nome di *richiesta di interruzione*, mentre la sequenza di istruzioni che il processore esegue in seguito ad un'interruzione viene detta *routine di servizio*. Tutte le IRQ generate dalle periferiche di I/O originano interruzioni hardware asincrone mascherabili.

I componenti fondamentali del meccanismo di interruzione sono:

- Una linea di controllo del bus dedicata alla trasmissione di segnali di interruzione originati dalle periferiche e destinati al processore: è chiamata *linea di richiesta di interruzione*, detta anche *linea di IRQ*.

- Una linea di controllo del bus dedicata alla trasmissione di segnali di *conferma dell'interruzione* (o *linea di ACK*, Acknowledge) originati dal processore e diretti verso le periferiche.
- Circuiti all'interno dell'interfaccia della periferica per abilitare la generazione di segnali di interruzione in predeterminati casi.
- Circuiti all'interno del processore per interrompere l'esecuzione del programma in esecuzione e passare all'esecuzione di una procedura predefinita ogni volta che la linea di richiesta di interruzione viene asserita.

A differenza del Polling, con il meccanismo delle interruzioni il processore non ha più l'obbligo di esaminare in continuazione lo stato delle periferiche. Infatti la CPU deve solo scrive opportuni comandi sull'interfaccia della periferica per programmare una determinata operazione e abilitare la generazione di una interruzione alla sua conclusione. Così il processore si dedica ad eseguire altri programmi, mentre la periferica svolge l'operazione richiesta.
Quando la periferica termina l'operazione, asserisce la linea di IRQ (in questa fase si dice che la richiesta di interruzione è *pending*), così i circuiti del processore, dedicati al controllo della linea di IRQ, rilevano il cambio di stato e interrompono l'esecuzione del programma in esecuzione (salvando le informazioni necessarie per riprendere la sua esecuzione in futuro). La CPU passa ad eseguire una procedura predefinita, chiamata *gestore dell'interruzione* (*interrupt handler*), il cui primo compito è asserire la linea di ACK dell'interruzione. L'interfaccia della periferica de-asserisce la linea di IRQ non appena verifica che la linea di ACK è stata asserita. Il processore passa ad eseguire una procedura specifica per la periferica, chiamata *Interrupt Service Routine* (ISR), che interroga l'interfaccia della periferica e svolge le azioni opportune (ad esempio, programma l'operazione successiva). Al termine dell'ISR il controllo torna all'interrupt handler, che esegue una istruzione speciale "ritorno da interruzione" (ad esempio, *iret* in IA-32) ed il processore ritorna ad eseguire il programma interrotto.

I calcolatori moderni utilizzano un circuito apposito chiamato *controllore di interruzione* per gestire le interruzioni e arbitrare le loro diverse priorità. Il controllore di interruzione (*interrupt controller*) è spesso realizzato da un

chip esterno al processore: le linee di IRQ delle periferiche sono connesse al controllore, e non al processore. Nei calcolatori recenti con architettura x86 esistono diversi controllori di interruzione:

- PIC (*Programmable Interrupt Controller*): è un dispositivo hardware che consente di gestire interruzioni vettorizzate con priorità, per conto di un processore. Uno dei PIC più conosciuti è l'Intel 8259 che riesce a gestire fino ad 8 sorgenti di interruzioni e può essere utilizzato in cascata, fino ad un limite di 8 dispositivi, per gestire quindi un numero massimo di 64 sorgenti di interruzione. Prodotto dall'Intel Corporation, riveste un'importanza storica poiché venne utilizzato dal 1980 in tutti i PC XT e AT IBM e compatibili. Attualmente è fuori produzione [9] [10] [11].
- Local APIC e I/O APIC: per i moderni sistemi multiprocessore simmetrici (SMP) i PIC non sono adeguati, è stato quindi introdotto un nuovo sistema per la gestione degli interrupt: l'architettura APIC (*Advanced Programmable Interrupt Controller*). Suddetta architettura consta di due componenti: il Local APIC (LAPIC) e l'I/O APIC. Il LAPIC è integrato in ogni CPU di sistema, mentre l'I/O APIC risiede nei chip della scheda madre, riceve le linee di IRQ provenienti dalle periferiche e genera messaggi di interruzione sul bus [12].

Il meccanismo Interrupt driven I/O riesce a gestire periferiche che effettuano dei trasferimenti sporadici di piccoli blocchi di dati. Quando si deve però gestire periferiche che trasferiscono frequentemente grossi blocchi di dati, allora l'interrupt driven I/O diventa un meccanismo dal costo proibitivo.

1.2 Direct Memory Access Controller

Con la tecnica di interrupt-driven I/O, il processore spreca parte del suo tempo a gestire le interruzioni e trasferire dati tra la memoria e la periferica (ciò è particolarmente oneroso quando si devono trasferire ingenti sequenze di dati). Inoltre se il tempo necessario alla CPU per gestire un'interruzione è maggiore del tempo che intercorre tra due interruzioni, si possono avere perdite di dati. Per evitare l'intervento della CPU nella fase di trasferimento dati, è stata introdotta la tecnica di *Direct Memory Access* (DMA), che

prevede l'utilizzo di un controllore hardware per gestire direttamente le operazioni memoria-periferica [2]. Il dispositivo in questione viene denotato con la sigla DMAC (*Direct Memory Access Controller*).

Ovviamente ogni trasferimento di dati coinvolgerà sempre il bus (o dei bus) a cui sono collegati periferica e memoria centrale. Nei bus di concezione più vecchia o semplice (ad esempio l'ISA), il controllore DMA è un processore ausiliario che svolge solo le operazioni di trasferimento. Mentre nei bus più recenti e sofisticati (ad esempio, PCI), le periferiche possono accedere direttamente alla memoria grazie a circuiti che implementano un controllore DMA dedicato. In altri termini, le periferiche di bus recenti come PCI possono assumere il ruolo di unità master per il controllo del bus. Per questo motivo l'accesso diretto alla memoria è anche conosciuto come *bus mastering*. Nel seguito ci focalizzeremo sui controllori DMA dedicati.

1.2.1 Integrazione nel sistema

Il *DMA Controller* ha il compito di gestire i dati passanti nel bus, permettendo a periferiche che lavorano a velocità diverse di comunicare senza assoggettare la CPU ad un enorme carico di interrupt, che ne interromperebbero continuamente il rispettivo ciclo di elaborazione; inoltre deve essere in grado di negoziare con la CPU l'acquisizione del controllo del bus ed il suo rilascio.

Il funzionamento di un DMA Controller in generale si può dividere in due fasi: la *fase di programmazione* e la *fase di trasferimento*. Queste due fasi non possono avvenire per ovvie ragioni simultaneamente, infatti durante un trasferimento, non è possibile programmare il controllore.

Nella fase di programmazione il Controller viene visto come un normale dispositivo di I/O dotato di registri, nei quali vengono definiti gli indirizzi base, la dimensione del trasferimento da effettuare e le comunicazioni con il processore. In questa fase, il Controller si comporta da *slave* e le operazioni da eseguire vengono impartite dalla CPU. Inoltre il DMA Controller deve anche comunicare con le periferiche coinvolte nel trasferimento dei dati, ad esempio, per assegnare ad ognuna una linea di richiesta di DMA (*Canale DMA*), decidere la politica di arbitraggio per le richieste simultanee e definire il tipo di trasferimento.

Dopo aver definito tutti questi parametri si passa alla fase di trasferimento, nella quale, dopo aver supposto che il DMA Controller sia stato configurato correttamente, inizia il trasferimento dei dati:

- direttamente verso il DMAC: in questo caso il DMAC avrà almeno un registro dati (data register) dove registrare i dati;
- attraverso il processore (attivato su interrupt);
- utilizzando periferiche in grado di trasferire direttamente il dato verso la memoria.

In Figura 1.2 viene mostrato come un DMAC è connesso con gli altri elementi del sistema, per semplicità si è assunto un bus indirizzi di 24 bit ed un bus dati di 16 bit:

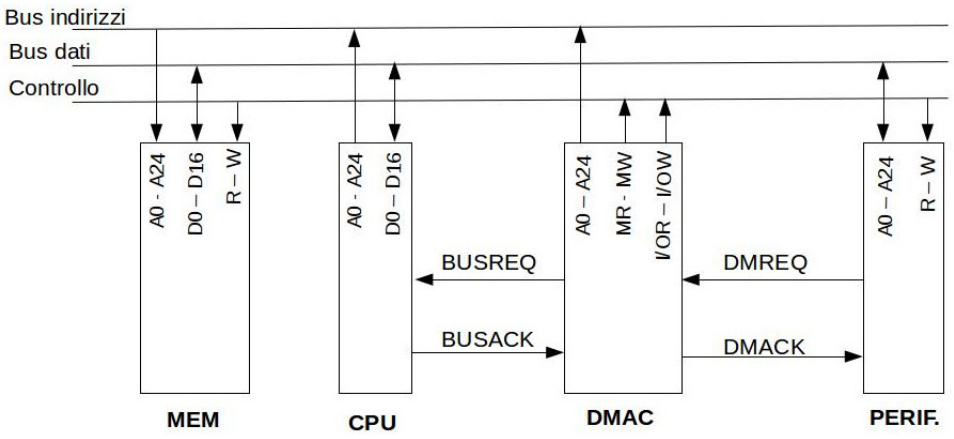

Figura 1.2: Esempio di connessione del DMAC

Il significato dei segnali riportati in Figura 1.2 è il seguente:

- DMREQ: richiesta di trasferimento da parte della periferica;
- BUSREQ: richiesta del bus da parte del DMAC alla CPU;
- BUSACK: risposta favorevole al BUSREQ da parte della CPU al DMAC;
- DMACK: risposta favorevole al DMREQ da parte del DMAC alla periferica;

Generalmente un trasferimento DMA si svolge nel seguente modo [15]:

1. L'interfaccia della periferica asserisce DMREQ per segnalare che ha dati da trasferire.

2. Il DMAC asserisce BUSREQ per richiedere l'uso esclusivo del bus.

3. La CPU risponde asserendo BUSACK. Questo segnale indica che la CPU ha posto in alta impedenza i pin corrispondenti al bus dei dati e degli indirizzi. La CPU resterà in tale stato (detto stato di HOLD) fino a quando BUSREQ resta asserito.

4. Adesso il DMAC può pilotare il bus degli indirizzi ed asserire la coppia di segnali per regolare la direzione del trasferimento memoria-periferica. Infine il controllore invia un segnale di DMACK alla periferica.

5. Terminato il trasferimento BUSREQ è disattivato e la CPU ritorna a controllare il bus disasserendo BUSACK.

1.2.2 Registri e Canali DMA

In generale, un controllore di DMA è in grado di programmare più trasferimenti alla volta, in modo da gestire contemporaneamente più periferiche, ad ognuna delle quali viene assegnato un diverso *canale DMA* presente nel controllore. Si definisce *canale DMA* l'insieme di registri e circuiti di controllo dedicati ad un singolo trasferimento, di norma ogni canale ha la seguente interfaccia:

- Un registro per la posizione dei dati da trasferire sulla periferica (può essere assente).

- Un registro per l'indirizzo dei dati in memoria centrale.

- Un registro per la quantità di dati da trasferire.

- Un registro di comandi (ad esempio: per indicare la direzione del trasferimento e per avviarla).

- Un registro di stato, spesso unificato con il registro di comandi (ad esempio: un flag per indicare se l'operazione di trasferimento è terminata).

Per meglio chiarire il concetto di canale DMA analizziamo la struttura interna di un generico DMAC PIC24F [1] (Figura 1.3), che si presta egregiamente allo scopo [16] [17].

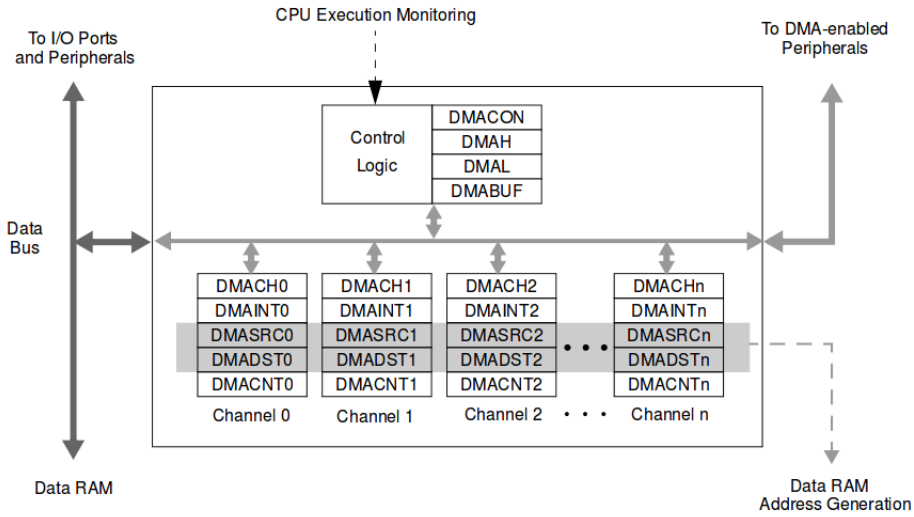

Figura 1.3: Organizzazione dei canali DMA nel PIC24F DMA Controller [16]

Il DMA Controller PIC24F è composto da un certo numero di canale DMA[2], ognuno dei quali può essere programmato in modo indipendente per trasferire dati tra le diverse regioni di memoria. Inoltre è possibile far lavorare insieme più canali DMA, al fine di effettuare il trasferimento di dati più complessi senza l'intervento della CPU. Come è possibile notare ci sono due tipologia di registi: quelli generali usati dal modulo *control logic* del DMAC e quelli locali replicati su ogni canale DMA. I registri globali sono 4 (uno per il controllo e tre per la gestione buffer/indirizzi):

- **DMACON**: il *DMA Engine Control Register* permette di controllare il funzionamento complessivo del motore DMA e viene usato per determinare la priorità di scansione dei canali.
- **DMAH** e **DMAL**: *High* e *Low Address Limit Registers* delimitano lo spazio di indirizzi a 16 bit in cui è possibile effettuare le operazioni DMA.
- **DMABUF**: *DMA Data Buffer* è un registro di 16 bit per memorizzare i dati trasferiti in un'operazione DMA.

[1] Controller DMA utilizzato nella PIC24F Family
[2] Il numero di canali dipende dal dispositivo specifico

In ogni canale DMA ci sono 5 registri (due per il controllo e tre per la gestione buffer/indirizzi):

- **DMACH$_x$**: il *Control Register* è usato per abilitare il DMA $Channel_x$ e configurarne la modalità di trasferimento dei dati.
- **DMAINT$_x$**: l'*Interrupt Control Register* del DMA è usato per associare una periferica, che richiede un trasferimento, al DMA $Channel_x$ e per gestire gli interrupt relativi al suddetto canale.
- **DMASRC$_x$**: il *Data Source Address Pointer* del DMA $Channel_x$ è usato per memorizzare l'indirizzo sorgente coinvolto nel trasferimento.
- **DMADST$_x$**: il *Data Destination Source* del DMA $Channel_x$ è usato per memorizzare l'indirizzo destinazione coinvolto nel trasferimento.
- **DMACNT$_x$**: *Transaction Counter* del DMA $Channel_x$ è usato per memorizzare il numero di dati (parole o byte, a seconda della modalità) da trasferire.

I trasferimenti DMA, fin qui descritti, hanno coinvolto sempre un dispositivo di I/O e una regione di memoria, ma non è inusuale disporre di DMAC che permettano di effettuare trasferimenti *Memory to Memory* e *I/O-device to I/O-device* [18]. In particolare i trasferimenti *Memory to Memory* sono di preziosa utilità per gestire operazioni che lavorano su moli di dati considerevoli, infatti il passaggio di tutti i dati attraverso i registri della CPU renderebbe lunghissimo e oneroso il trasferimento.

1.3 Trasferimento Dati

Le tecniche di trasferimento utilizzate nei diversi DMA Controller possono essere di tipo: *fetch-and-deposit* o *flyby*. Nei trasferimenti di tipo *fetch-and-deposit* , il DMAC, dopo aver assunto il controllo del bus, legge il dato dalla periferica/memoria lo memorizza temporaneamente in un suo registro interno per poi scriverlo, in un secondo tempo, sul dispositivo desiderato. I trasferimenti di tipo *flyby* invece non prevedono nessuna memorizzazione temporanea dei dati.
Indipendentemente dalla tecnica utilizzata, i dati possono essere trasferiti con le seguenti modalità: *burst transfer*, *cycle stealing* e *transparent/hidden*.

1.3.1 Tecniche di Trasferimento

La tecnica di trasferimento di tipo *flyby* [8] [14] (detta anche *single-cycle* o *single-address*) è sicuramente la più prestante in termini di cicli di clock, infatti è necessaria una sola operazione di bus per effettuare il trasferimento. In un'operazione flyby il device che richiede un trasferimento asserisce la linea di richiesta DMA relativa al canale del DMAC a lui associato. Il DMA Controller , ricevuto il segnale, ottiene il controllo del bus di sistema dalla CPU e lo prepara per il trasferimento. Contemporaneamente il DMAC invia un segnale di Acknowledge al device. Quest'ultimo ricevuto il segnale si appresta ad inserire o prelevare il dato sul bus di sistema. In pratica un trasferimento flyby è paragonabile ad un ciclo di lettura/scrittura in memoria, in cui il DMAC prepara il bus e il device I/O legge o scrive il dato. Poiché i trasferimenti flyby coinvolgono un singolo ciclo di memoria per effettuare un'operazione DMA, è possibile utilizzare questa tecnica solo per i trasferimenti da memoria ad I/O o da I/O a memoria. In Figura 1.4 è mostrato il *signal protocol* del flyby.

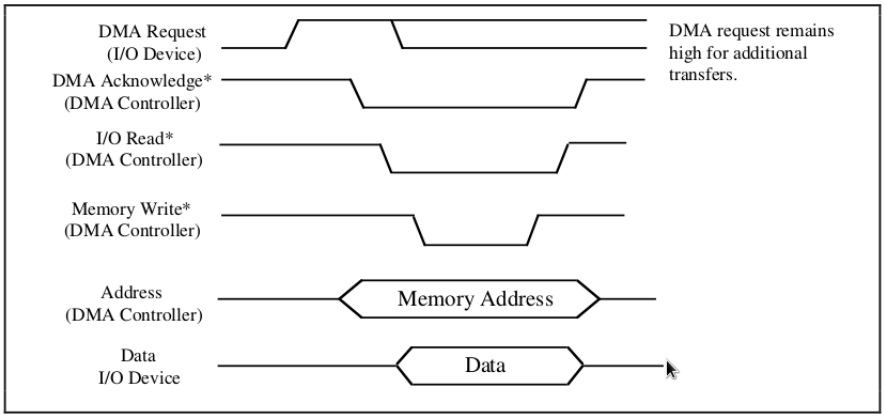

Figura 1.4: Signal protocol Flyby DMA [8]

La tecnica di trasferimento di tipo *fetch-and-deposit* [8] [14] (detta anche *dual-cycle*, *dual-address*, *flow-through*) richiede due cicli di memoria o I/O. Il dato che deve essere trasferito verrà letto dalla sorgente (memoria o device I/O) e salvato in un registro temporaneo del DMAC. Nel ciclo successivo si procederà alla scrittura del dato in memoria o sulla periferica I/O. In Figura 1.5 è mostrato il *signal protocol* del fetch-and-deposit.

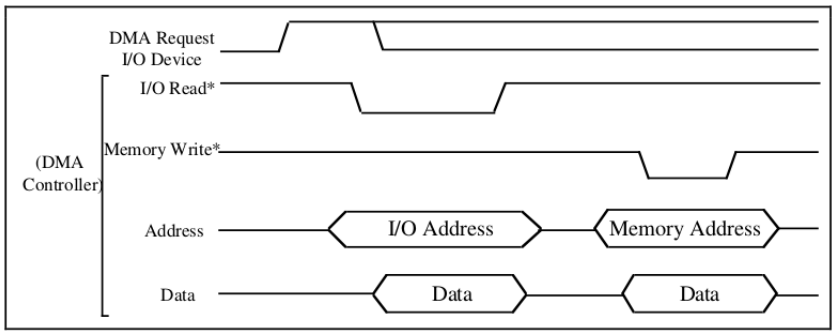

Figura 1.5: Signal protocol Fetch-and-Deposit DMA [8]

Sebbene inefficiente, poiché il controllore DMA esegue due cicli e mantiene il bus di sistema più a lungo, questo tipo di trasferimento è utile per interfacciare i dispositivi che sono collegati a dei bus dati di dimensioni diverse. Ad esempio, un controllore DMA può eseguire due operazioni di lettura, ognuna di 16 bit, verso una sorgente A ed seguire un'operazione di scrittura, di 32 bit, verso una destinazione B. Per supportare questo tipo di trasferimento il DMA Controller deve essere dotato di due registri di indirizzo per canale (indirizzo di origine e destinazione) e di registri bus-size, oltre ai soliti registri (conteggio e controllo). Inoltre a differenza del *flyby* i trasferimenti di tipo *fetch-and-deposit* possono anche essere del tipo: *Memory to Memory* e *I/O-device to I/O-device*.

1.3.2 Modalità di Trasferimento

Il trasferimento dei dati in DMA può avvenire in vari modi:

- **trasferimento a blocchi** (*burst transfer*)
- **trasferimento con cycle stealing**
- **trasferimento transparent/hidden**

Nel trasferimento a blocchi il DMAC prende il controllo del bus di sistema e avvia il trasferimento di tutte le *word* del blocco di dati, successivamente rilascia il controllo del bus che tornerà sotto il controllo della CPU. La grandezza del blocco dati è determinata in fase di inizializzazione dalla CPU. Con questo metodo si ottiene la più alta velocità di trasferimento tra dispositivi di I/O e memoria. Lo svantaggio è che la CPU non sarà in grado di accedere alla memoria per tutta la durata del trasferimento. Il

burst transfer è particolarmente utile per periferiche quali i dischi magnetici, dove il trasferimento del blocco dati non può essere interrotto.

Nel secondo caso (*cycle stealing*), il DMAC richiede il controllo del bus di sistema con le stesse modalità del *burst transfer*. Una volta ottenuto il bus dalla CPU, il trasferimento di dati avviene una *word* alla volta, quindi il bus di sistema sarà sottratto alla CPU per brevi intervalli di tempo. Le richieste di trasferimento per un blocco di dati saranno uguali al numero di word del blocco stesso, questo porta ad un maggior overhead (rispetto al burst transfer) dovuto alle molteplici negoziazioni del bus e setup delle transazioni. Inoltre a seguito della sottrazione dei cicli di bus la CPU subirà un conseguente rallentamento prestazionale, mitigato nella maggior parte dei casi dall'uso della memoria cache.

Infine nel *trasparent/hidden* il DMAC (o un'unità hardware apposita) è in grado di rilevare quando la CPU non utilizza il bus, e solo in quei periodi esegue il trasferimento dei dati. In pratica il trasferimento DMA di una *word* avviene quando il processore esegue un'istruzione che lascia sufficienti cicli di clock vuoti. In tal modo la CPU non è praticamente rallentata dal DMA Controller, ma il trasferimento di un blocco di dati può richiedere più tempo rispetto alle precedenti modalità.

Capitolo 2

Intel I/O Acceleration Technology

Nel corso degli anni le prestazioni delle CPU dei server e la larghezza di banda delle reti sono migliorate, ma il metodo principale per il trasferimento dei dati è rimasto invariato. Fino ad oggi, infatti, il processore si fa quasi completamente carico della gestione dello stack di rete (TCP/IP) ossia della: ricezione, elaborazione e memorizzazione di ogni pacchetto dati [20]. Questo porta ad un utilizzo della CPU inefficiente ed un limitato throughput di rete.

L'Intel I/O Acceleration Tecnology (ovvero Input/Output Acceleration Technology) è un insieme di tecnologie sviluppate per ridurre l'utilizzo della CPU nella gestione dello stack di rete e per migliorare la velocità di trasmissione fra interfacce di rete e applicazioni server.

2.1 TCP/IP Overhead

Prima di presentare i benefici delle tecnologie I/OAT dell'Intel, è bene analizzare la natura del problema, che si vuole risolvere, e le soluzioni già esistenti. Quindi si procederà all'analisi dell'intera sequenza di operazioni necessarie al server per: ricevere, processare e inviare un pacchetto dati.
In dettaglio i vari step necessari alla ricezione-invio di un pacchetto dati (vedi Figura 2.1) sono [21]:

- 1: Un client invia una richiesta via pacchetto dati TCP/IP, che il server riceve attraverso l'interfaccia della scheda di rete (NIC, Network Interface Card).

- 2: Il server elabora l'header TCP/IP del pacchetto e manda il *payload* all'applicazione. Nello specifico il server deve prima eseguire i calcoli richiesti dallo stack TCP/IP, effettuare degli accessi in memoria per reperire i vari descrittori del pacchetto, copiare il payload nello spazio di memoria dell'applicazione e gestire altre attività (per esempio: interrupt, buffer e così via).
- 3-5: L'applicazione riconosce la richiesta del client e preleva i dati dallo *storage* per soddisfare la richiesta.
- 6: L'applicazione completa l'elaborazione della richiesta client utilizzando i dati prelevati.
- 7: Il server costruisce il pacchetto dati TCP/IP e lo invia al client.

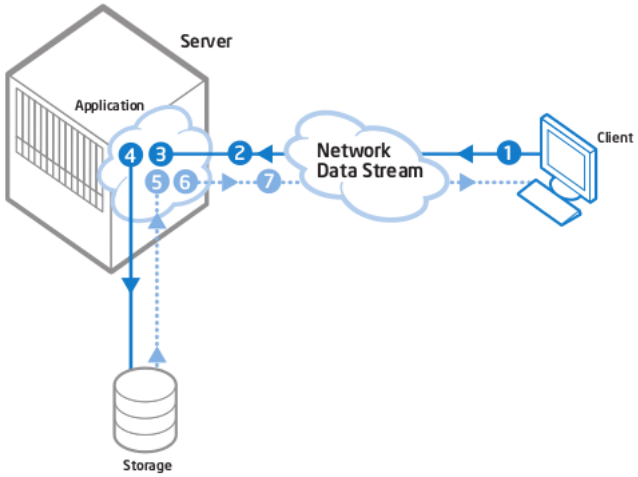

Figura 2.1: Request-Response Data Path [21]

Gli step sopra illustrati sono rimasti sostanzialmente invariati per più di un decennio, ma questo meccanismo con il volume di traffico e la larghezza di banda delle attuali reti è fonte di un degrado prestazionale dell'intero sistema. Analizzando il processo di ricezione-invio di un pacchetto dati è possibile individuare tre *classi di overhead* [19] [22] [28]:

- *System Overhead*: gestione delle interruzioni, dei buffer e dei context switch.
- *TCP/IP Processing*: processamento del codice dello stack TCP/IP.
- *Memory Access*: operazioni in memoria e CPU Stall.

E' importante notare che, prima dell'avvento dell'Intel I/OAT, esistevano già alcune tecnologie capaci di ridurre il *sender-side overhead* della CPU nell'elaborazione dei pacchetti:

- **TCP Segmentation Offload (TSO)**
  Per inviare grandi quantità di dati su una rete, è necessario suddividere il carico in segmenti che possono transitare attraverso vari elementi intermedi (ad esempio router e switch), posti tra il computer di origine e quello di destinazione. Questo processo è denominato *segmentazione* ed è spesso effettuato dal protocollo TCP del computer host. Il *TCP Segmentation Offload* permette al sistema operativo di inviare buffer, che eccedono la MTU (Maximum Transmission Unit), al *network controller*, il quale si farà carico dell'operazione di segmentazione. Ricevuto il buffer di dati, il controllore lo partizionerà in pacchetti Ethernet che poi trasmetterà sulla rete. Utilizzando questa tecnica si ha una conseguente diminuzione dell'overhead della CPU, in quanto il processore interagisce meno frequentemente con il controller di rete [23] [24].

- **Zero-copy Operation**
  L'approccio *Zero-copy* permette di inviare dati dimezzando i cambi di contesto e le copie in memoria. Ovviamente l'efficacia reale di questo metodo dipende dal sistema operativo, ad esempio in molte distribuzioni Linux e UNIX è realizzato tramite la funzione `sendfile()`. Con la *Zero-copy* il kernel non ha bisogno di copiare i dati utente nel buffer di rete, ma può usare come sorgente le pagine che sono nella *page cache*. In questo modo, si evita anche di sporcare la cache della CPU con dati che non saranno più usati. [25] [26] [27].

2.2 Le funzionalità dell'I/OAT

Le tecnologie Intel I/O Acceleration sono state sviluppate per ridurre l'*overhead receiver-side* della CPU nell'elaborazione dei pacchetti. L'utilizzo di un motore DMA dedicato e l'introduzione di controller di rete con nuove funzionalità, hanno permesso di mitigare l'impatto delle tre classi di overhead sulle prestazioni del sistema. Infatti le operazioni di copia in

memoria adesso sono eseguite dal motore DMA, con un conseguente alleggerimento del carico della CPU, che può dedicarsi ad altri lavori. Inoltre, le nuove funzionalità dei controller di rete permettono anche di distribuire la gestione delle interruzioni su più CPU migliorando così l'utilizzo della cache.

Grazie a queste tecnologie è stato possibile introdurre nuove tecniche atte al miglioramento delle prestazioni in ricezione, le principali sono [30]:

- **Split Headers**
- **Multiple Receive Queues**
- **Direct Cache Access**
- **DMA Copy Offload Engine**

Ognuna di queste funzionalità può essere implementata/supportata senza richiedere modifiche radicali allo stack di rete del kernel Linux.

2.2.1 Split Headers

Quando il controllore della scheda di rete deve memorizzare un pacchetto ricevuto, questi tipicamente trasferisce gli header TCP/IP ed Ethernet insieme ai dati (payload) in un singolo buffer.
La tecnica di *Split Headers* dell'I/OAT permette al controllore di separare in ogni pacchetto di rete la parte header da quella dati, così da memorizzarle in due buffer distinti. In questo modo si hanno diversi vantaggi:

- Si ha un perfetto allineamento degli header e dei dati.
- Il buffer dei dati di rete può essere diviso in un piccolo *slab-allocated header buffer* e in un più grande *page-allocated data buffer*. Sorprendentemente, a seguito di questo partizionamento l'allocazione dei suddetti buffer risulta più veloce di quella ottenuta per il singolo, ciò è da imputare all'implementazione del *buddy allocator*.
- Inoltre visto che gli header hanno una più alta frequenza di accessi rispetto alla parte dati, con la tecnica di *split headers* si ha un miglior utilizzo della cache (che non viene sporcata dalla parte dati dei pacchetti).

2.2.2 Multiple Receive Queues

La tecnica di *Multiple Receive Queues* permette di distribuire l'elaborazione dei pacchetti *receiver-side* su più CPU, migliorando così l'utilizzo delle risorse di sistema disponibili ed il throughput anche in presenza di carichi considerevoli. Prima di questa tecnica la gestione degli interrupt del controllore di rete era delegata ad un'unica CPU (anche su architetture multi processor), che rischiava di diventare il collo di bottiglia del sistema. L'intel ha risolto questo problema fornendo ai nuovi controller di rete più code di ricezione in cui memorizzare i pacchetti, in modo da poter parallelizzare l'elaborazione di rete su più CPU. Inoltre nei sistemi multi-core che supportano anche l'*HyperThreading* si garantisce che i thread usati per l'elaborazione dei pacchetti non condividano lo stesso core fisico.

I pacchetti possono essere associati ad una coda di ricezione in due modi [31] [32] [33]:

1. **RSS** (*Receive Side Scaling*): la coda di destinazione viene determinata calcolando l'hash su alcuni campi header del pacchetto, in modo da far gestire alla stessa CPU un intero flusso TCP [34].

2. **VMDq** (*Virtual Machine Device Queues*): la coda di destinazione viene determinata in base al MAC address o al VLAN tag.

Alcuni esempi di controller che supportano la tecnica di *multiple receive queues* sono:

- l'*Intel 82575 Gigabit Ethernet Controller* che per ogni porta ha 4 code per la trasmissione e 4 per la ricezione;

- l'*Intel 82598 10 Gigabit Ethernet Controller* che per ogni porta ha 32 code per la trasmissione e 64 per la ricezione.

2.2.3 Direct Cache Access

Il *Direct Cache Access* migliora le prestazioni del sistema permettendo la scrittura dei dati, acquisiti dalla scheda di rete, verso la memoria cache del processore designato. In un normale sistema di acquisizione infatti il dato ricevuto deve transitare dalla scheda di rete alla memoria di sistema, prima di poter raggiungere la memoria cache del processore, aggiungendo quindi

all'operazione una notevole latenza. Inoltre bisogna considerare l'ulteriore overhead dovuto ai meccanismi che mantengono la coerenza dei dati contenuti nella cache. Il DCA risolve questi problemi permettendo alla scheda di rete di scrivere il dato appena acquisito direttamente all'interno della cache del processore. I principali benefici del DCA sono: la considerevole riduzione dei cache miss e il miglioramento del tempo di risposta delle applicazioni.

2.2.4 DMA Copy Offload Engine

Durante l'elaborazione dei pacchetti *receive-side*, la CPU spende una porzione di tempo considerevole per le copie dei dati dalla memoria kernel a quella dell'applicazione [30]. L' I/OAT cerca di risolvere questo problema, introducendo un motore DMA asincrono (ADCE, Asynchronus DMA Copy Engine) dedicato ai trasferimenti Memory to Memory. L'ADCE è stato implementato all'interno del chipset MCH (Memory Controller Hub) come un *pci-enumerate device* e dispone di più canali indipendenti che hanno accesso diretto alla memoria principale. Così mentre il motore DMA esegue le copie in memoria, la CPU è libera di procedere con l'elaborazione del pacchetto successivo, o con altre attività precedentemente sospese. Finita l'operazione di copia il motore DMA, può opzionalmente generare un'interruzione per avvisare il processore della conclusione del lavoro. La Figura 2.2 mostra i vari elementi di sistema coinvolti in un'operazione di copia tramite CPU e motore DMA.

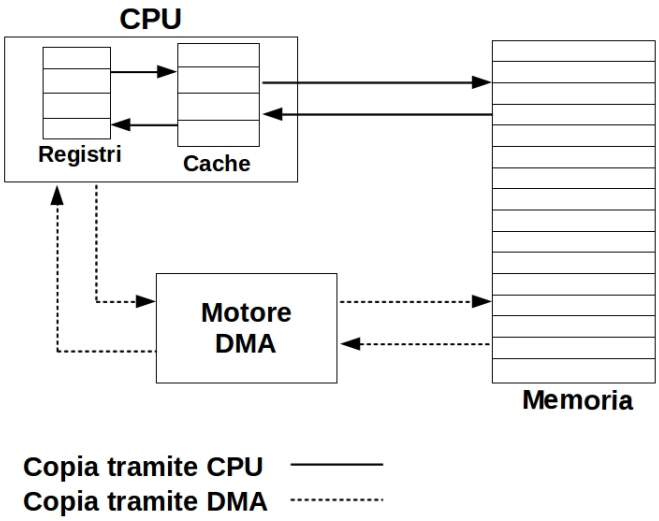

Figura 2.2: Operazione di copia su CPU vs motore DMA

Come menzionato in [35] l'introduzione di un motore dedicato per i trasferimenti in memoria, oltre a diminuire l'overload della CPU, apporta significativi benefici in termini di:

- *Utilizzo delle risorse della CPU*: le copie in memoria sono implementate attraverso una serie di operazioni di *load* e *store*, che coinvolgono la memoria e i registri della CPU. Quindi l'operazione di copia avviene tramite i registri del processore che hanno una grandezza di 32 o 64 bit. Il motore DMA effettua le operazioni di copia direttamente sulla memoria, quindi la grandezza del blocco di dati non soffre di suddetta limitazione. Inoltre, le istruzioni di *load* e *store* usate nelle operazioni di copia tramite CPU possono occupare: le *load/store queues*, il ROB (*re-order buffer*) e il *line fill buffer* [36]; limitando di fatto la capacità del processore di utilizzare l'*instruction window* [37] [38] (insieme di istruzioni contenute nel ROB) per eseguire altre istruzioni.

- *Cache Polluction*: le operazioni di copia, che coinvolgono blocchi di dati di dimensioni considerevoli, possono 'inquinare' significativamente la memoria cache. Inoltre se i blocchi di memoria non vengono usati dall'applicazione, l'allocazione di questi dati in cache può inutilmente provocare la rimozione di preziose risorse precedentemente memorizzate. Nell'elaborazione TCP/IP, come menzionato in [36], dopo un trasferimento di dati dallo spazio di memoria kernel a quello utente raramente si accede al blocco di memoria sorgente. Infatti una volta passato il payload dei pacchetti di rete all'applicazione, la copia residente nella memoria kernel di norma non è più usata. Per la copia locata in memoria utente, la situazione non è dissimile, infatti molte applicazioni come i *web server* non utilizzano immediatamente i dati di rete. L'utilizzo di un motore DMA permette di effettuare le copie dirette in memoria senza mettere i dati in cache, cosi da mitigare la cache pollution.

Anche se l'ADCE offre numerosi vantaggi, ci sono alcuni aspetti che ne limitano i benefici. Prima di tutto, bisogna considerare che il *memory controller* lavora con indirizzi fisici, così un singolo trasferimento in memoria non può coinvolgere pagine fisiche non contigue. Quindi in questo caso le operazioni in memoria dovrebbero essere effettuate tramite più trasferimenti di singole

pagine. Infine quando si effettua un trasferimento dati tramite un motore DMA, si dovrebbe controllare se la regione di memoria coinvolta (ossia il buffer destinazione che viene scritto) è locata anche nella cache del processore, nel qual caso è necessario ripristinare la sua coerenza: aggiornando o cancellando i suddetti dati in cache.

2.3 I/OAT nel kernel Linux

Le tecnologie I/OAT sono ampiamente supportate dai sistemi Linux, infatti è stata la stessa casa produttrice Intel ad implementare il driver/modulo del motore DMA asincrono (ioatdma.ko) e a modificare il codice kernel dello stack di rete. La prima patch dell'Intel fu rilasciata il 3 Marzo 2006 e venne inserita nel kernel Linux 2.6.18 [39], che fu il primo a fornire un supporto alle tecnologie di I/OAT. Nei kernel più recenti come il 3.9.2, che sarà preso come riferimento nel resto della trattazione, il codice per il supporto dell'I/OAT si è arricchito di nuove funzionalità, che però non hanno snaturato l'architettura software originaria. Quest'ultima infatti è sempre stata caratterizzata da tre componenti fondamentali: il driver del motore DMA, il sottosistema DMA ed il supporto I/OAT nello stack di rete. Nel seguito si darà maggior risalto al sottosistema DMA, analizzando come questi è usato nello stack di rete.

2.3.1 Il Sottosistema DMA

Il motore DMA I/OAT è stato introdotto per migliorare le prestazioni dei server soggetti ad alti carichi di rete, ma il suo funzionamento non è direttamente associato con il sottosistema di rete o con il driver del network controller. L'intel infatti ha implementato nel kernel Linux un sottosistema DMA, che fornisce un'interfaccia generica per le operazioni di copia effettuate in modo asincrono. Grazie a questa scelta progettuale è possibile utilizzare le funzionalità del motore DMA anche in altre parti del kernel, se opportunamente modificate. Inoltre altri driver di motori DMA, che sono in grado di effettuare le copie in memoria in modo asincrono, possono registrarsi per sfruttare l'astrazione fornita da questo sottosistema. Come mostrato in Figura 2.3 il sottosistema DMA fa da intermediario tra le

richieste client (codice del kernel che vuole utilizzare il motore DMA) ed il driver.

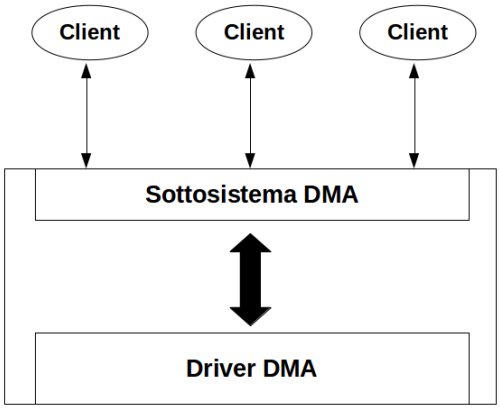

Figura 2.3: Sottosistema DMA

Un client che vuole effettuare trasferimenti DMA, deve innanzitutto interagire con il sottosistema per farsi assegnare le risorse di cui necessita. Il sottosistema supporta due modalità diverse di assegnazione:

- **private channel**: Il client, tramite la funzione `dma_request_channel()`, richiede al sottosistema l'uso esclusivo di un canale DMA. Il sottosistema verifica se sono presenti canali liberi, nel qual caso ne assegna uno al richiedente. Terminato il trasferimento il client rilascia il canale, eseguendo `dma_release_channel()`.

- **public channel**: Il client non richiede il canale DMA in modo esclusivo, ma si registra al sottosistema, tramite la funzione `dmaengine_get()`, che incrementa i *reference count* dei canali liberi. Conclusa la registrazione il client esegue `dma_find_channel()` per richiedere un canale al sottosistema, quest'ultimo analizzerà determinati parametri e assegnerà al client il canale a lui più idoneo. Infine concluso il lavoro il client può cancellare la sua registrazione al sottosistema con `dmaengine_put()`.

In entrambe le modalità di assegnazione è richiesto al client di specificare il tipo di operazione richiesta al canale, infatti il sottosistema DMA fornisce una serie di attributi `dma_transaction_type` che combinati permettono di descrivere il tipo di operazione desiderata. Ad esempio se un client necessitasse di un trasferimento Memory to Memory in modalità *private*

channel, per descrivere il tipo di operazione dovrebbe utilizzare i seguenti attributi: `DMA_MEMCPY` e `DMA_PRIVATE`.
Nel seguito sono mostrati i `dma_transaction_type` supportati dal sottosistema DMA:

```
enum dma_transaction_type
{
    DMA_MEMCPY,
    DMA_XOR,
    DMA_PQ,
    DMA_XOR_VAL,
    DMA_PQ_VAL,
    DMA_MEMSET,
    DMA_INTERRUPT,
    DMA_SG,
    DMA_PRIVATE,
    DMA_ASYNC_TX,
    DMA_SLAVE,
    DMA_CYCLIC,
    DMA_INTERLEAVE,
    DMA_TX_TYPE_END,
};
```

Una volta che il client ha ottenuto un canale DMA, per effettuare le operazioni di copia in memoria, può utilizzare le seguenti funzioni [40], che permettono di lavorare sia con buffer che pagine di memoria:

```
dma_cookie_t dma_async_memcpy_buf_to_buf(struct dma_chan *chan, void *dest,
  void *src, size_t len)

dma_cookie_t dma_async_memcpy_buf_to_pg(struct dma_chan *chan,
  struct page *page, unsigned int offset, void *kdata, size_t len)

dma_cookie_t dma_async_memcpy_pg_to_pg(struct dma_chan *chan,
  struct page *dest_pg, unsigned int dest_off, struct page *src_pg,
  unsigned int src_off, size_t len)
```

Come è facile notare, le funzioni richiedono un descrittore `struct dma_chan` che identifica il canale DMA assegnato al client e ritornano un `dma_cookie_t`. Quest'ultimo è utilizzato dal sottosistema per identificare le transazioni sottomesse dal client.

Poiché le copie in memoria avvengono in maniera asincrona, il client per verificare lo stato della transazione può utilizzare la funzione:

```
enum dma_status dma_sync_wait(struct dma_chan *chan, dma_cookie_t cookie)
```

che ritorna i seguenti valori:

- DMA_SUCCESS: la transazione DMA associata al cookie è terminata in modo corretto.
- DMA_IN_PROGRESS: la transazione DMA associata al cookie non è ancora terminata.
- DMA_ERROR: la transazione DMA associata al cookie è fallita per qualche ragione.

Infine è interessante analizzare la struttura dati dma_chan, utilizzata dal sottosistema per rappresenta e gestire un canale DMA:

```
struct dma_chan
{
  struct dma_device *device;
  dma_cookie_t cookie;
  dma_cookie_t completed_cookie;
  ..........
  struct list_head device_node;
  int client_count;
  ..........
  };
```

Il campo device è utilizzato per risalire al dispositivo che ha messo a disposizione il canale. cookie è l'ultimo dma_cookie ritornato ad un client, mentre completed_cookie è il dma_cookie dell'ultima transazione completata. Infine il campo device_node è usato per collegare il canale alla lista dei canali gestiti dallo stesso dispositivo e client_count tiene traccia di quanti client stanno usando il canale.

2.3.2 I/OAT nello Stack di Rete

Nello stack di rete l'operazione di copy_to_user è utilizzata per copiare i pacchetti, contenuti in un array di strutture sk_buff (ogni elemento generalmente rappresenta un pacchetto di rete), nello spazio di memoria dell'applicazione utente (un array di strutture iovec, dove ogni elemento rappresenta un buffer utente).
Grazie alle tecnologie I/OAT è stato possibile parallelizzare il processo di copia con l'elaborazione dei pacchetti, delegando la prima operazione al motore DMA e la seconda al processore. Prima di usare il motore DMA, come detto precedentemente, i client devono interagire con il sottosistema DMA. In questo caso i client sono le interfacce di rete, che in fase di inizializzazione (mentre viene eseguita la funzione __dev_open() locata

in `net/core/dev.c`) si registrano al sottosistema, tramite la funzione `net_dmaengine_get()`.

```
/* include/linux/dmaengine.h */
#define net_dmaengine_get() dmaengine_get()

/* net/core/dev.c */
int dev_open(struct net_device *dev)
{
  ......
  ret = __dev_open(dev);
  ......
}

static int __dev_open(struct net_device *dev)
{
  ......
  net_dmaengine_get();
  ......
}
```

Le restanti operazioni sono eseguite nello stack TCP dalla funzione `tcp_recvmsg()` (locata in `net/ipv4/tcp.c`), che è utilizzata per copiare i dati di rete nella memoria utente. Analizzando il codice della suddetta funzione è possibile individuare le azioni necessarie al trasferimento DMA:

a) lock delle pagine utente

b) richiesta di un public channel

c) trasferimento DMA

d) verifica/attesa della conclusione del trasferimento

e) unlock delle pagine utente

```
int tcp_recvmsg(struct kiocb *iocb, struct sock *sk, struct msghdr *msg,
    size_t len, int nonblock, int flags, int *addr_len)
{
    ........
(a) dma_pin_iovec_pages(msg->msg_iov, len);
    ........
(b) tp->ucopy.dma_chan = net_dma_find_channel();
    ........
    if (tp->ucopy.dma_chan) {
(c)     tp->ucopy.dma_cookie = dma_skb_copy_datagram_iovec(
(c)     tp->ucopy.dma_chan, skb, offset,
(c)     msg->msg_iov, used,
(c)     tp->ucopy.pinned_list);
    .......
```

```
(d)   tcp_service_net_dma(sk, true);
      tp->ucopy.dma_chan = NULL;
      .......
(e)   dma_unpin_iovec_pages(tp->ucopy.pinned_list);
      .......
  }
```

Durante una copia in memoria, tramite motore DMA, si deve evitare che le pagine utente siano swappate su disco. Per far fronte a questa spiacevole situazione, è possibile usare la funzione `get_user_pages()`, che permette di bloccare le pagine utente in memoria. Quindi prima di effettuare un trasferimento è necessario eseguire il lock delle pagine utente e richiedere un canale DMA al sottosistema. Infine una volta che il lavoro del motore DMA è terminato, bisogna marcare le pagine utente come 'dirty' e sbloccarle.

Capitolo 3

Panoramica delle Soluzioni Esistenti

Le tecnologie I/OAT sono state le prime, ma non le uniche, ad utilizzare un motore DMA per le operazioni di copia tra due regioni di memoria. Nel seguito verrà effettuata una panoramica sui principali lavori riguardanti sistemi che, tramite hardware dedicato, eseguono trasferimenti di dati in memoria.

3.1 Benefici dell'I/OAT nelle applicazioni di rete

Le tecnologie I/OAT sono state introdotte per ridurre l'overhead del processore nell'elaborazione receive-side dei pacchetti di rete. L'articolo [41] è uno dei primi lavori che analizza i benefici dell'I/OAT in ambito di rete. Gli autori tramite l'uso del benchmark *ttcp*, che misura la banda di rete, analizzano l'utilizzazione della CPU su due configurazioni:

- **Non-I/OAT**: il sistema su cui viene fatto il test non utilizza le tecnologie I/OAT.
- **I/OAT**: il sistema su cui viene fatto il test sfrutta le tecnologie I/OAT.

I test prestazionali sono stati eseguiti su un sistema composto da due nodi ognuno avente un processore Intel dual dual-core da 3.46 GHZ, 2MB di cache L2 e tre schede di rete Intel PRO 1000MBit (ogni scheda di rete ha due porte).

Nel primo test gli autori misurano su un solo nodo del sistema la banda di rete e l'utilizzazione della CPU, al crescere del numero di porte di rete attivate. Come mostrato in Figura 3.1 la banda misurata nella configurazione I/OAT è simile a quella della Non-I/OAT, anche variando il numero di porte.

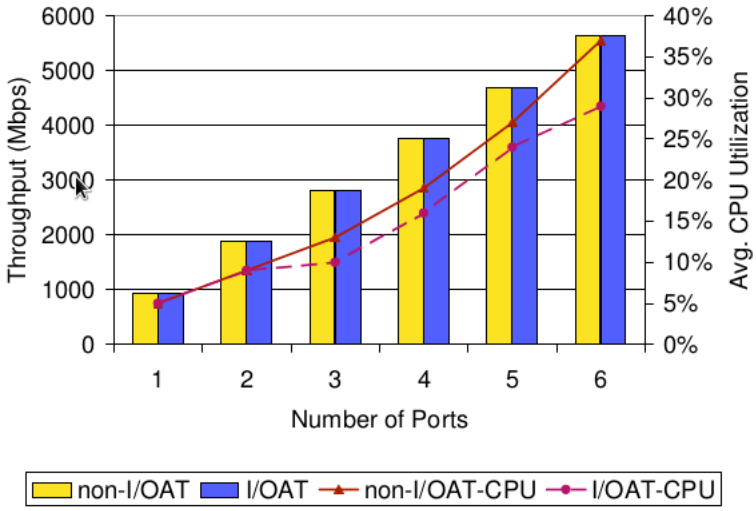

Figura 3.1: Banda e Overhead CPU, su un nodo del Sistema [41]

Analizzando invece l'utilizzazione della CPU è possibile notare una differenza di prestazioni tra le due configurazioni. La configurazione I/OAT utilizza di meno la CPU rispetto alla Non-I/OAT, questa differenza è evidente quando si utilizzano almeno tre porte di rete e cresce all'aumento delle suddette. Infatti analizzando le prestazioni del sistema che espone 6 porte di rete si nota che la configurazione Non-I/OAT ha un'utilizzazione del 37% della CPU rispetto al 29% della I/OAT. Quindi con questo primo test è possibile asserire che i benefici introdotti dalle tecnologie I/OAT aumentano al crescere del carico di rete.

Gli autori del [41] presentano un secondo test, in cui vengono utilizzati entrambi i nodi del sistema, al fine di analizzare la banda di rete in entrambe le direzioni e l'utilizzazione della CPU. Ogni nodo si serve di $2 \cdot N$ thread, così partizionati: N thread svolgono il ruolo di server ed N da client. Ogni thread client di un nodo è connesso con un unico thread server dell'altro nodo, così da avere $2 \cdot N$ connessione tra i due nodi. Su ogni connessione per generare traffico di rete è eseguito il benchmark *ttcp*. Per l'analisi delle perfomance di banda viene calcolata la banda totale, ossia: la somma di tutte le bande su tutti i thread dei nodi in entrambe le direzioni. Nel suddetto esperimento il

parametro N è pari al numero di porte di rete. Come mostrato in Figura 3.2 la banda totale misurata nella configurazione I/OAT è simile a quella della Non-I/OAT, anche variando il numero di porte.

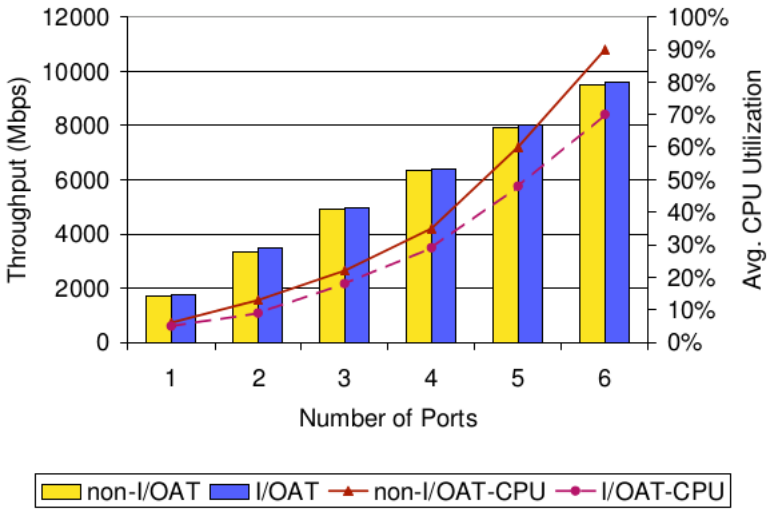

Figura 3.2: Banda e Overhead CPU, su due nodi del Sistema [41]

Analizzando invece l'utilizzazione CPU è possibile notare una differenza di prestazioni tra le due configurazioni. La configurazione I/OAT utilizza di meno la CPU rispetto alla Non-I/OAT questa differenza è evidente quando si utilizzano almeno due porte di rete e cresce all'aumento del numero delle suddette. Infatti analizzando le prestazioni del sistema che espone 6 porte di rete, si nota che la configurazione Non-I/OAT ha un'utilizzazione del 90% della CPU mentre quella della I/OAT è del 70%. Quindi rispetto al primo test, avendo un carico di rete quasi raddoppiato, si ha un conseguente incremento dei benefici apportati dall'I/OAT nell'utilizzo delle risorse della CPU.
L'articolo [41] inoltre presenta altri test eseguiti in ambiente cloud, che mostrano come l'utilizzo delle tecnologie I/OAT riesca anche in questa casistica ad abbassare l'utilizzo del processore. Quindi in conclusione le applicazioni di rete che fanno uso delle tecnologie I/OAT occupano in maniera minore le risorse della CPU rispetto a quelle che utilizzano la comunicazione di rete tradizionale.

3.2 Benefici dell'I/OAT nelle operazioni di copia in memoria

Il motore DMA dell'I/OAT può essere sfruttato anche in contesti diversi da quello di rete, ad esempio gli articoli [42] [43] presentano quattro configurazioni di sistema, singolo processore e multi processore, che sfruttano il motore DMA o la CPU per eseguire l'operazione di *memcpy*:

1. **SCI** (*Single-Core with I/OAT*): Sistema composto da una singola CPU, che utilizza il motore DMA per l'operazione di *memcpy*.

2. **MCI** (*Multi-Core with I/OAT*): Sistema composto da più CPU, che utilizza il motore DMA per l'operazione di *memcpy*.

3. **SCNI** (*Single-Core with No I/OAT*): Sistema composto da una singola CPU, che non utilizza il motore DMA.

4. **MCNI** (*Multi-Core with No I/OAT*): Sistema composto da più CPU, che non utilizza il motore DMA e delega ad un processore l'operazione di *memcpy*.

La configurazione SCI effettua le operazioni di copia in memoria tramite il motore DMA ed utilizza un modulo kernel per esportare alle applicazioni utente le funzionalità dell'hardware dedicato[1] (vedi Figura 3.3). Quindi un'applicazione utente che vuole effettuare un'operazione di *memcpy* deve interagire con il modulo del kernel, che a sua volta inizializzerà uno o più canali DMA adibiti a svolgere suddetta operazione. Infine al termine della *memcpy*, il modulo del kernel si preoccuperà di notificare la conclusione dell'operazione all'applicazione utente.

[1] Il motore DMA è accessibile e configurabile solo in spazio kernel

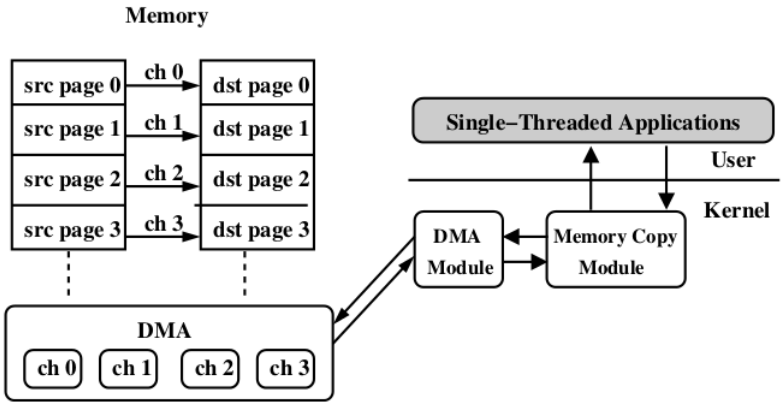

Figura 3.3: Configurazione di Sistema SCI [43]

La configurazione di sistema SCI riduce il carico della CPU derivante dalle operazioni di copia in memoria ed evita in parte i fenomeni di Cache Pollution e CPU Stall, ma è affetta principalmente da due tipologie di overhead: il setup del motore DMA e il lock/unlock delle pagine di memoria utente. Per mitigare questi overhead gli autori dell'articolo [43] hanno introdotto la configurazione MCI (vedi Figura 3.4), che si serve di un kernel thread, locato su una CPU diversa da quella su cui viene eseguita l'applicazione utente, per effettuare le varie operazioni connesse al motore DMA (setup, lock/unlock pagine utente, intermediazione tra driver del motore e applicazione utente ed altre ancora). Il kernel thread per gestire le richieste delle applicazioni utente si serve di due liste: una per tenere traccia delle operazioni di *memcpy* che sono in corso e una seconda per quelle che sono terminate ma devono essere ancora notificate.

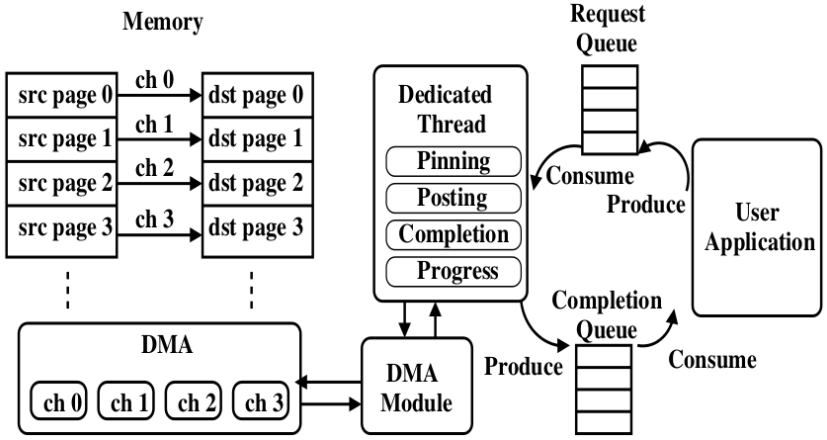

Figura 3.4: Configurazione di Sistema MCI [43]

La configurazione SCNI non necessita di spiegazione, in quanto rappresenta un sistema convenzionale a singola CPU che effettua l'operazione *memcpy* tramite processore. Infine la configurazione MCNI sfrutta il sistema multi-core per dedicare in maniera esclusiva una CPU alle operazione di copia in memoria, le restanti caratteristiche sono identiche a quelle della MCI.

Per confrontare le prestazioni delle quattro configurazioni, gli autori dell'articolo [43] hanno eseguito dei test relativi al tempo di esecuzione e alla *memory bandwidth* dell'operazione di *memcpy*. Il sistema su cui sono state eseguite le misurazioni ha le seguenti caratteristiche: un processore Intel dual dual-core da 3.46 GHZ, 2MB di cache L2, scheda madre SuperMicro XDB8 e kernel Linux 2.6.20.
Nel primo test gli autori misurano i tempi d'esecuzione della *memcpy* al crescere delle dimensioni dei blocchi di dati. I risultati mostrano che per buffer di dati non superiori alla dimensione di 1MB, i tempi della *memcpy* nelle configurazioni SCNI e MCNI sono di gran lunga inferiori a quelli delle configurazioni SCI e MCI. Mentre, per buffer di dati che superano la dimensione di 2MB, la situazione è invertita. Come è possibile intuire le configurazioni SCNI e MCNI hanno prestazioni migliori fino a quando il blocco di dati è interamente in cache L2. Però al crescere della dimensione del buffer aumentano anche gli accessi in memoria principale, con un conseguente aumento dei tempi d'esecuzione della *memcpy*.
Il secondo test mira a misurare la *memory bandwidth* della *memcpy* al crescere delle dimensioni dei blocchi di dati. I risultati sono analoghi a quelli del primo test, infatti per buffer di dati non superiori alla dimensione di 1MB, le prestazioni di *memory bandwidth* nelle configurazioni SCNI e MCNI sono di gran lunga migliori di quelle delle configurazioni SCI e MCI. Mentre, per buffer di dati che superano la dimensione di 2MB, la situazione è invertita. Le prestazioni sono influenzate come nel test precedente dalla cache L2.

In conclusione, gli articoli [42] [43] mettono in luce i benefici e le limitazioni circa l'uso del motore DMA dell'I/OAT per i trasferimenti di dati in memoria. Infatti se i buffer coinvolti nelle operazioni di copia hanno dimensioni maggiori della cache L2, si ha un incremento della *memory bandwidth* e una diminuzione del tempo d'esecuzione della *memcpy*. Altresì per buffer di dimensioni inferiori alla memoria cache L2, il motore DMA a seguito dei

costi temporali dovuti al *setup* e al *lock/unlock* delle pagine di memoria utente, presenta prestazioni nettamente inferiori a quelle della CPU.

3.3 Processor-DMA per le operazioni di copia in memoria

Le tecnologie I/OAT non sono state le uniche a servirsi di un hardware dedicato atto alle operazioni in memoria, infatti recentemente sono emersi numerosi lavori su tale argomento. L'articolo [44], ad esempio, propone l'utilizzo di un *Processor DMA Engine*, ossia un motore DMA locato all'interno del processore, a cui delegare le operazioni di copia in memoria. E' possibile così ridurre il carico della CPU e migliorare le prestazioni dei trasferimenti di dati in memoria. Il *processor DMA engine* è stato inserito tra il core del processore e la memoria cache L2 (vedi Figura 3.5), al fine di minimizzare il costo della comunicazione con la CPU.

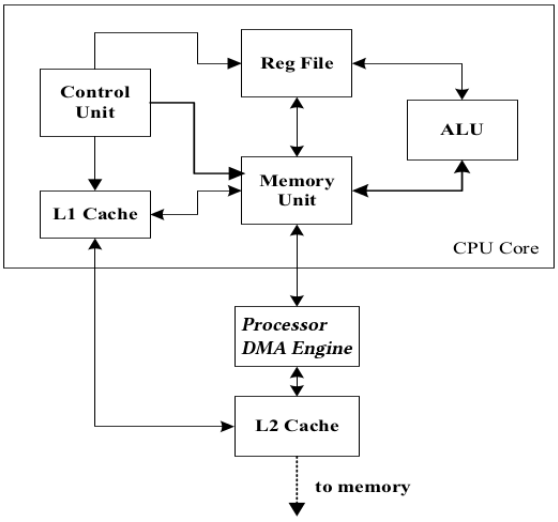

Figura 3.5: Architettura di Sistema con processor DMA engine [44]

Gli autori dell'articolo [44], confrontano le prestazioni del *Processor DMA Engine* con quelle di una configurazione che effettua l'operazione di *memcpy* in modo tradizionale, ossia tramite CPU. Il sistema su cui sono stati eseguiti i test è composto da: un processore Godson3 (anche chiamato Loongson 3 [45]) a 64 bit in cui è stato abilitato un solo core, cache L1 da 64KB e cache L2 da 512KB.

La Figura 3.6 mostra i risultati del test eseguito sulle due configurazioni di sistema al variare della grandezza del blocco di dati passato in input alla funzione *memcpy*.

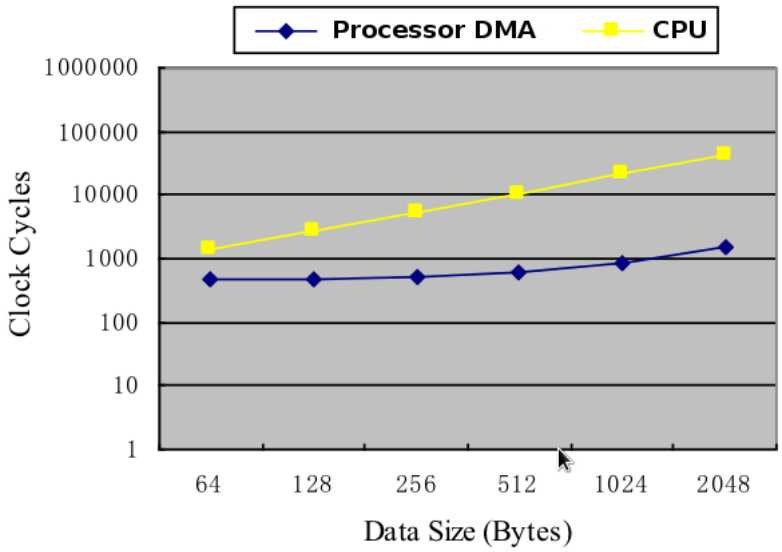

Figura 3.6: Cicli necessari alla memcpy su Processor-DMA e CPU [44]

Come è possibile notare, il tempo di esecuzione della della *memcpy* sul *Processor DMA Engine* è abbastanza costante, mentre quello sulla CPU è direttamente proporzionale alla grandezza del blocco di dati. Inoltre il tempo necessario per eseguire l'operazione di copia con il *Processor DMA Engine* è in media il 93.2% più basso di quello della CPU. Infine quando la dimensione dei dati supera i 2048 byte, si raggiunge il valore massimo del 96.5%, che corrisponde ad uno *speedup* di 28.58 (calcolato con la seguente formula [46]: $S_p = \frac{T_{CPU}}{T_{DMA}}$).
Quindi è possibile concludere, che il *Processor DMA Engine* riesce ad eseguire in maniera efficiente l'operazione di *memcpy* riducendo significativamente il carico della CPU.

Capitolo 4

L'Estensione DMA Copy From To User

Per molti sistemi operativi e device driver il trasferimento di dati tra memoria utente e kernel è un'operazione critica che, in determinate situazioni, può peggiorare le prestazioni dell'intero sistema. Nessun lavoro, di quelli mostrati nel capitolo precedente, propone soluzioni da utilizzare sull'intero spettro dei trasferimenti di dati tra memoria user e kernel. Infatti gli articoli [42] [43] [44] utilizzano motori o processori DMA per eseguire l'operazione di *memcpy* che coinvolge solo le regioni di memoria utente, mentre l'articolo [41] sfrutta la tecnologia I/OAT per effettuare copie di pacchetti di rete dalla memoria kernel a quella utente. Quest'ultimo lavoro purtroppo è circoscritto solo alle operazioni dello stack di rete e sfrutta il motore DMA per eseguire delle copie di dati in un unica direzione: dalla memoria kernel a quella utente.

La Figura 4.1 mostra il numero di trasferimenti di dati tra memoria utente e kernel effettuati da un sistema, sottoposto ad un alto carico di lavoro, in una finestra temporale di 100 sec. I trasferimenti graficati sono ordinati in base alla dimensione del blocco di dati coinvolto.

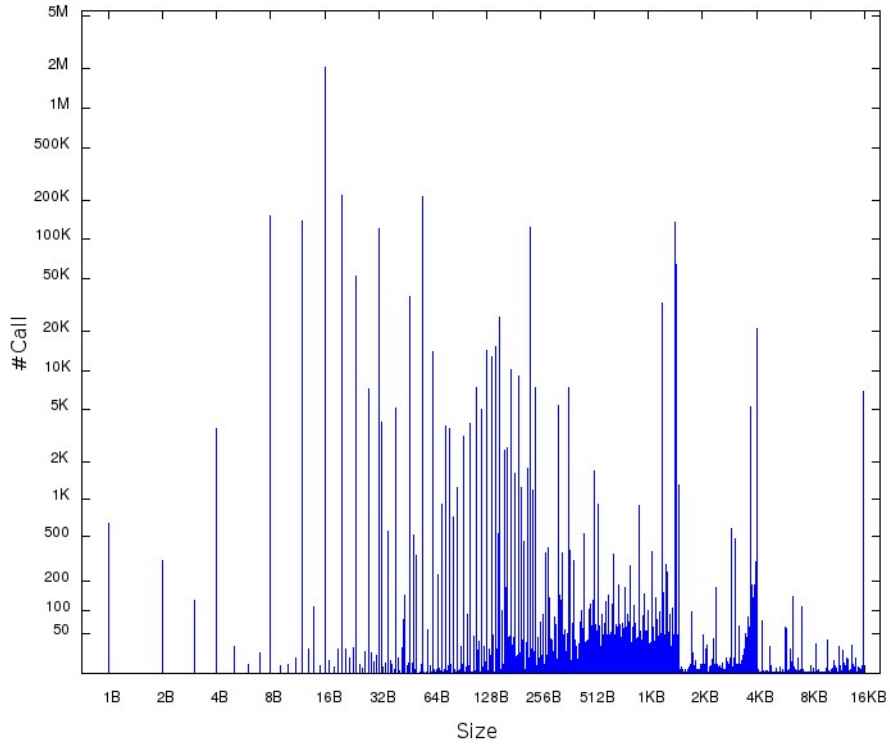

Figura 4.1: Cardinalità dei trasferimenti di dati tra memoria utente e kernel, in una finestra temporale di 100 secondi

Sintetizzando le misurazioni emerge che un sistema operativo, soggetto ad un alto carico di lavoro, effettua circa $3.6 \cdot 10^6$ operazioni di copia tra memoria utente e kernel, in 100 secondi di attività. Queste misurazioni comprovano quanto siano critiche per cardinalità queste operazioni. Così utilizzare un motore DMA, per effettuare le copie tra memoria kernel e utente può, e nel seguito della trattazione sarà dimostrato, apportare benefici all'intero sistema.

4.1 Integrazione nel Sistema

L'estensione DMA Copy From To User, implementata all'interno del kernel Linux, fornisce un'interfaccia generica per le operazioni di copia tra memoria kernel e utente effettuate tramite motore DMA.
Suddetta estensione è costituita da quattro componenti fondamentali:

- **Driver I/OAT-CFTU**
- **Sottosistema DMA-CFTU**

- **Politiche Statiche CFTU**
- **Politiche Dinamiche CFTU**

La Figura 4.2 mostra come sono collegati i vari componenti DMA-CFTU e l'integrazione di questi con il sistema.

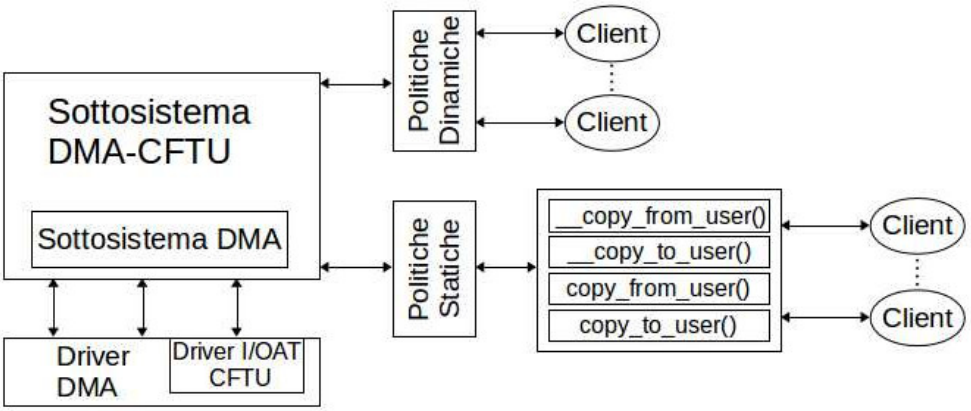

Figura 4.2: Architettura DMA-CFTU

Il *Driver I/OAT-CFTU* estende le funzionalità del driver I/OAT del motore DMA ed è collegato direttamente al *Sottosistema DMA-CFTU*, che fa da intermediario tra i *driver DMA* e le *Politiche CFTU*. Il *Sottosistema DMA-CFTU* estende le funzionalità dell'interfaccia generica introdotta dalle tecnologie I/OAT, permettendo così alle varie *Politiche CFTU* di non essere legate ad un particole motore DMA. Le *Politiche CFTU* regolano l'assegnazione dei canali DMA e specializzano le funzionalità messe a disposizione dal *Sottosistema DMA-CFTU*. Suddette politiche si dividono in due classi: *Politiche Statiche* e *Politiche Dinamiche*. Le prime sono integrate nel codice del kernel durante la sua configurazione, quindi non possono essere sospese/attivate a sistema avviato, mentre le seconde non soffrono di questa limitazione. Le *Politiche Statiche CFTU* specializzano ed integrano le funzionalità del *Sottosistema DMA-CFTU* direttamente nelle funzioni di copia del kernel, così da effettuare tutti i trasferimenti di dati del sistema (tra memoria kernel e utente) tramite motore DMA. Le *Politiche Dinamiche CFTU* non modificano le funzioni del kernel, ma mettono a disposizione delle API per effettuare le operazioni di copia tramite motore DMA.

4.1.1 Driver I/OAT-CFTU

Il *Driver I/OAT-CFTU* è l'unico componente dell'estensione DMA-CFTU che implementa funzionalità legate ad una particolare classe di motori DMA, quelli della tecnologia I/OAT. Questo componente permette al driver I/OAT del motore DMA di disattivare la DCA e di sospendere e risvegliare i canali DMA.

La predicibilità delle operazioni in un sistema real-time è fondamentale, quindi l'utilizzo di un componente impredicibile come la cache, mina l'affidabilità del sistema. Nel driver I/OAT è presente una funzionalità chiamata DCA (*Direct Cache Access*), che permette al motore DMA di caricare i dati nella cache, al fine di ridurre i miss. Il driver dell'I/OAT abilita di default la DCA sul motore DMA e non fornisce un metodo per disattivare tale funzionalità. Come sarà illustrato successivamente, una delle *Politiche CFTU* può essere usata su sistemi real-time, quindi è stato necessario modificare il driver dell'I/OAT per consentire l'attivazione/disattivazione della DCA.
Durante l'inizializzazione del motore DMA I/OAT, il kernel per verificare se la DCA è abilitata nel Bios invoca la funzione `dca_enabled_in_bios()` che ritorna zero se la suddetta funzionalità non è attiva. Così è stata introdotta una *configuration variable* `IOATDMA_DISABLE_DCA` che, se definita, forza la funzione `dca_enabled_in_bios()` a ritornare il valore zero.

```
int system_has_dca_enabled(struct pci_dev *pdev)
{
  #ifndef CONFIG_IOATDMA_DISABLE_DCA
    ........
  #else
  printk(KERN_WARNING "ioat: DCA is disabled \n");
  return 0;
  #endif
}
```

Grazie alla procedura appena illustrata è stato possibile estromettere dalla configurazione di sistema l'uso della DCA.

Nei motori DMA dell'I/OAT è possibile sospendere (*freezare*) i canali per un breve periodo, ma questa caratteristica hardware non è supportata dal driver. Suddetta limitazione non consente di implementare politiche a priorità in cui il bus del motore DMA è assegnato in modo esclusivo, per un determinato intervallo di tempo, ad uno o più canali. Per far fronte a questo problema, è

stata implementata la funzione `ioat_suspend()` utilizzata per sospendere un determinato canale DMA. Per risvegliare il canale è possibile usare la funzione `ioat_resume()` implementata nel driver dell'I/OAT.
Di seguito il codice delle due funzioni:

```
static inline void ioat_suspend(struct ioat_chan_common *chan)
{
  u8 ver = chan->device->version;
  writeb(IOAT_CHANCMD_SUSPEND, chan->reg_base + IOAT_CHANCMD_OFFSET(ver));
}

static inline void ioat_resume(struct ioat_chan_common *chan)
{
  u8 ver = chan->device->version;
  writeb(IOAT_CHANCMD_RESUME, chan->reg_base + IOAT_CHANCMD_OFFSET(ver));
}
```

Come è possibile notare, per modificare lo stato dei canali è necessario scrivere nel loro registro di controllo. I valori `IOAT_CHANCMD_SUSPEND` e `IOAT_CHANCMD_RESUME` sonno utilizzati per sospendere e risvegliare il canale descritto dall'oggetto `chan`.

4.2 Il Sottosistema DMA-CFTU

Il *Sottosistema DMA*, introdotto dalle tecnologie I/OAT, fornisce un'interfaccia generica per effettuare le copie asincrone tramite motore DMA, ma non implementa alcune funzionalità necessarie per eseguire trasferimenti di dati tra memoria utente e kernel. Per far fronte a questo problema si è estesa l'interfaccia dell'I/OAT, inglobando il vecchio *Sottosistema DMA* nel nuovo *Sottosistema DMA-CFTU*. Quest'ultimo espone un insieme di funzioni che permettono di creare politiche per la gestione dei canali DMA e di semplificare l'interazione dei client con il sottosistema.

4.2.1 Supporto alle operazioni di copia

Per consentire la copia dei dati di una pagina utente in un buffer del kernel, il *Sottosistema DMA-CFTU* introduce la funzione `dma_async_memcpy_pg_to_buf()`. Il frammento di codice sottostante mostra le operazioni principali eseguite dalla suddetta funzione che utilizza `dma_map_single()` e `dma_map_page()` per ottenere gli indirizzi fisici, relativi al buffer del kernel e alla pagina utente, da passare al motore DMA. La funzione `device_prep_dma_memcpy()` è utilizzata per creare il descrittore `tx` che

rappresenta la transazione di dati richiesta [1]. Per sottomettere l'operazione di copia al canale DMA si utilizza la funzione `tx_submit()`, che restituisce un `cookie`. Se la transazione è stata inserita nella coda dei lavori pendenti del motore DMA, il valore del campo `cookie` sarà positivo, mentre se si è incorsi in un errore (ad esempio quando la coda dei lavori del motore DMA è piena) suddetto campo avrà un valore negativo.

```
dma_cookie_t dma_async_memcpy_pg_to_buf(struct dma_chan *chan,
    struct page *page, unsigned int offset, void *kdata, size_t len)
{
  ........
  dma_dest = dma_map_single(dev->dev, kdata, len, DMA_FROM_DEVICE);
  dma_src = dma_map_page(dev->dev, page, offset, len, DMA_TO_DEVICE);
  tx = dev->device_prep_dma_memcpy(chan, dma_dest, dma_src, len, flags);
  ........
  cookie = tx->tx_submit(tx);
  return cookie;
  .........
}
```

Un trasferimento di dati può coinvolgere più pagine utente, quindi è stato necessario introdurre la struttura `request_copy` che consente di raccogliere e gestire tutte le transazioni di un trasferimento dati. Di seguito la definizione della struttura ed una descrizione di ognuno dei sui campi:

```
struct request_copy
{
  struct dma_chan *chan;
  void __user *user_base;
  void __kernel  *kernel_base;
  size_t len;
  struct page **pages;
  int nr_pages;
  dma_cookie_t  dma_cookie;
  enum dma_status status;
};
```

`chan` identifica il canale DMA che sarà coinvolto nel trasferimento. I campi `user_base` e `kernel_base` memorizzano gli indirizzi dei buffer kernel e utente coinvolti nell'operazione di copia e `len` è la dimensione in byte dei suddetti buffer. La lista `pages` serve per identificare tutte le pagine di memoria del buffer utente e `nr_pages` è la cardinalità di suddetta lista. Infine `dma_cookie` è il `dma_cookie_t` dell'ultima transazione sottomessa al motore DMA e `status` descrive lo stato del trasferimento.

[1] Trasferimento dati che coinvolge una solo pagina di memoria

Inoltre per supportare le politiche basate su priorità è stata realizzata la struttura `request_copy_prio` che è analoga alla struttura `request_copy`, l'unica differenza risiede nell'introduzione del un nuovo campo `priority`. Quest'ultimo permette di assegnare una priorità all'operazione di copia che sarà così eseguita su un determinato Canale DMA.

Durante una copia in memoria, tramite motore DMA, si deve evitare che le pagine utente siano swappate su disco. Per far fronte a questa spiacevole situazione, il *Sottosistema DMA-CFTU* mette a disposizione le seguenti funzioni:

```
int dma_pin_user_pages(struct request_copy *req_cp, int write, int force)

void dma_unpin_user_pages(struct request_copy *req_cp, int dirty)

int dma_pin_user_pages_prio(struct request_copy_prio *req_cp,
    int write, int force)

void dma_unpin_user_pages_prio(struct request_copy_prio *req_cp,
    int dirty)
```

La funzione `dma_pin_user_pages()` effettua il lock sulle pagine di memoria utente, `req_cp` è il descrittore del trasferimento, mentre `write` e `force` sono dei flag che descrivono la natura del locking richiesto. Il frammento di codice sottostante illustra le principali operazioni eseguite da `dma_pin_user_pages()` che nell'ordine sono:

a) il calcolo del numero di pagine di memoria utilizzate dal buffer utente;

b) l'allocazione in memoria (tramite la funzione `kmalloc()`) di una lista per tenere traccia delle pagine utente;

c) l'invocazione della funzione `get_user_page()` per recuperare le pagine utente e per eseguire il locking su queste;

d) la verifica che la funzione `get_user_page()` abbia bloccato tutte le pagine utente in memoria.

```
   int dma_pin_user_pages(struct request_copy *req_cp , int write, int force)
   {
     .....
(a)  req_cp->nr_pages = ((PAGE_ALIGN((unsigned long)req_cp->user_base +
(a)    req_cp->len) - ((unsigned long)req_cp->user_base & PAGE_MASK))
(a)    >> PAGE_SHIFT);
```

```
(b)   req_cp->pages = kmalloc((sizeof(struct page*) * req_cp->nr_pages),
(b)     GFP_KERNEL);
      .....
(c)   down_read(&current->mm->mmap_sem);
(c)   ret = get_user_pages(current, current->mm, (unsigned long)
(c)   req_cp->user_base, req_cp->nr_pages, write, force, req_cp->pages, NULL);
(c)   up_read(&current->mm->mmap_sem);

(d)   if (ret != req_cp->nr_pages)
(d)     goto unpin;
      .....
    }
```

Le pagine utente che vengono bloccate in memoria, al termine del trasferimento dati devono essere sbloccate con la funzione `dma_unpin_user_pages()`. Quest'ultima scandisce tutte le pagine contenute nella lista `req_cp->pages` e le sblocca con la funzione `page_cache_release()`. Inoltre, se le pagine sono state scritte vengono marcate come *dirty*. Prima di terminare, la funzione `dma_unpin_user_pages()` libera la memoria allocata per la lista `req_cp->pages`.

```
void dma_unpin_user_pages(struct request_copy *req_cp , int dirty)
{
  .....
  for (i = 0; (i < req_cp->nr_pages) && req_cp->pages[i] != NULL ; i++)
  {
    if(dirty)
    {
      set_page_dirty_lock(req_cp->pages[i]);
    }

    page_cache_release(req_cp->pages[i]);
  }
  kfree(req_cp->pages);
}
```

Le funzioni `dma_pin_user_pages_prio()` e `dma_unpin_user_pages_prio()` sono usate per i trasferimenti di dati basati su priorità. il loro funzionamento è analogo a quello delle funzioni sopra descritte[2], l'unica differenza consiste nel campo `struct request_copy_prio` passato in fase di invocazione.

[2] `dma_pin_user_pages()` e `dma_unpin_user_pages()`

4.2.2 Supporto ai Trasferimenti di dati

Per permettere alle varie *Politiche CFTU* di eseguire le operazioni di copia, sono state implementate quattro funzioni:

```
static int __dma_copy_from_user(struct request_copy *req_cpy_local)

static int __dma_copy_to_user(struct request_copy *req_cpy_local)

int __dma_copy_from_user_prio(struct request_copy_prio *req_cpy_local)

int __dma_copy_to_user_prio(struct request_copy_prio *req_cpy_local)
```

La funzione `__dma_copy_from_user()` copia i dati dallo spazio utente a quello kernel, ma a differenza della `dma_async_memcpy_pg_to_buf()`, supporta trasferimenti che coinvolgono più pagine di memoria. Il frammento di codice sottostante illustra le principali operazioni eseguite dalla funzione `__dma_copy_from_user()`, che nell'ordine sono:

a) il calcolo di quanti byte di dati della prima pagina utente bisogna trasferire (non è detto che il buffer utente inizi dal primo byte della pagina di memoria);

b) la copia dei dati di una pagina utente nella memoria del kernel tramite la funzione `dma_async_memcpy_pg_to_buf()`;

c) l'aggiornamento di `tmp_len` che tiene traccia di quanti byte devono essere ancora copiati. Se non ci sono più dati da trasferire si eseguono le operazioni del punto **e**;

d) l'aggiornamento degli indirizzi sorgente e destinazione per eseguire una nuova copia di dati tramite la funzione `dma_async_memcpy_pg_to_buf()`;

e) l'attesa che l'ultima transazione sia terminata, l'aggiornamento dello stato del trasferimento ed il ritorno dei byte che non sono stati copiati.

```
 static int __dma_copy_from_user(struct request_copy *req_cpy_local)
 {
    .....
(a)    byte_offset = ((unsigned long)req_cpy_local->user_base & ~PAGE_MASK);
(a)    copy = min_t(int, PAGE_SIZE - byte_offset, tmp_len);

    while(1)
    {
```

```
(b)         req_cpy_local->dma_cookie = dma_async_memcpy_pg_to_buf(
(b)           req_cpy_local->chan, req_cpy_local->pages[page_index],
(b)           byte_offset ,tmp_addr_kernel, copy);
            .....
(c)         tmp_len -= copy;
(c)         if(!tmp_len)
(c)           goto end_copy;

(d)         tmp_addr_user += copy;
(d)         tmp_addr_kernel += copy;
(d)         byte_offset = 0;
(d)         page_index++;
(d)         copy = min_t(int, PAGE_SIZE, tmp_len);
          }

(e)     end_copy:
(e)       req_cpy_local->status = dma_sync_wait(req_cpy_local->chan,
(e)         req_cpy_local->dma_cookie);

(e)       return tmp_len;
        }
```

La funzione `__dma_copy_to_user()` copia i dati dallo spazio kernel a quello utente, ma a differenza della `dma_async_memcpy_buf_to_pg()`, supporta trasferimenti che coinvolgono più pagine di memoria. Il frammento di codice sottostante illustra le principali operazioni eseguite dalla funzione `__dma_copy_to_user()`, che nell'ordine sono:

a) il calcolo di quanti byte di dati si possono copiare nella prima pagina utente (non è detto che il buffer utente inizi dal primo byte della pagina di memoria);

b) la copia dei dati dalla memoria kernel alla pagina utente tramite la funzione `dma_async_memcpy_buf_to_pg()`;

c) l'aggiornamento di `tmp_len` che tiene traccia di quanti byte devono essere ancora copiati. Se non ci sono più dati da trasferire si eseguono le operazioni del punto **e**;

d) l'aggiornamento degli indirizzi sorgente e destinazione per eseguire una nuova copia di dati tramite la funzione `dma_async_memcpy_buf_to_pg()`;

e) l'attesa che l'ultima transazione sia terminata, l'aggiornamento dello stato del trasferimento ed il ritorno dei byte che non sono stati copiati.

```
        static int __dma_copy_to_user(struct request_copy *req_cpy_local)
        {
          .....
(a)       byte_offset = ((unsigned long)req_cpy_local->user_base & ~PAGE_MASK);
(a)       copy = min_t(int, PAGE_SIZE - byte_offset, tmp_len);

          while(1)
          {
(b)         req_cpy_local->dma_cookie = dma_async_memcpy_buf_to_pg(
(b)           req_cpy_local->chan, req_cpy_local->pages[page_index],
(b)           byte_offset ,tmp_addr_kernel, copy);
            .....
(c)         tmp_len -= copy;
(c)         if(!tmp_len)
(c)           goto end_copy;

(d)         tmp_addr_user += copy;
(d)         tmp_addr_kernel += copy;
(d)         byte_offset = 0;
(d)         page_index++;
(d)         copy = min_t(int, PAGE_SIZE, tmp_len);
          }

(e)     end_copy:
(e)       req_cpy_local->status = dma_sync_wait(req_cpy_local->chan,
(e)         req_cpy_local->dma_cookie);

(e)       return tmp_len;
        }
```

Le funzioni `__dma_copy_from_user_prio()` e `__dma_copy_to_user_prio()` sono usate per i trasferimenti di dati basati su priorità, il loro funzionamento è analogo a quello delle funzioni sopra descritte[3], l'unica differenza consiste nel campo `struct request_copy_prio` passato in fase di invocazione.

4.2.3 Meccanismo trasparente per la gestione dei Canali DMA

Il *Sottosistema DMA-CTFU* introduce un insieme di funzioni generiche per l'inizializzazione e la gestione dei canali DMA, sfruttabili da qualsiasi *Politica CFTU*. Il comportamento di queste funzioni è specializzato in fase di configurazione del kernel. Naturalmente le *Politiche Dinamiche* non specializzano suddette funzioni, così da essere utilizzabili e rimovibili a sistema avviato.

[3] __dma_copy_from_user() e __dma_copy_to_user()

Nello specifico le funzioni generiche messe a disposizione dal *Sottosistema DMA-CTFU* sono:

1. `boot_init_dma_chan()` utilizzata per inizializzare le strutture `dma_chan_copy` [4];
2. `boot_complete_dma_chan()` utilizzata per collegare i canali DMA alle strutture `dma_chan_copy`[4] o per registrare un client al *Sottosistema DMA*[5];

La struttura dati `dma_chan_copy` rappresenta lo stato di un canale DMA e consta di due campi: `chan` che è un puntatore al descrittore del canale ed il flag `use` che rappresenta lo stato del canale (un valore diverso da zero indica che il canale è stato inizializzato dal sistema).

```
struct dma_chan_copy
{
  struct dma_chan *chan;
  short int use;
};
```

4.3 Politiche Statiche CFTU

Le *Politiche Statiche CFTU* integrano le funzionalità del motore DMA direttamente nel codice delle funzioni: `__copy_from_user()`, `__copy_to_user()`, `copy_from_user()` e `copy_to_user()`. Questo permette all'intero sistema di trasferire dati tra memoria kernel e utente senza sprecare preziose risorse del processore. In fase di configurazione del kernel è possibile selezionare una politica statica che così sarà utilizzata ad ogni avvio di sistema. Le operazioni classiche per i trasferimenti di dati tra memoria Kenel e utente sono utilizzate solo in fase di boot, quando i canali del motore DMA non sono ancora attivi.
L'estensione DMA-CFTU implementa due *Politiche Statiche*:

- **Exclusive Channel**
- **Shared Channel**

[4]Solo per le politiche che gestiscono *private channel*
[5]Solo per le politiche che gestiscono *public channel*

La politica *Exclusive Channel* permette di associare in modo esclusivo (*private channel*) un canale DMA ad ogni CPU, così da sfruttare il bilanciamento del carico effettuato dal sistema sui processori. Mentre, nella politica *Shared Channel* i canali DMA sono condivisi da tutti i client, quindi non ci sono vincoli circa il loro utilizzo ed il bilanciamento del carico sui canali è eseguito dal driver del motore DMA.

4.3.1 Gestione dei canali DMA

Le *Politiche Statiche CFTU* specializzano le funzioni `boot_init_dma_chan()` e `boot_complete_dma_chan()`, fornite dal *Sottosistema DMA-CFTU*, per inizializzare i canali DMA alla conclusione della fase di boot del kernel [6].

La politica *Exclusive Channel*, per associare i canali DMA alle CPU di sistema, utilizza l'oggetto `percpu_chan_copy` per-CPU di tipo `struct dma_chan_copy` che viene inizializzato dalla funzione `boot_init_dma_chan()`. Quest'ultima deve impostare i flag `use` dell'oggetto `percpu_chan_copy` a zero per segnalare che i canali DMA non sono stati ancora attivati dal driver. Quando il motore DMA diventa disponibile, viene invocata la funzione `boot_complete_dma_chan()` che richiede i canali DMA in modo esclusivo (*private channel*) e li associa ad ogni copia di `percpu_chan_copy`. Inoltre la funzione `boot_complete_dma_chan()` impostare su ogni *private channel* la tipologia di trasferimento `DMA_MEMCPY`.

```
#ifdef CONFIG_EXCL_COPY_TO_FROM_IOATDMA
#include <linux/percpu.h>

DEFINE_PER_CPU(struct dma_chan_copy , percpu_chan_copy);
EXPORT_PER_CPU_SYMBOL(percpu_chan_copy);

void __init boot_init_dma_chan(void)
{
  uint i;
  struct dma_chan_copy *cpu_chan;

  for_each_present_cpu(i)
  {
    cpu_chan = &per_cpu(percpu_chan_copy , i);
    cpu_chan->use = 0;
    cpu_chan->chan = NULL;
```

[6] Alla terminazione della funzione *kernel_ init_ freeable()*

```
  }
}

void __init boot_complete_dma_chan(void)
{
  uint i;
  dma_cap_mask_t mask;
  struct dma_chan *chan;
  struct dma_chan_copy *cpu_chan;

  dma_cap_zero(mask);
  dma_cap_set(DMA_MEMCPY, mask);

  for_each_present_cpu(i)
  {
    cpu_chan = &per_cpu(percpu_chan_copy , i);
    chan = dma_request_channel(mask, NULL , NULL);
    .....
  }
  .....
  return;
}
#endif
```

La politica *Shared Channel* non richiede i canali DMA in modo esclusivo, quindi non necessita delle strutture utilizzate dalla politica *Exclusive Channel*. Infatti la funzione `boot_init_dma_chan()` non è implementata e l'unica operazione eseguita dalla funzione `boot_complete_dma_chan()` è quella di registrarsi al *Sottosistema DMA* tramite la `copy_dmaengine_get()`.

```
#ifdef CONFIG_SHARED_COPY_TO_FROM_IOATDMA

void __init boot_init_dma_chan(void)
{

}

void __init boot_complete_dma_chan(void)
{
  copy_dmaengine_get();
}
#endif
```

4.3.2 Trasferimenti di Dati tramite DMA

Le *Politiche Statiche CFTU* implementano una funzione generica `copy_dma_get_channel_boot()` utilizzata per richiedere un canale DMA. Questa funzione varia il proprio comportamento in base alla politica statica usata dal sistema. Infatti grazie alle opzioni di configurazioni inserite nel kernel, è

stato possibile utilizzare la funzione `copy_dma_get_channel_boot()` per richiedere:

- un canale DMA di tipo *private*, se il sistema utilizza la politica statica *Exclusive Channel*. In questo caso sarà definita la macro `CONFIG_EXCL_COPY_TO_FROM_IOATDMA`.
- un canale DMA di tipo *public*, se il sistema utilizza la politica statica *Shared Channel*. In questo caso sarà definita la macro `CONFIG_SHARED_COPY_TO_FROM_IOATDMA`.

Naturalmente la funzione `copy_dma_get_channel_boot()` ritornerà `NULL`, se invocata durante la fase di boot del kernel, quando ancora i canali DMA non sono inizializzati.

```
struct dma_chan *copy_dma_get_channel_boot(void)
{
  struct dma_chan *chan;
chan = NULL;

  #ifdef CONFIG_EXCL_COPY_TO_FROM_IOATDMA
    chan = get_cpu_var(percpu_chan_copy).chan;
    put_cpu_var(percpu_chan_copy);
  #endif

  #ifdef CONFIG_SHARED_COPY_TO_FROM_IOATDMA
    if(copy_dmaengine_ref_count > 0)
    {
    chan = dma_find_channel(DMA_MEMCPY);
    }
  #endif
  .....
  return chan;
}
```

Grazie al meccanismo appena descritto, è possibile implementare delle funzioni per il trasferimento di dati, tramite motore DMA, che sono indipendenti dal tipo di *Politica Statica CFTU*.

La funzione `dma_copy_to_user()` effettua un trasferimento di dati, descritto dall'oggetto `req_cpy_local`, dalla memoria kernel a quella utente. Il frammento di codice sottostante mostra le principali operazioni eseguite da questa funzione, che nell'ordine sono:

a) Controllo della richiesta di trasferimento.

b) Richiesta di un canale DMA. La funzione `copy_dma_get_channel_boot()` può fallire (`req_cpy_local->chan == NULL`) per due ragioni, quali:

- tutti i canali DMA sono occupati;
- i canali DMA non sono stati ancora inizializzati.

In entrambi questi casi la funzione `dma_copy_to_user()` termina ritornando un errore (`-EBUSY`).

c) Lock in scrittura (`PIN_WRITE`) delle pagine di memoria utente.

d) Copia del buffer del kernel nelle pagine utente.

e) Unlock delle pagine utente che vengono marcate *dirty* (`SET_DIRTY_ON`) prima di essere sbloccate.

f) Ritorno del numero di byte non copiati.

```
      int dma_copy_to_user(struct request_copy *req_cpy_local)
      {
        int ret = 0;

(a)     if( req_cpy_local->len > 0)
        {
(b)       req_cpy_local->chan = copy_dma_get_channel_boot();

(b)       if (req_cpy_local->chan == NULL )
(b)         return -EBUSY;

(c)       ret = dma_pin_user_pages(req_cpy_local , PIN_WRITE , 0) ;
          .....
(d)       ret = __dma_copy_to_user(req_cpy_local);
(e)       dma_unpin_user_pages(req_cpy_local , SET_DIRTY_ON);
        }
(f)     return ret;
    }
```

La funzione `dma_copy_from_user()` effettua un trasferimento di dati, descritto dall'oggetto `req_cpy_local`, dalla memoria utente a quella kernel. Il frammento di codice sottostante mostra le principali operazioni eseguite da questa funzione, che nell'ordine sono:

a) Controllo della richiesta di trasferimento.

b) Richiesta di un canale DMA. La funzione `copy_dma_get_channel_boot()` può fallire (`req_cpy_local->chan == NULL`) per due ragioni, quali:

- tutti i canali DMA sono occupati;
- i canali DMA non sono stati ancora inizializzati.

In entrambi questi casi la funzione `dma_copy_from_user()` termina ritornando un errore (`-EBUSY`).

c) Lock in letture (`PIN_READ`) delle pagine di memoria utente.

d) Copia delle pagine utente nel buffer del kernel.

e) Unlock delle pagine utente.

f) Ritorno del numero di byte non copiati.

```
      int dma_copy_from_user(struct request_copy *req_cpy_local)
      {
        int ret = 0;

(a)     if( req_cpy_local->len > 0)
        {
(b)       req_cpy_local->chan = copy_dma_get_channel_boot();

(b)       if (req_cpy_local->chan == NULL )
(b)         return -EBUSY;

(c)       ret = dma_pin_user_pages(req_cpy_local , PIN_READ , 0);
          .....
(d)       ret = __dma_copy_from_user(req_cpy_local);
(e)       dma_unpin_user_pages(req_cpy_local , SET_DIRTY_OFF);
        }
(f)     return ret;
      }
```

4.3.3 Modifica delle funzioni del kernel

Le funzioni `dma_copy_to_user()` e `dma_copy_from_user()` sono indipendenti dal tipo di *Politica Statica CFTU* utilizzata dal sistema, quindi possono essere integrate nelle funzioni di copia del kernel con estrema facilità. Per favorire questa integrazione è necessario sviluppare un meccanismo che permetta di effettuare:

- trasferimenti di dati in maniera tradizionale (tramite processore) durante la fase di boot del kernel;
- trasferimenti di dati tramite motore DMA, dopo la fase di boot del kernel, quando i canali DMA sono inizializzati.

Nel seguito saranno descritte le modifiche apportate alle funzioni `copy_from_user()` e `copy_to_user()`, mentre non saranno illustrate quelle relative alle `__copy_from_user()` e `__copy_to_user()` perché identiche.

copy_to_user()

Se il sistema non utilizza le *Politiche Statiche CFTU*, la funzione `copy_to_user()` lavorerà in maniera tradizionale invocando la `copy_to_user_pure()`. Quest'ultima funzione è la `copy_to_user()` originaria del kernel, che effettua i trasferimenti di dati tramite CPU. Mentre, se è attiva una *Politica Statica CFTU*, la funzione `copy_to_user()` invocherà la `copy_to_user_dma_pure()` per effettuare l'operazione di copia tramite motore DMA.

```
static __always_inline __must_check
int copy_to_user(void __user *dst, const void *src, unsigned size)
{
  int r;

  #if defined(CONFIG_SHARED_COPY_TO_FROM_IOATDMA) ||
      defined(CONFIG_EXCL_COPY_TO_FROM_IOATDMA)
    r = copy_to_user_dma_pure(dst, src, size);
  #else
    r = copy_to_user_pure(dst, src, size);
  #endif

  return r ;
}
```

La funzione `copy_to_user_dma_pure()` inizializza `req_cpy_local`, che è il descrittore del trasferimento di dati, e lo usa per invocare la `dma_copy_to_user()`. Quest'ultima funzione effettua la copia dei dati tramite DMA e può fallire per i seguenti motivi:

- errori interni al Motore DMA che non hanno permesso il trasferimento;
- tutti i canali DMA sono occupati;

- i canali DMA ancora non sono inizializzati, quindi il sistema è ancora in fase di boot del kernel.

Se la `dma_copy_to_user()` fallisce, viene invocata la funzione `copy_to_user_pure()` e l'operazione di trasferimento è eseguita in maniera tradizionale. Infine nel caso in cui non ci siano errori, la funzione `dma_copy_to_user()` restituisce il numero di byte non copiati. Suddetto valore sarà utilizzato anche per il ritorno della funzione `copy_to_user_dma_pure()`.

```
static __always_inline __must_check
int copy_to_user_dma_pure(void __user *dst, const void *src, unsigned size)
{
  int r;
  struct request_copy req_cpy_local;
  init_req_copy(&req_cpy_local , dst , (void *)src , (size_t) size);
  r = dma_copy_to_user(&req_cpy_local);

  if(r < 0 ||req_cpy_local.status != DMA_SUCCESS)
    goto error_dma;

  return r;

  error_dma:
    r = copy_to_user_pure(dst, src, size);
  return r;
}
```

copy_from_user()

Se il sistema non utilizza le *Politice Statiche CFTU*, la funzione `copy_from_user()` lavorerà in maniera tradizionale invocando la `copy_from_user_pure()`. Quest'ultima funzione è la `copy_from_user()` originaria del kernel, che effettua i trasferimenti di dati tramite CPU. Mentre, se è attiva una *Politica Statica CFTU*, la funzione `copy_from_user()` invocherà la `copy_from_user_dma_pure()` per effettuare l'operazione di copia tramite motore DMA.

```
static inline unsigned long __must_check
copy_from_user(void *to, const void __user *from, unsigned long n)
  {
    #if defined(CONFIG_SHARED_COPY_TO_FROM_IOATDMA) ||
        defined(CONFIG_EXCL_COPY_TO_FROM_IOATDMA)
      n = copy_from_user_dma_pure(to, from, n);
    #else
      n = copy_from_user_pure(to, from, n);
    #endif
```

```
  return n;
}
```

La funzione `copy_from_user_dma_pure()` inizializza `req_cpy_local`, che è il descrittore del trasferimento di dati, e lo usa per invocare la `dma_copy_from_user()`. Quest'ultima funzione effettua la copia dei dati tramite DMA e può fallire per i seguenti motivi:

- errori interni al Motore DMA che non hanno permesso il trasferimento;
- tutti i canali DMA sono occupati;
- i canali DMA ancora non sono inizializzati, quindi il sistema è ancora in fase di boot del kernel.

Se la `dma_copy_from_user()` fallisce, viene invocata la funzione `copy_from_user_pure()` e l'operazione di trasferimento è eseguita in maniera tradizionale. Infine nel caso in cui non ci siano errori, la funzione `dma_copy_from_user()` restituisce il numero di byte non copiati. Suddetto valore sarà utilizzato anche per il ritorno della funzione `copy_from_user_dma_pure()`.

```
static inline unsigned long __must_check
copy_from_user_dma_pure(void *to, const void __user *from, unsigned long n)
{
  int r ;
  struct request_copy req_cpy_local;
  init_req_copy(&req_cpy_local , (void *) from, to , (size_t) n);
  r = dma_copy_from_user(&req_cpy_local);

  if(r < 0 || req_cpy_local.status != DMA_SUCCESS)
    goto error_dma;

  return r;

error_dma:
  r = copy_from_user_pure(to, from, n);
  return r;
}
```

4.4 Politiche Dinamiche CFTU

Le *Politiche Dinamiche CFTU* mettono a disposizione delle API per eseguire trasferimenti di dati tra memoria kernel e utente tramite motore DMA. Le funzioni di copia del kernel non vengono modificate, quindi è possibile

cambiare e disabilitare le politiche di questa classe a sistema avviato. Inoltre le API messe a disposizione da queste politiche sono utilizzate dopo la fase di boot del kernel, così non è stato necessario specializzare le funzioni di `boot_init_dma_chan()` e `boot_complete_dma_chan()`.

L'estensione DMA-CFTU implementa 3 politiche dinamiche:

1. **Exclusive Channel**
2. **Shared Channel**
3. **Priority Channel**

Le politiche dinamiche *Exclusive Channel* ed *Shared Channel* hanno lo stesso comportamento delle loro omonime statiche, l'unica differenza risiede nell'interfaccia che espongo.

Di seguito sono mostrate le funzioni messe a disposizione dalla politica dinamica *Exclusive Channel*:

```
void start_policy_excl(void)

void stop_policy_excl(void)

int _dma_copy_from_user_excl(struct request_copy *req_cpy_local)

int _dma_copy_to_user_excl(struct request_copy *req_cpy_local)
```

segue una breve descrizione:

- `start_policy_excl()`: richiede i canali DMA in modo esclusivo e li associa ad un oggetto per-CPU.
- `stop_policy_excl()`: rilascia i canali DMA.
- `_dma_copy_from_user_excl()`: effettua un trasferimento di dati, descritto da `req_cpy_local`, dallo spazio di memoria utente a quello kernel [7]. L'operazione di copia sarà sottomessa al canale DMA, di tipo *private*, che è associato con la CPU del processo corrente.

[7] Questa funzione non effettua il lock e l'unlock delle pagine di memoria utente, quindi è necessario l'uso di `dma_pin_user_pages()` e `dma_unpin_user_pages()` prima e dopo la sua invocazione.

- `_dma_copy_to_user_excl()`: effettua un trasferimento di dati, descritto da `req_cpy_local`, dallo spazio di memoria kernel a quello utente[7]. L'operazione di copia sarà sottomessa al canale DMA, di tipo *private*, che è associato con la CPU del processo corrente.

Le funzioni messe a disposizione dalla politica dinamica *Shared Channel* sono mostrate nel codice sottostante:

```
void start_policy_sh(void)

void stop_policy_sh(void)

int _dma_copy_from_user_sh(struct request_copy *req_cpy_local)

int _dma_copy_to_user_sh(struct request_copy *req_cpy_local)
```

segue una breve descrizione:

- `start_policy_sh()`: si registra al *Sottosistema DMA*.
- `stop_policy_sh()`: cancella la registrazione al *Sottosistema DMA*.
- `_dma_copy_from_user_sh()`: effettua un trasferimento di dati, descritto da `req_cpy_local`, dallo spazio di memoria utente a quello kernel [8]. L'operazione di copia sarà sottomessa ad un canale DMA di tipo *public*.
- `_dma_copy_to_user_sh()`: effettua un trasferimento di dati, descritto da `req_cpy_local`, dallo spazio di memoria kernel a quello utente [8]. L'operazione di copia sarà sottomessa ad un canale DMA di tipo *public*.

4.4.1 La Politica Priority Channel

Le *Politiche CFTU* illustrate fino ad ora, non forniscono un meccanismo per discrimine l'assegnazione delle risorse DMA in base alla criticità dell'operazione. Così, per sopperire a questa mancanza, è stata sviluppata la politica *Priority Channel* che consente di assegnare una priorità all'operazione di trasferimento, la quale sarà presa in carico dal canale DMA che gestisce

[8] Questa funzione non effettua il lock e l'unlock delle pagine di memoria utente, quindi è necessario l'uso di `dma_pin_user_pages()` e `dma_unpin_user_pages()` prima e dopo la sua invocazione.

quella classe di priorità. Naturalmente il solo partizionamento dei canali non garantisce che le operazioni critiche siano gestite nel minor tempo possibile. Infatti tutti i canali DMA condividono lo stesso bus, quindi gestire i task di diverse classi di priorità su canali distinti non contribuisce a risolvere tale problema. Così è stato implementato un meccanismo che permette di assegnare, in modo esclusivo, il bus del motore DMA a determinati canali per un certo intervallo di tempo. Tramite le funzioni `ioat_suspend()` e `ioat_resume()` è stato possibile sospende tutti i canali DMA a bassa priorità, per un certo intervallo di tempo, in modo da far utilizzare il bus esclusivamente ai canali che gestiscono task ad alta priorità.

La politica *Priority Channel* divide i canali DMA in due classi:

- *CH-HIGH*: canali ad alta priorità;
- *CH-LOW*: canali a bassa priorità.

Durante ogni intervallo di tempo $T_c = t_h + t_l$, i canali *CH-LOW* vengono sospesi per un tempo t_h, mentre i canali *CH-HIGH* sono sempre attivi. Inoltre, per non avere problemi di risoluzione temporale si utilizza un timer ad alta risoluzione, che gestire il meccanismo della sospensione/attivazione dei canali DMA.

Per descrivere lo stato dei canali DMA è stata introdotta la struttura `list_dma_chan_prio`:

```
struct list_dma_chan_prio
{
  struct dma_chan *chan_prio[DMA_MAX_CHAN_PRIO];
  unsigned short int priority[DMA_MAX_CHAN_PRIO];
  unsigned short int size;
};
```

che consta dei seguenti campi:

- `chan_prio`: una lista dei canali disponibili nel sistema;
- `priority`: le priorità associate a `chan_prio`;
- `size`: la cardinalità di `chan_prio`.

Per gestire il meccanismo di sospensione/attivazione dei canali DMA è stata introdotta la struttura `priority_timer`, così definita:

```
typedef void (*prio_ptr_func)(struct list_dma_chan_prio * ,
  unsigned short int);

struct priority_timer
{
  struct list_dma_chan_prio *list_chan;
  struct hrtimer timer;
  unsigned short int tick;
  unsigned long overruns_2;
  ktime_t quantum[MAX_QUANTUM];
  prio_ptr_func prio_func_array[MAX_QUANTUM];
};
```

di seguito il significato dei campi:

- `list_chan`: un oggetto di tipo `struct list_dma_chan_prio` ;
- `timer`: un oggetto `hrtimer` usato per rappresentare un timer ad alta risoluzione;
- `tick`: un contatore che viene aggiornato ad ogni scadenza del timer ed è utilizzato per selezionare un elemento dell'array `prio_func_array`;
- `overruns_2`: registra quanti *overruns* > 1 ci sono stati durante il tempo di vita della politica *Priority Channel*;
- `quantum`: gli intervalli di tempo t_h e t_l.
- `prio_func_array`: contiene i puntatori alle funzioni utilizzate per sospendere e attivare i canali DMA.

La politica *Priority Channel* viene avviata invocando la funzione `start_policy_prio2()`, che inizializza gli oggetti `list_dma_chan_prio` e `priority_timer` ed attiva un `hrtimer`.
Nel seguito è mostrato come la funzione `start_policy_prio2()` inizializza ed attiva un `hrtimer`:

```
struct list_dma_chan_prio list_chan_prio;
static struct priority_timer prio_timer;

static void suspend_chan_prio(struct list_dma_chan_prio *list_prio,
  unsigned short int priority);

static void release_dma_chan_prio(void);

int start_policy_prio2(long nanosecs_high , long nanosecs_low)
```

```
{
  .....
  init_dma_chan_prio();
  .....
  prio_timer.list_chan = &list_chan_prio;
  prio_timer.tick = (unsigned long long) MAX_QUANTUM;
  prio_timer.overruns_2 = 0;
  prio_timer.quantum[0] = ktime_set( 0 , nanosecs_high);
  prio_timer.quantum[1] = ktime_set( 0 , nanosecs_low);
  prio_timer.prio_func_array[0] = suspend_chan_prio;
  prio_timer.prio_func_array[1] = resume_chan_prio;

  hrtimer_init(&prio_timer.timer, CLOCK_MONOTONIC, HRTIMER_MODE_REL);
  prio_timer.timer.function = prio_hrtimer_callback;
  hrtimer_start(&prio_timer.timer, prio_timer.quantum[0],
    HRTIMER_MODE_REL);
  .....
}
```

gli argomenti `nanosecs_high` e `nanosecs_low` definiscono gli intervalli di tempo t_h e t_l e vengono memorizzati nell'oggetto *prio_ timer* usato per definire il comportamento del timer ad ogni scadenza. Le funzioni `suspend_chan_prio` e `resume_chan_prio` permettono di sospendere e riattivare i canali DMA. L'`hrtimer`, identificato dall'oggetto `prio_timer.timer`, è inizializzato invocando la funzione `hrtimer_init()`, terminata la quale si procede con l'assegnazione della funzione di *callback* `prio_hrtimer_callback()` invocata ad ogni scadenza. Infine la funzione `hrtimer_start()` attiva `prio_timer.timer` nel sistema.

La funzione `prio_hrtimer_callback()` viene invocata ad ogni scadenza del timer ed è utilizzata per sospendere/attivare i canali DMA e per riattivare il timer. Nel seguito è mostrato il codice della funzione:

```
static enum hrtimer_restart prio_hrtimer_callback(struct hrtimer *timer)
{
  struct priority_timer *prio;
  u64 ret;

  prio = container_of(timer, struct priority_timer , timer);
  prio->tick = prio->tick % MAX_QUANTUM;
  prio->prio_func_array[prio->tick](prio->list_chan , DMA_HIGH_PRIO);
  ret = hrtimer_forward_now(timer, prio->quantum[prio->tick]);
  ++prio->tick;
  .....
  return HRTIMER_RESTART;
}
```

`timer` è il puntatore all'oggetto `hrtimer` della funzione di *callbak*, quindi utilizzando la macro `container_of()` è possibile recuperare l'oggetto

prio_timer, che conterrà le informazioni di stato della politica *Priority Channel*. Il valore di prio->tick è utilizzato per invocare la funzione prio_func_array[prio->tick](), che attiva/sospende i canali DMA e per determinare la prossima scadenza (prio->quantum[prio->tick]). Infine la funzione prio_hrtimer_callback() imposta il nuovo *expires* del timer invocando hrtimer_forward_now(), incrementa il valore di prio->tick e termina.

Le restanti funzioni messe a disposizione dalla politica *Priority Channel* sono riportate nel frammento di codice sottostante.

```
void stop_policy_prio(void)

_dma_copy_from_user_prio(struct request_copy_prio *req_cpy_local)

__dma_copy_to_user_prio(struct request_copy_prio *req_cpy_local)
```

segue una breve descrizione delle suddette:

- stop_policy_prio(): utilizzata per interrompere la politica *Priority Channel*
- _dma_copy_from_user_prio(): effettua un trasferimento di dati, descritto da req_cpy_local, dallo spazio di memoria utente a quello kernel. Il campo priority, locato nell'oggetto req_cpy_local, è utilizzato per determinare su quale classe di canali DMA verrà eseguita l'operazione di copia[9].
- _dma_copy_to_user_prio(): effettua un trasferimento di dati, descritto da req_cpy_local, dallo spazio di memoria kernel a quello utente. Il campo priority, locato nell'oggetto req_cpy_local, è utilizzato per determinare su quale classe di canali DMA verrà eseguita l'operazione di copia [9].

[9] Questa funzione non effettua il lock e l'unlock delle pagine di memoria utente, quindi è necessario l'uso di dma_pin_user_pages_prio() e dma_unpin_user_pages_prio() prima e dopo la sua invocazione.

Capitolo 5

Tool per l'Analisi delle Prestazioni

Per analizzare e confrontare le prestazioni di un sistema che utilizza l'*Estensione DMA-CFTU*, è stato necessario sviluppare un insieme di strumenti che permettessero di:

- studiare il comportamento delle funzioni di copia del kernel;
- analizzare il tempo di esecuzione dei trasferimenti di dati eseguiti tramite CPU e DMA;
- analizzare la larghezza di banda del sistema per i trasferimenti di dati eseguiti tramite CPU e DMA;
- studiare il fenomeno della *Cache Pollution*.

Nel seguito della trattazione, verranno descritti gli strumenti che permettono di effettuare questa analisi sul sistema.

5.1 Il Sistema di Profiling

Il *profiling* è un processo di analisi volto a misurare il tempo di esecuzione delle singole funzioni o componenti di sistema, al fine di individuare i blocchi di istruzioni computazionalmente più pesanti. Inoltre, a differenza del *benchmarking*, consente di monitorare il codice in ambienti di esecuzione reali così da ottenere misurazioni più significative. Quindi per quanto esposto sopra, si è deciso di implementare un *Sistema di Profiling* atto a studiare il comportamento delle seguenti funzioni di copia del kernel:

- `__copy_from_user();`
- `__copy_to_user();`
- `copy_from_user();`
- `copy_to_user()`.

Il codice di ognuna di queste funzione è stato instrumentato così ad ogni loro invocazione è possibile ottenere le seguenti informazioni:

- quante volte la funzione è stata invocata dall'avvio del sistema;
- il numero di byte copiato;
- l'id della CPU che ha iniziato l'esecuzione della funzione;
- l'id della CPU che ha terminato l'esecuzione della funzione;
- il tempo di esecuzione;
- l'indirizzo di ritorno della funzione;
- il tipo di trasferimento eseguito (DMA/CPU).

In Figura 5.1 sono mostrati i vari componenti del *Sistema di Profiling* e come questi interagiscono tra loro e con il sistema. Ogni funzione instrumentata, prima di terminare, memorizza le sue informazioni di profiling in un elemento di un buffer circolare per-CPU. Suddetti buffer sono allocati in fase di boot del kernel e utilizzano una *discard policy* di tipo *Wine*, in modo da non sovrascrivere i dati a buffer pieno. Per esportare i dati di profiling nello spazio utente, è stato implementato un driver virtuale che crea una serie di device file ognuno dei quali è associato ad un buffer circolare. I demoni così, per recuperare le misure di profiling, effettuano ciclicamente la chiamata di sistema `read` sul device file associato al buffer circolare di loro interesse.

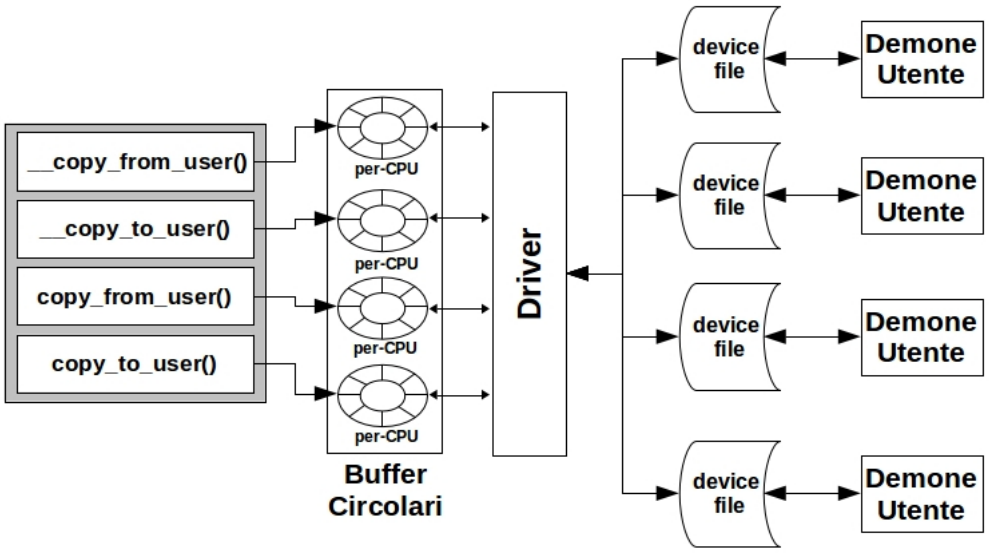

Figura 5.1: Architettura del Sistema di Profiling

Su ogni device file è stata implementata la *file operation* `read()` utilizzata per copiare in un buffer utente, passato in fase di invocazione dal demone, un numero specificato di misurazioni. Nello specifico, la copia dei dati nel buffer utente è effettuata invocando la funzione `copy_to_user_pure()`, ossia una versione della `copy_to_user()` non instrumentata. In questo modo il sistema di *Sistema di Profiling* evita di raccogliere misure sulla sua esecuzione che altererebbero l'analisi sul sistema.

Questo tipo di *profiling* può essere eseguito su:

- sistemi che operano in maniera standard, ossia che eseguono i trasferimenti di dati tra memoria kernel e utente tramite CPU. In questo caso al termine della misurazione si avranno i tempi di copia della CPU.
- Sistemi che utilizzano una delle Politiche Statiche precedentemente introdotte. In questo caso al termine della misurazione si avranno i tempi di copia del motore DMA.

5.2 Il Micro-Benchmark Execution Time

Il *Sistema di Profiling* sopra descritto ha un limite: quello di inserire, nelle misurazioni relative ai trasferimenti DMA, i tempi di *lock/unlock* delle pagine utente. Per ovviare a questo problema, è stato implementato il

Micro-Benchmark Execution Time che esegue un controllo preventivo della regione di memoria user, allo scopo di effettuare il *lock* prima della misura e l'*unlock* dopo.

Il *Micro-Benchmark Execution Time* permette di misurare il tempo di esecuzione di una determinata operazione di trasferimento. Le configurazioni di sistema supportate sono:

- *CPU*: l'operazione di trasferimento è effettuata tramite CPU;
- *DMA Excl*: l'operazione di trasferimento è effettuata utilizzando le API della politica dinamica *Exclusive Channel*;
- *DMA Sh*: l'operazione di trasferimento è effettuata utilizzando le API della politica dinamica *Shared Channel*.

La Tabella 5.1. mostra le operazioni di trasferimento supportate dalle varie configurazioni di sistema.

CPU	**DMA Excl**	**DMA Sh**
`copy_from_user()`	`_dma_copy_from_user_excl()`	`_dma_copy_from_user_sh()`
`copy_from_user()`^{nc}	`_dma_copy_from_user_excl()`^{nc}	`_dma_copy_from_user_sh()`^{nc}
`copy_to_user()`	`_dma_copy_to_user_excl()`	`_dma_copy_to_user_sh()`
`copy_to_user()`^{nc}	`_dma_copy_to_user_excl()`^{nc}	`_dma_copy_to_user_sh()`^{nc}

Tabella 5.1: Operazioni supportate dal Micro-Benchmark Exectution Time

Al fine di studiare gli effetti della cache sui trasferimenti di dati, il *Micro-Benchmark Execution Time* mette a disposizione le operazioni ^{nc} (*no cache*). Suddette operazioni si differenziano da quelle tradizionali, poiché garantiscono che i dati coinvolti nel trasferimento non siano in cache.

Il *Micro-Benchmark Execution Time*, in base alla configurazione di sistema, effettuerà le operazioni su una CPU o su un canale DMA e le relative misurazioni saranno effettuate in spazio kernel. Al termine del test verrà effettuata una sintesi delle misurazioni, in modo da calcolare gli indici prestazionali di ogni tupla < *operazione* , *dimensione del blocco di dati* >. Segue l'elenco di suddetti indici:

- la media dei tempi di esecuzione;

- la mediana dei tempi di esecuzione;
- il minimo dei tempi di esecuzione;
- il massimo dei tempi di esecuzione;
- la varianza dei tempi di esecuzione.

Il *Micro-Benchmark Execution Time*, come mostrato in Figura 5.1, è costituito da due componenti: un driver virtuale e un demone locato nello spazio utente. Una volta determinata la configurazione di sistema che si vuole testare, il driver può procedere alla creazione dei device file e alla specializzazione del loro comportamento. Ogni operazione è associata ad un device file, che è usato dal demone per sottomettere il test al driver.

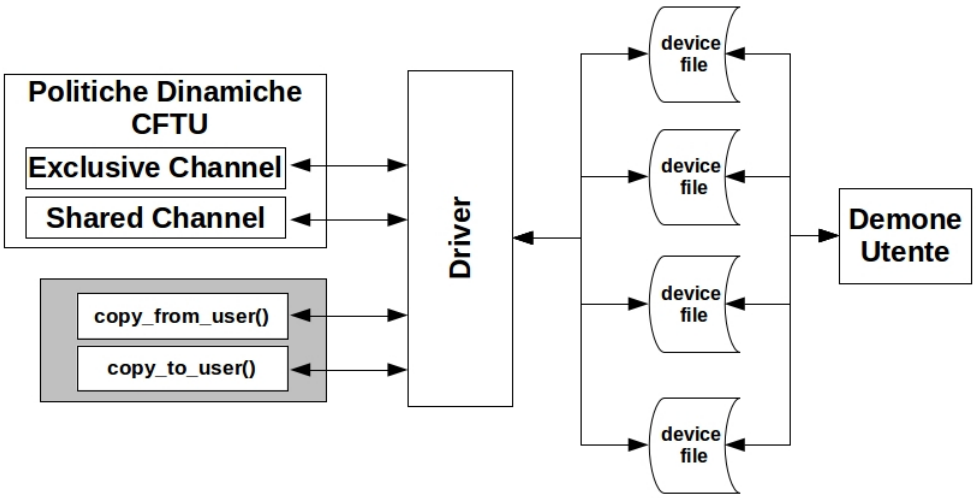

Figura 5.2: Architettura del Micro-Benchmark Execution Time

Grazie all'implementazione di determinate *file operation* dei device file, è stato possibile creare un meccanismo che consentisse di:

- definire il test da effettuare in spazio kernel, tramite la chiamata di sistema `write`;
- ricevere i dati delle misurazioni, tramite la chiamata di sistema `read`.

In particolare, la chiamata `write` viene invocata dal demone per inviare al driver l'indirizzo e la dimensione del buffer utente su cui verrà eseguito il test. La funzione `write` eseguirà le misurazioni sull'operazione di trasferimento

a lei associata[1], salverà in memoria le misure e terminerà. Così il demone, per recuperare suddette misure, invocherà la funzione `read` che le copierà in memoria utente.

5.3 Micro-Benchmark Bandwidth

Il *Micro-Benchmark Bandwidth* è utilizzato per misurare la banda relativa ai trasferimenti di dati, che sono eseguiti in parallelo su tutte le risorse del sistema (CPU o canali DMA in base alla configurazione di sistema). Questo micro-benchmark, a differenza del precedente, può essere utilizzato anche per testare le operazioni fornite dalla politica dinamica *Priority Channel*. Infatti, misurando la velocità di trasferimento dei diversi canali DMA, è possibile comprendere il comportamento del/dei canale/i ad alta priorità rispetto ai restanti.

Le configurazioni di sistema supportate sono:

- *CPU*: l'operazione di trasferimento è effettuata tramite CPU;
- *DMA Excl*: l'operazione di trasferimento è effettuata utilizzando le API della politica dinamica *Exclusive Channel*;
- *DMA Sh*: l'operazione di trasferimento è effettuata utilizzando le API della politica dinamica *Shared Channel*;
- *DMA Prio*: l'operazione di trasferimento è effettuata utilizzando le API della politica dinamica *Priority Channel*.

La Tabella 5.2 mostra le operazioni di trasferimento supportate dalle varie configurazioni di sistema.

	Op_1	Op_2	Op_3
CPU	`copy_from_user()`	`copy_to_user()`	`Op_1 + Op_2`
DMA Excl	`_dma_copy_from_user_excl()`	`_dma_copy_to_user_excl()`	`Op_1 + Op_2`
DMA Sh	`_dma_copy_from_user_sh()`	`_dma_copy_to_user_sh()`	`Op_1 + Op_2`
DMA Prio	`_dma_copy_from_user_prio()`	`_dma_copy_to_user_prio()`	`Op_1 + Op_2`

Tabella 5.2: Operazioni supportate dal Micro-Benchmark Exectution Time

[1]Le operazioni di *lock/unlock* delle pagine di utente, sono eseguite al di fuori della misurazione

Il *Micro-Benchmark Bandwidth*, in base alla configurazione di sistema, effettuerà le operazioni coinvolgendo tutte le CPU o tutti i canale DMA. Al termine del test saranno disponibili gli indici prestazionali di ogni tupla $<$ *operazione*, *dimensione del blocco di dati*, *canale* [2] $>$. Segue l'elenco di suddetti indici:

- il tempo di esecuzione [3];
- la larghezza di banda.

Il *Micro-Benchmark Bandwidth*, come mostrato in Figura 5.3 è costituito da due componenti: un driver virtuale e un insieme di thread generati dal demone dello spazio utente. Una volta determinata la configurazione di sistema che si vuole testare, il driver può procedere alla creazione dei device file e alla specializzazione del loro comportamento. Ogni thread è associato ad una CPU, mentre i device files sono associati ad un canale DMA in caso di trasferimenti tramite *Politiche Dinamiche CFTU*. I thread possono sottomettere le operazioni al driver usando le *file operation* dei device file.

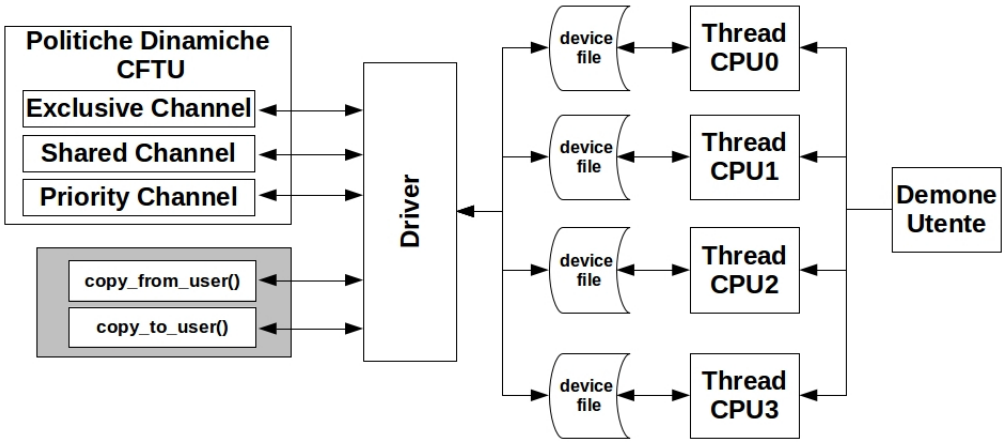

Figura 5.3: Architettura del Micro-Benchmark Bandwidth

Anche in questo micro-benchmark le *file operation* rivestono un ruolo fondamentale, infatti grazie a queste è stato possibile creare un meccanismo per:

[2] In base alla configurazione di sistema, il termine 'canale' può identificare una CPU o un canale DMA

[3] Il tempo necessario per eseguire la sequenza di operazioni dello stesso tipo

- definire e preparare l'ambiente per il test, tramite la chiamata di sistema `ioctl`;
- eseguire le operazioni da misurare, tramite le chiamata di sistema `write` e `read`.

In particolare, la chiamata `ioctl` è usata dai thread per effettuare le operazioni di lock/unlock delle pagine di memoria utente coinvolte nei trasferimenti. Le operazioni delle configurazione di sistema sono eseguite tramite l'invocazione della funzione `write`, associata con `Op_1` e della funzione `read` associata con `Op_2`. Così grazie a questo meccanismo, è stato possibile effettuare le misurazioni direttamente in spazio utente.

5.4 Micro-Benchmark Cache Pollution

Il pacchetto *Automotive* del benchmark *MiBench* [47] è composto da una serie di test volti a mostrare l'utilizzo dei processori embedded nei sistemi di controllo. Quindi, è possibile utilizzare questi programmi per simulare il comportamento di un'applicazione critica che sarà eseguita insieme ad altre operazioni di sistema. Il *Micro-Benchmark Cache Pollution* misura il *local miss rate* [4] di alcune applicazioni del pacchetto *Automotive*, mentre sono eseguiti dei trasferimenti di dati.
Di seguito sono illustrati i test del pacchetto *Automotive* utilizzati dal *Micro-Benchmark Cache Pollution*:

- **Qsort**: ordina un array di stringhe utilizzando l'algoritmo di ordinamento *quick sort*. Questo programma può lavorare sui seguenti input:
  - *small data set*: una lista di parole da ordinare;
  - *large data set*: una lista di punti da ordinare in base alla loro distanza dall'origine.
- **Susan**: è un insieme di programmi utilizzato per il riconoscimento di immagini, sviluppato per individuare angoli e bordi delle immagini celebrali delle risonanze magnetiche. Questo programma può lavorare sui seguenti input:

[4] Il *local miss rate* è il numero di miss in una cache diviso il numero totale di accessi alla stessa cache [48]

- *small input data*: un'immagine di un semplice rettangolo in bianco e nero;
- *large input data*: un'immagine complessa di medie dimensioni;
- *huge input data*: un'immagine complessa di grandi dimensioni.

Il *Micro-Benchmark Cache Pollution* esegue i test del *MiBench* sulla CPU 1 e parallelamente sulla CPU 0 effettua dei trasferimenti di dati, così da creare due contesti d'esecuzione separati che condividono l'ultimo livello di cache (L2 o L3 a seconda dell'architettura del sistema). Inoltre è possibile effettuare i trasferimenti di dati tramite motore DMA[5] o CPU, così da analizzare la *Cache Pollution* generata dalle due modalità di trasferimento.

Le configurazioni di sistema supportate sono:

- *CPU*: l'operazione di trasferimento è effettuata tramite CPU.
- *DMA Sh*: l'operazione di trasferimento è effettuata utilizzando le API della politica dinamica *Shared Channel*.

La Tabella 5.3 mostra le operazioni di trasferimento supportate dalle varie configurazioni di sistema.

	Op_1	Op_2	Op_3
CPU	`copy_from_user()`	`copy_to_user()`	`Op_1 + Op_2`
DMA Sh	`_dma_copy_from_user_sh()`	`_dma_copy_to_user_sh()`	`Op_1 + Op_2`

Tabella 5.3: Operazioni supportate dal Micro-Benchmark Cache Pollution

I due contesti di esecuzione del *Micro-Benchmark Cache Pollution* sono mostrati in Figura 5.3. Come è possibile notare sulla CPU 0 sono in esecuzione un driver virtuale e un insieme di thread creati dal Demone 1 dello spazio utente. Una volta determinata la configurazione di sistema che si vuole testare, il driver può procedere alla creazione dei device file e alla specializzazione del loro comportamento. Le *file operation* dei device file

[5]Anche con i trasferimenti DMA è necessario il contesto di esecuzione separato, infatti l'applicazione che genera i trasferimenti deve comunque essere eseguita sulla CPU. Inoltre, è bene ricordare, che il motore DMA prima di ogni transazione deve essere inizializzato (fase di *setup*) dalla CPU.

sono utilizzate per invocare dallo spazio utente le operazioni di trasferimento messe a disposizione dalla configurazione di sistema. I thread vengono eseguiti solo sulla CPU 0 e utilizzano le *file operation* di un device file (uno per ogni thread) per effettuare trasferimenti di dati e quindi generare *Cache Pollution*.
Il Demone 2 crea un thread, che verrà eseguito sulla CPU 1, utilizzato per lanciare i vari test del *MiBench*. Ogni test si servirà del *Sottosistema Performance Counter* per tenere traccia dei propri accessi e miss in cache. Questo sottosistema permette di utilizzare speciali registri hardware della CPU per monitorare il verificarsi di determinati eventi [49] [50]. Ogni test del *MiBench*, prima di terminare, invia le misurazioni al thread, che a sua volta le renderà disponibili al Demone 2 per il calcolo degli indici di prestazione.

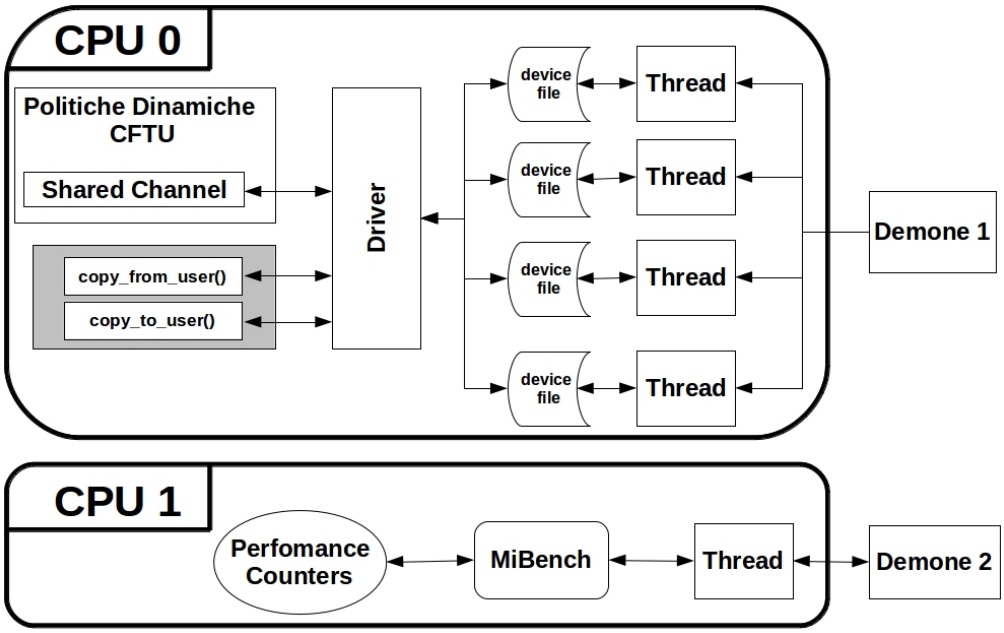

Figura 5.4: Architettura del Micro-Benchmark Cache Pollution

Prima di terminare, il *Micro-Benchmark Cache Pollution* sintetizza le misure raccolte su ogni test del *MiBench* e calcola i seguenti indici prestazionali:

- il numero medio di accessi in cache L2;
- il numero medio di miss in cache L2;
- la media dei *local miss rate*.

Capitolo 6

Analisi dei Risultati

Si passerà ora a presentare e discutere i risultati ottenuti tramite il *Sistema di Profiling* ed i micro-benchmark precedentemente descritti.

La macchina su cui sono stati eseguiti i test ha le seguenti caratteristiche:

- **Server Familiy:** Apple Xserve G5
- **CPU 1:** Intel(R) Xeon(R) CPU 5150 @ 2.66GHz 64 bit
- **CPU 2:** Intel(R) Xeon(R) CPU 5150 @ 2.66GHz 64 bit
- **CACHE:** $L1_{d/i}$ 32KB, L2 4MB
- **RAM:** 4 GB
- **MCH:** 5000X Chipset Memory Controller Hub
- **DMA:** 5000 Series Chipset DMA Engine 64 bit
- **Sistema operativo:** Linux Slackware, kernel v. 3.9.2 (patch dma-cftu)

6.1 Profiling delle copie in memoria

Analizzando le informazioni di profiling è possibile calcolare, per un dato intorno temporale, quante volte una determinata operazione è stata chiamata dal kernel e partizionare suddette invocazioni in base alla grandezza del blocco di memoria su cui operano.

La Figura 6.1 mostra il numero di invocazioni delle funzioni `copy_from_user()` e `__copy_from_user()` in una finestra temporale di 100 secondi.

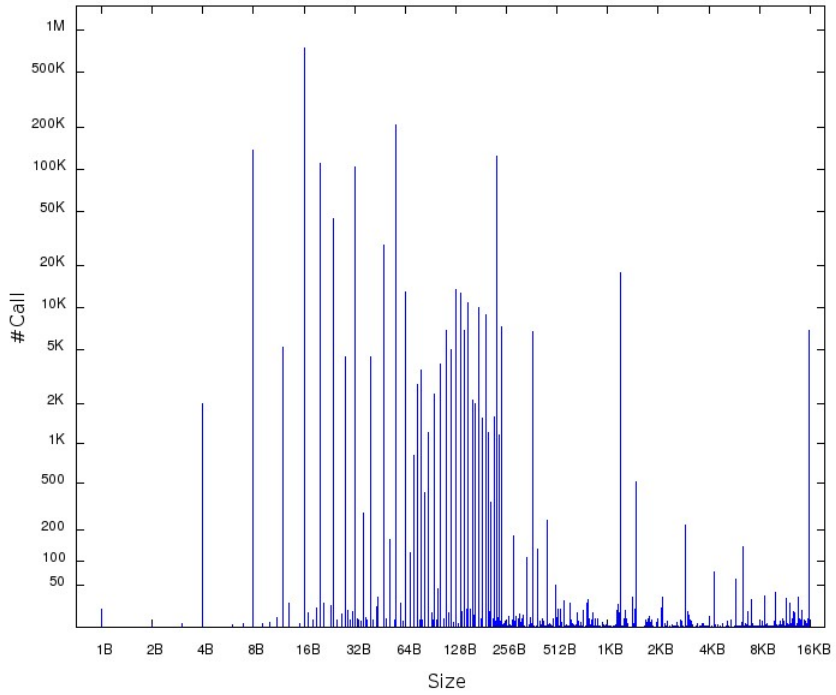

Figura 6.1: Profiling copy_from_user() e __copy_from_user()

Sintetizzando le misurazioni emerge che un sistema operativo, soggetto ad un alto carico di lavoro, effettua circa $1.68 \cdot 10^6$ operazioni di copia dalla memoria kernel a quella utente in 100 secondi di attività. In questo caso i trasferimenti di dati coinvolgono principalmente i blocchi di memoria del seguente intorno $[8 - 256]$ byte, anche se ci sono dei picchi in corrispondenza dei blocchi da 1224 byte e 16064 byte.

La Figura 6.2 mostra il numero di invocazioni delle funzioni `copy_to_user()` e `__copy_to_user()` in una finestra temporale di 100 secondi.

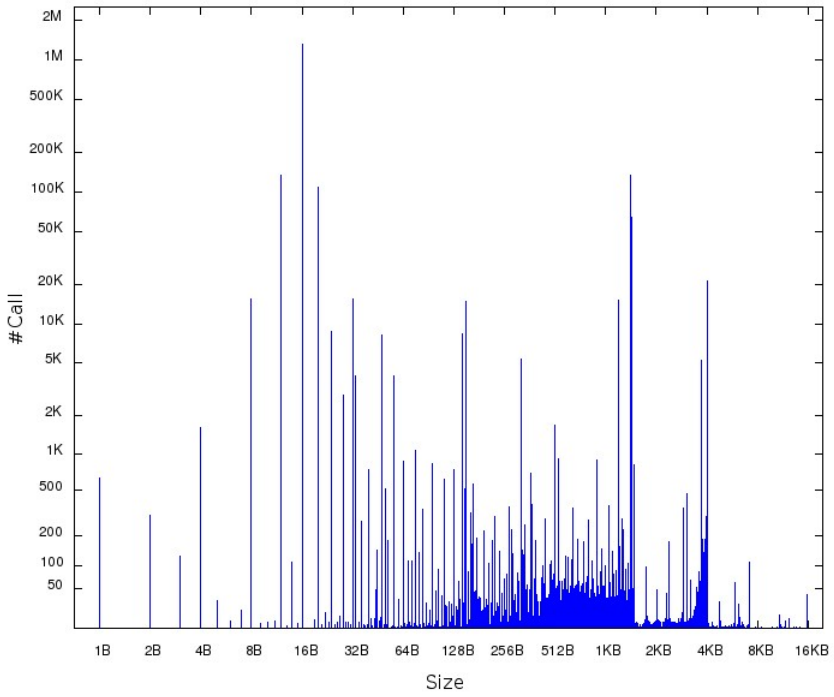

Figura 6.2: Profiling copy_to_user() e __copy_to_user()

Sintetizzando le misurazioni emerge che un sistema operativo, soggetto ad un alto carico di lavoro, effettua circa $1.9 \cdot 10^6$ operazioni di copia dalla memoria utente a quella kernel in 100 secondi di attività. In questo caso i trasferimenti di dati coinvolgono principalmente i blocchi di memoria dei seguenti intorni $[8 - 1224]$ e $[3776 - 4096]$ byte.

6.2 Analisi del tempo di esecuzione

Questo paragrafo è dedicato all'analisi dei risultati ottenuti mediante l'esecuzione del *Micro-Benchmark Execution Time*. Per fornire una panoramica esaustiva sulle prestazioni della CPU e del motore DMA, i test sono stati effettuati su blocchi di dati di dimensioni crescenti; l'intorno preso come riferimento è il seguente: $[1 - 2^{25}]$ byte. Ogni operazione è stata eseguita 300 volte per ogni blocco di dati, così da poterne graficare il tempo medio di esecuzione.

La Figura 6.3 mostra il tempo medio di esecuzione delle seguenti operazioni:

- **Op. CPU**: `copy_from_user()`
- **Op. Excl**: `_dma_copy_from_user_excl()`

- **Op. Sh**: `_dma_copy_from_user_sh()`

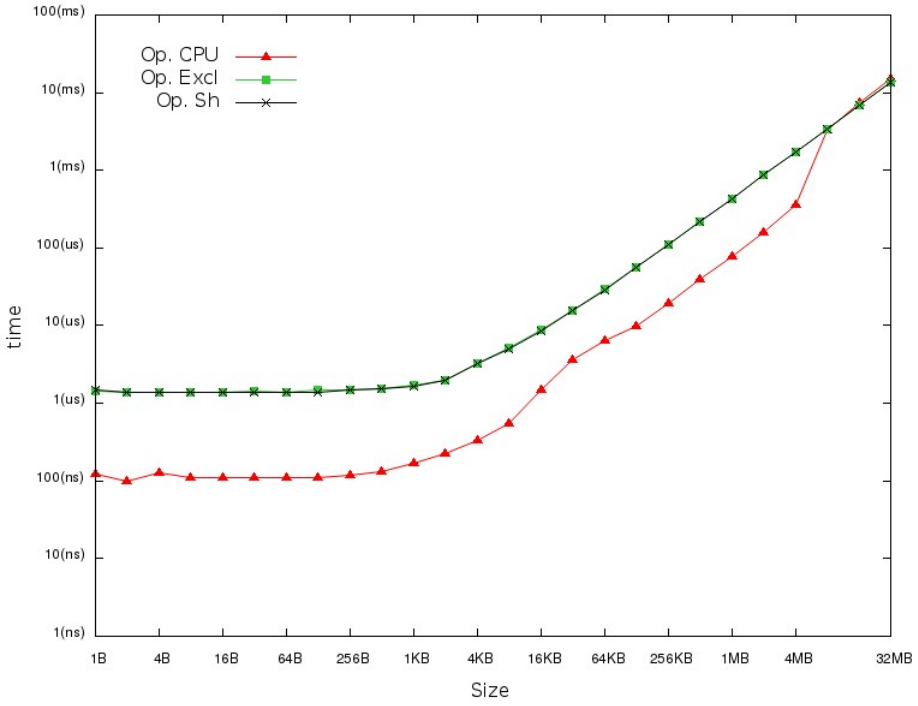

Figura 6.3: Tempo di copia dalla memoria utente a quella kernel

Se la dimensione del buffer è inferiore a 4 MB, le prestazioni della CPU (Op. CPU) sono nettamente migliori di quelle del motore DMA (Op. Excl e Op. Sh). Mentre, con buffer di dimensioni superiori, si riscontrano tempi di esecuzione simili per i vari tipi di trasferimento. Come è possibile notare, le operazioni eseguite dalla CPU presentano migliori prestazioni finché una buona parte dei dati da trasferire risiede in cache L2.

La Figura 6.4 mostra il tempo medio di esecuzione delle seguenti operazioni:

- **Op. CPU**: `copy_from_user()`^{nc}
- **Op. Excl**: `_dma_copy_from_user_excl()`^{nc}
- **Op. Sh**: `_dma_copy_from_user_sh()`^{nc}

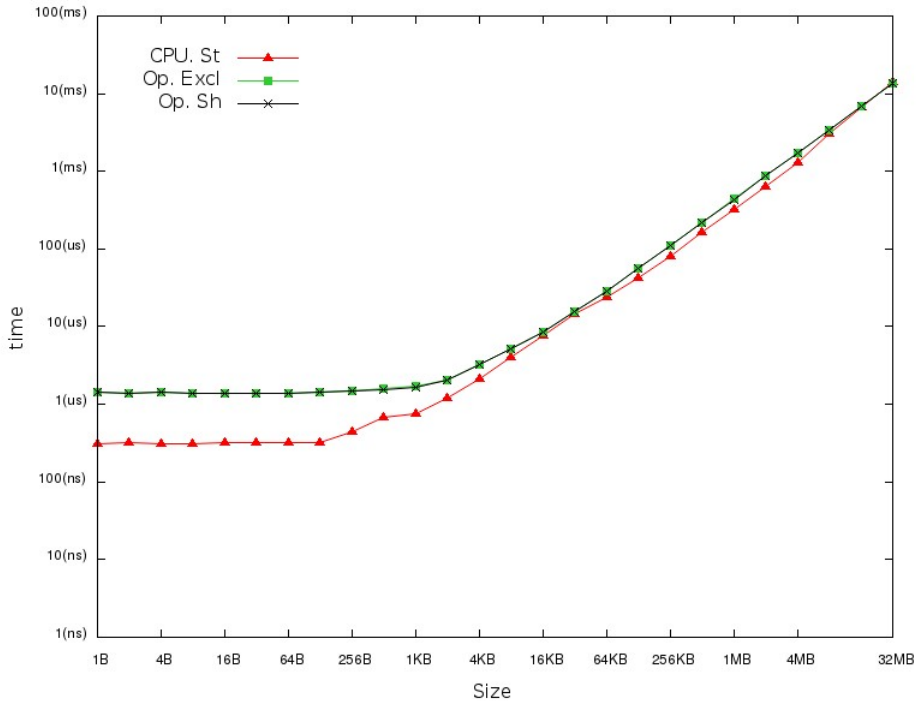

Figura 6.4: Tempo di copia dalla memoria utente a quella kernel (no cache L2)

Se la dimensione del buffer è inferiore a 32 KB, le prestazioni della CPU (Op. CPU) sono nettamente migliori di quelle del motore DMA (Op. Excl e Op. Sh). Mentre, con buffer di dimensioni superiori, si riscontrano tempi di esecuzione simili per i vari tipi di trasferimento. Le operazioni^{nc} non utilizzano l'ultimo livello di cache, quindi i trasferimenti eseguiti tramite CPU presentano migliori prestazioni finché una buona parte dei dati coinvolti risiede in cache L1.

La Figura 6.5 mostra il tempo medio di esecuzione delle seguenti operazioni:

- **Op. CPU**: `copy_to_user()`
- **Op. Excl**: `_dma_copy_to_user_excl()`
- **Op. Sh**: `_dma_copy_to_user_sh()`

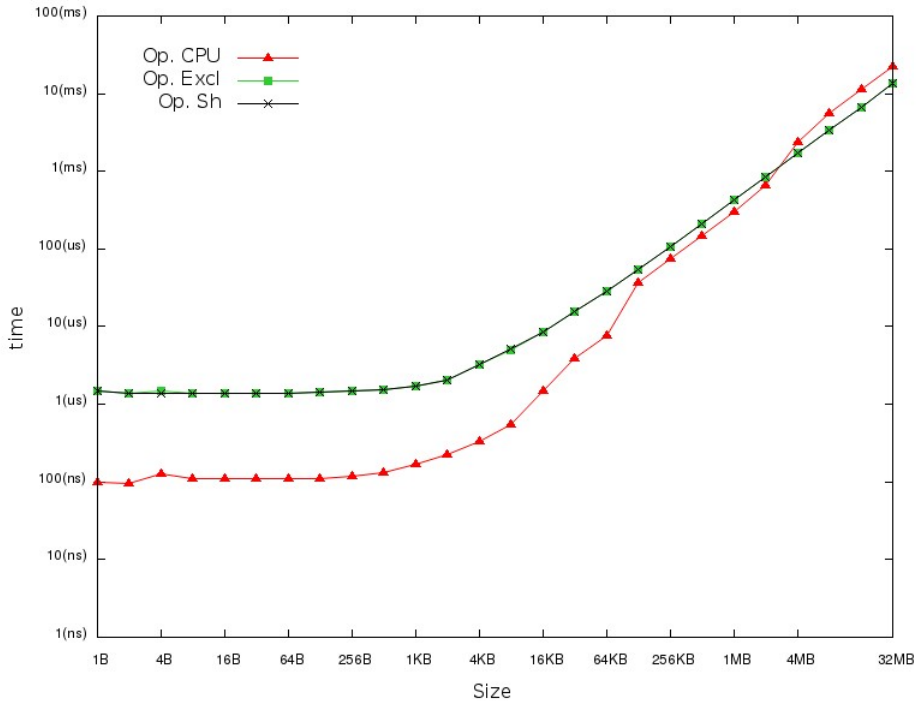

Figura 6.5: Tempo di copia dalla memoria kernel a quella utente

Se la dimensione del buffer è inferiore a 2 MB, le prestazioni della CPU (Op. CPU) sono nettamente migliori di quelle del motore DMA (Op. Excl e Op. Sh). Mentre, con buffer di dimensioni superiori, si riscontrano migliori prestazioni per le operazioni effettuate tramite motore DMA. Come è possibile notare, le operazioni eseguite dalla CPU presentano migliori prestazioni finché una buona parte dei dati da trasferire risiede in cache L2.

La Figura 6.6 mostra il tempo medio di esecuzione delle seguenti operazioni:

- **Op. CPU**: `copy_to_user()`^{nc}
- **Op. Excl**: `_dma_copy_to_user_excl()`^{nc}
- **Op. Sh** (Op. Sh): `_dma_copy_to_user_sh()`^{nc}

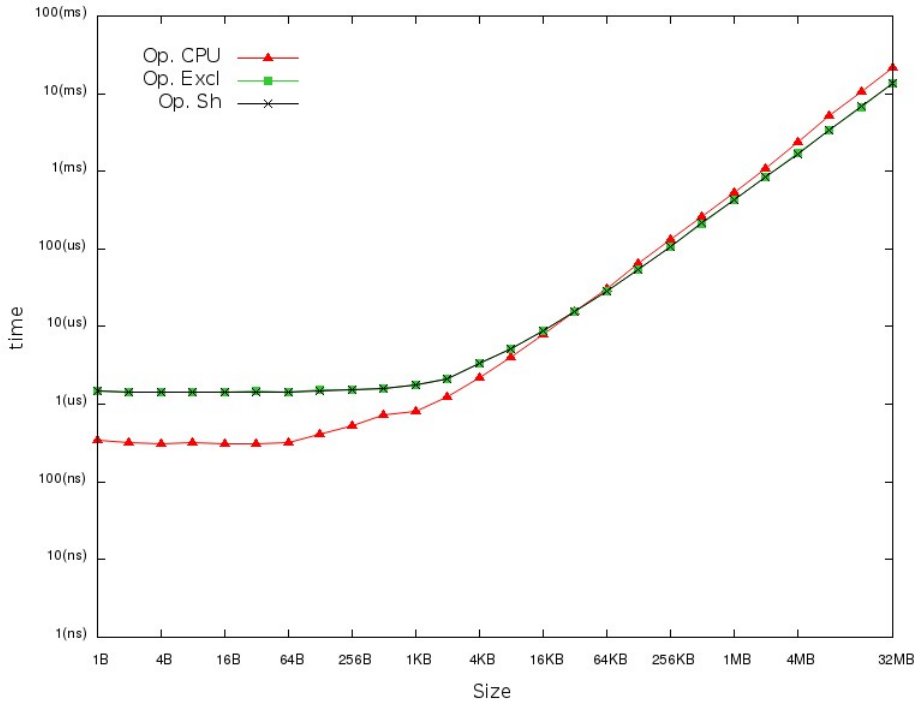

Figura 6.6: Tempo di copia dalla memoria kernel a quella utente (no cache L2)

Se la dimensione del buffer è inferiore a 32 KB le prestazioni della CPU (Op. CPU) sono nettamente migliori di quelle del motore DMA (Op. Excl e Op. Sh). Mentre, con buffer di dimensioni superiori a 128 KB, si riscontrano migliori prestazioni per le operazioni effettuate tramite motore DMA. Le operazioni^{nc} non utilizzano l'ultimo livello di cache, quindi i trasferimenti eseguiti tramite CPU presentano migliori prestazioni finché una buona parte dei dati coinvolti risiede in cache L1.

I test appena condotti mostrano che i tempi di esecuzione dei trasferimenti DMA sono predicibili, infatti non dipendono dallo stato della memoria cache. Osservando le varie curve a loro associate si riscontra una crescita lineare del tempo di trasferimento dovuto all'aumentare del blocco di dati da trasferire. L'unico particolare che può destare stupore è il valore costante di suddette curve nell'intervallo $[1 - 4096]$ byte, questo fenomeno è dovuto alla modalità di trasferimento del motore DMA che opera su blocchi di dati da 4KB.

6.3 Analisi della velocità di trasferimento

I precedenti test hanno sfruttato un numero limitato di risorse DMA o CPU per i trasferimenti di dati, ma parallelizzando le operazioni di copia è possibile coinvolgere tutte le risorse di sistema. I risultati, ottenuti mediante l'esecuzione del *Micro-Benchmark Bandwidth*, permettono di studiare la velocità di trasferimento del sistema, che utilizza tutte le risorse disponibili (tutte le CPU o tutti i canali DMA) per eseguire le operazioni di copia.
Al fine di fornire una panoramica esaustiva sulle prestazioni della CPU e del motore DMA, i test sono stati effettuati su blocchi di dati di dimensioni crescenti; l'intorno preso come riferimento è il seguente: $[2^{10} - 2^{25}]$ byte. E' bene specificare che la banda graficata sarà quella totale, ossia la somma delle velocità di trasferimento di tutte le CPU o di tutti i canali DMA a seconda della tipologia di operazione. Inoltre la configurazione *DMA Prio* utilizzerà un canale di tipo *CH-HIGH* e 3 di tipo *CH-LOW* con $t_h = 30$ μs e $t_l = 60$ μs.

La Figura 6.7 mostra la banda totale dei canali DMA o delle CPU che effettuano le seguenti operazioni di trasferimento:

- **Op. CPU**: `copy_from_user()`
- **Op. Excl**: `_dma_copy_from_user_excl()`
- **Op. Prio**: `_dma_copy_from_user_prio()`
- **Op. Sh**: `_dma_copy_from_user_sh()`

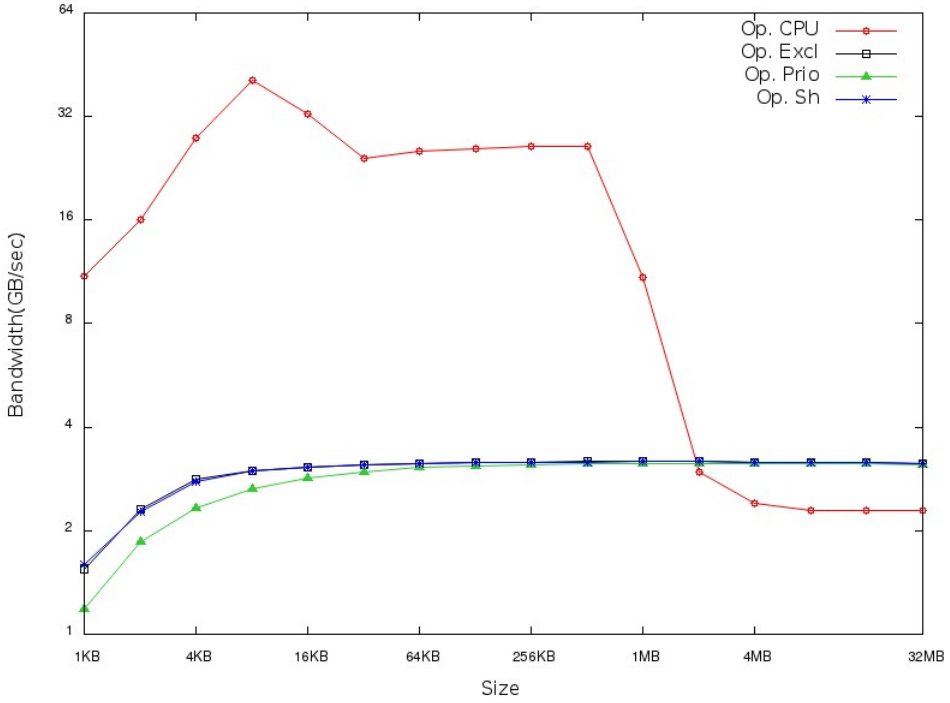

Figura 6.7: Bada totale relativa ai trasferimenti dalla memoria utente a quella kernel

Come è possibile notare se la dimensione del buffer è inferiore a 2 MB, le prestazioni delle CPU (Op. CPU) sono nettamente migliori di quelle del motore DMA (Op. Excl, Op. Sh, Op. Prio). Mentre, con buffer di dimensioni superiori, la velocità di trasferimento del motore DMA risulta più alta di quella delle CPU. Le operazioni eseguite dalle CPU presentano migliori prestazioni finché una buona parte dei dati da trasferire risiede in cache L2. Inoltre, è interessante notare che con buffer inferiori a 64 KB le operazioni Op. Prio presentano un velocità di trasferimento più bassa rispetto alle Op. Excl e Op. Sh. Suddetto fenomeno è causato dal tipo di gestione dei canali della politica *Priority Channel*, infatti durante l'intervallo t_h il motore DMA esegue solo i trasferimenti del canale ad alta priorità quindi risulta sottoutilizzato.

La Figura 6.8 mostra la banda totale dei canali DMA o delle CPU che effettuano le seguenti operazioni di trasferimento:

- **Op. CPU**: `copy_to_user()`
- **Op. Excl**: `_dma_copy_to_user_excl()`
- **Op. Prio**: `_dma_copy_to_user_prio()`

- **Op. Sh**: `_dma_copy_to_user_sh()`

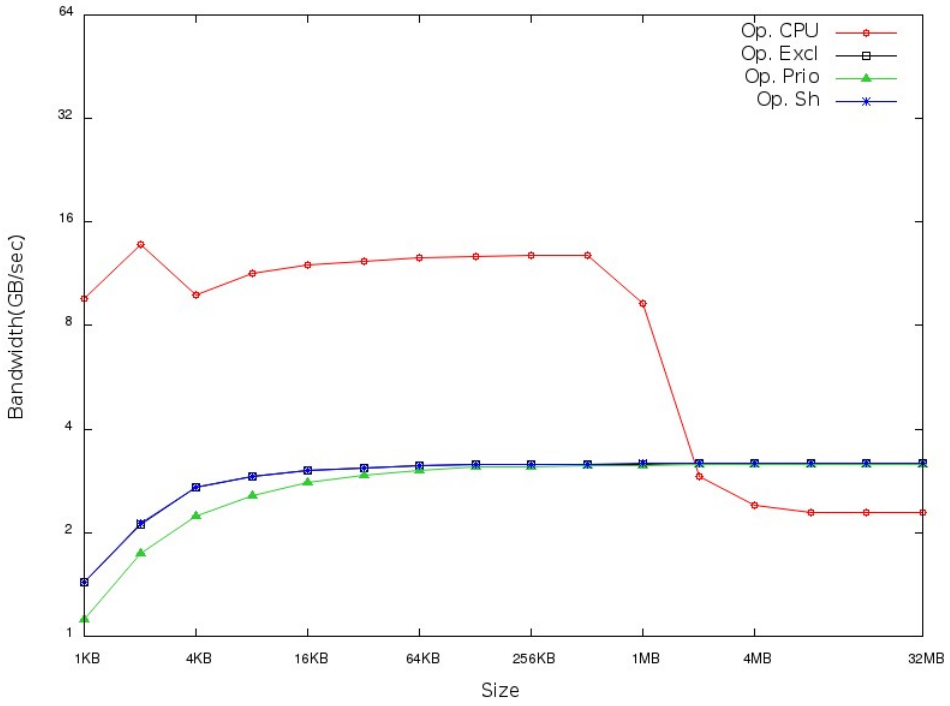

Figura 6.8: Bada totale relativa ai trasferimenti dalla memoria kernel a quella utente

Le considerazioni sulla Figura 6.8 sono analoghe a quelle della Figura 6.7.

La Figura 6.9 mostra la banda totale dei canali DMA o delle CPU che effettuano le seguenti operazioni di trasferimento:

- **Op. CPU**: `copy_from_user()` e `copy_to_user()`
- **Op. Excl**: `_dma_copy_from_user_excl()` e `_dma_copy_to_user_excl()`
- **Op. Prio**: `_dma_copy_from_user_prio()` e `_dma_copy_to_user_prio()`
- **Op. Sh**: `_dma_copy_from_user_sh()` e `_dma_copy_to_user_sh()`

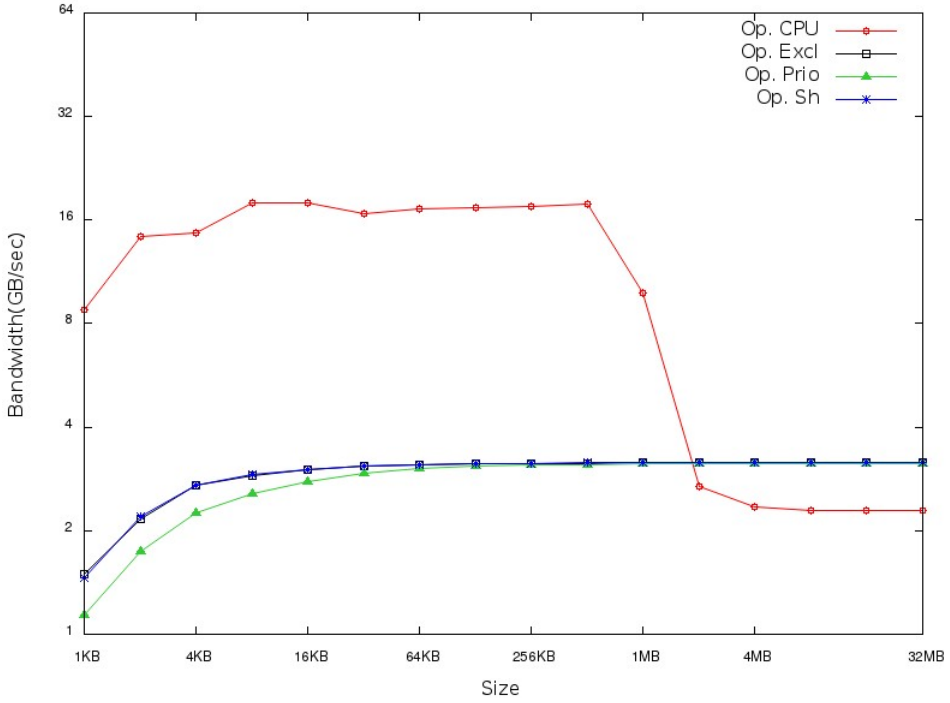

Figura 6.9: Bada totale relativa ai trasferimenti tra memoria kernel e utente

Le considerazioni sulla Figura 6.9 sono analoghe a quelle della Figura 6.7.

I test appena condotti mostrano che la velocità di trasferimento dei canali DMA è costante, mentre quella delle CPU dipende dallo stato della memoria cache e quindi anche dalle dimensioni del blocco di dati da copiare. L'unico particolare che può destare stupore è il valore crescete delle curve relative ai trasferimenti DMA nell'intervallo $[1 - 4096]$ byte, questo fenomeno è dovuto alla modalità di trasferimento del motore DMA che opera su blocchi di dati da 4KB.

6.3.1 Prestazioni della politica Priority Channel

Per effettuare un'analisi puntuale sulla politica *Priority Channel* è necessario analizzare la velocità di trasferimento dei singoli canali, così da mettere a confronto le prestazioni delle due classi *CH-HIGH* e *CH-LOW*. Quindi nel seguito verranno rianalizzati i test precedentemente presentati (Figure 6.7 6.8 6.9) al fine di studiare le prestazioni dei singoli canali DMA.

I grafici presentati mostreranno la velocità di trasferimento dei seguenti canali DMA:

- **chan 0**: canale ad alta priorità (*CH-HIGH*) gestito tramite politica *Priority Channel*;
- **chan 1**, **chan 2** e **chan 3**: canali a bassa priorità (*CH-LOW*) gestiti tramite politica *Priority Channel*;
- **chanExcl**: canale gestito tramite politica *Exclusive Channel*.

La Figura 6.10 mostra la banda dei canali DMA che effettuano le seguenti operazioni di trasferimento:

- **chan Excl**: `_dma_copy_from_user_excl()`
- **chan 0,1,2,3**: `_dma_copy_from_user_prio()`

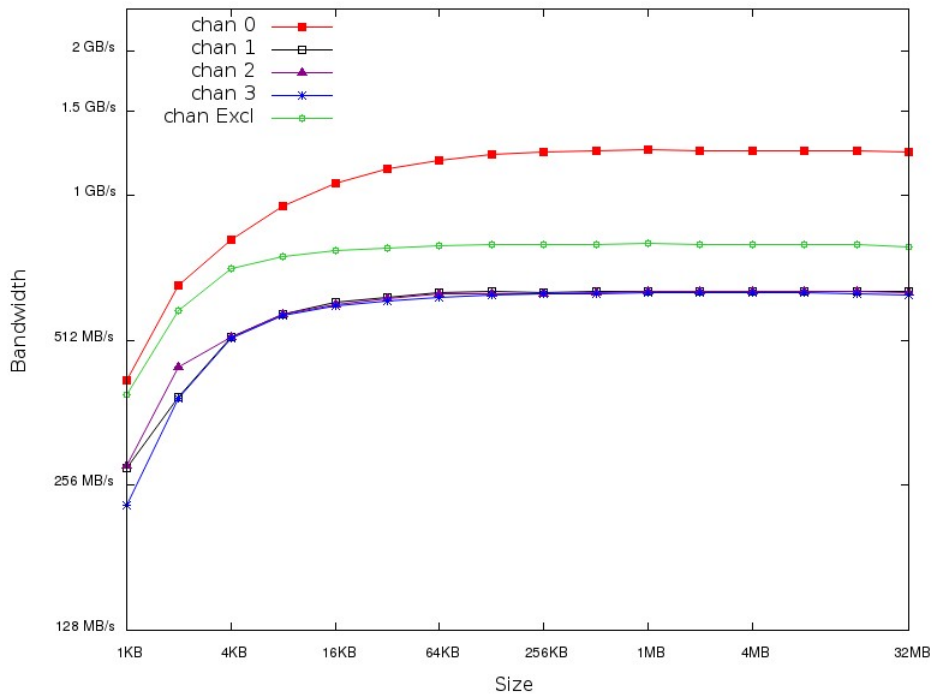

Figura 6.10: Banda dei singoli canali che effettuano trasferimenti dalla memoria utente a quella kernel

Come è possibile notare con buffer di dimensioni superiori ai 64 KB le prestazioni del canale ad alta priorità (*chan 0*) sono di gran lunga migliori di quelle dei restanti canali graficati. Infatti la velocità di trasferimento di *chan 0* raggiunge il valore di 1.2 GB/s, che è quasi il doppio di quella dei canali a bassa priorità (*chan 1,2,3*) che trasmettono a 0.65 GB/s. Questo alto

divario prestazione permane anche considerando la velocità di trasferimento del canale generico *chan Excl* che è pari a 0.8 GB/s.

La Figura 6.11 mostra la banda dei canali DMA che effettuano le seguenti operazioni di trasferimento:

- **chan Excl**: `_dma_copy_to_user_excl()`
- **chan 0,1,2,3**: `_dma_copy_to_user_prio()`

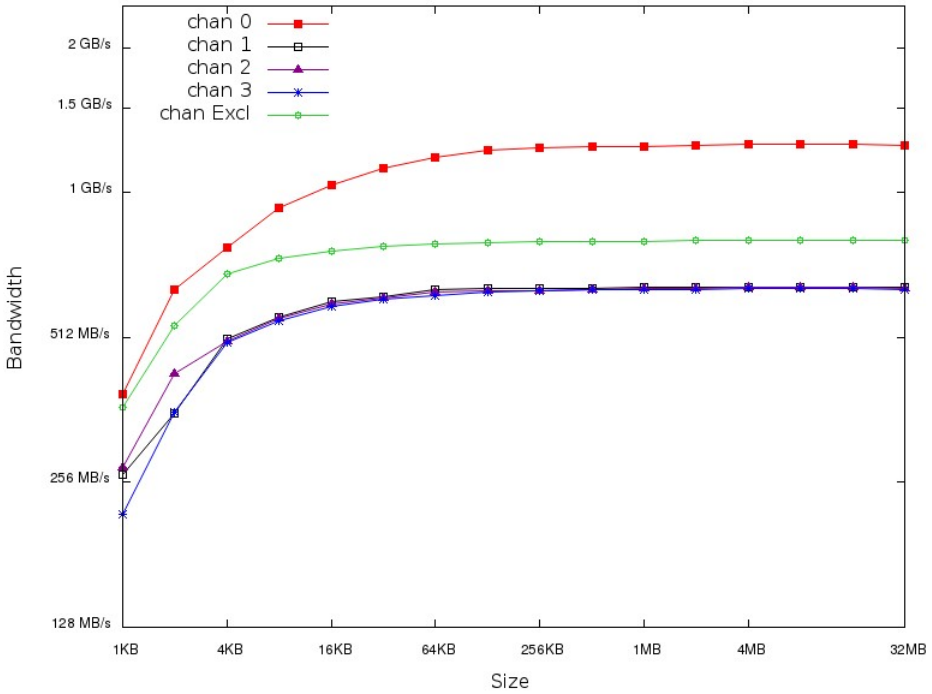

Figura 6.11: Banda dei singoli canali che effettuano trasferimenti dalla memoria kernel a quella utente

Le considerazioni sulla Figura 6.11 sono analoghe a quelle della Figura 6.10.

La Figura 6.12 mostra la banda dei canali DMA che effettuano le seguenti operazioni di trasferimento:

- **chan Excl**: `_dma_copy_from_user_excl()` e `_dma_copy_to_user_excl()`
- **chan 0,1,2,3**: `_dma_copy_from_user_prio()` e `_dma_copy_to_user_prio()`

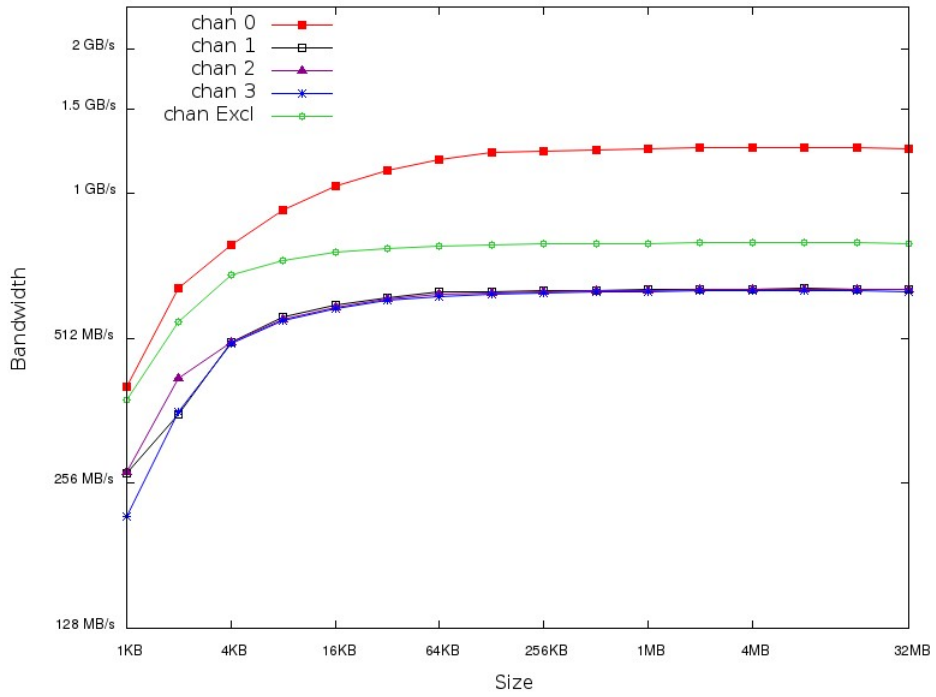

Figura 6.12: Banda dei singoli canali che effettuano trasferimenti tra memoria kernel e utente

Le considerazioni sulla Figura 6.12 sono analoghe a quelle della Figura 6.10.

I test appena condotti mostrano che la velocità di trasferimento del canale DMA ad alta priorità è costante e significativamente più alta dei restanti canali a bassa priorità. Così si può concludere che è possibile utilizzare la politica *Priority Channel* per discriminare l'assegnazione delle risorse DMA in base alla criticità dell'operazione e quindi delegare al/ai canale/i *CH-HIGH* l'espletamento dei task ad alta priorità.

6.4 Analisi della Cache Pollution

Il *Micro-Benchmark Cache Pollution* misura il *local miss rate* (della cache L2) relativo ad alcuni test del *Mibench* mentre sono in esecuzione delle copie in memoria. Grazie a questo micro-benchmark è possibile studiare la *Cache Pollution* generata dalle operazioni di copia in memoria.

Per eseguire un'analisi esaustiva sono stati considerati e studiati tre scenari diversi:

1. **Ideal System**: vengono eseguiti i test del benchmark *Mibench* sulla CPU 1 mentre la CPU 0 è scarica, rappresenta l'ottimo in termini di *local miss rate*.

2. **CPU System**: vengono eseguiti i test del benchmark *Mibench* sulla CPU 1 ed in parallelo sulla CPU 0 dei thread effettuano delle copie in memoria tramite processore.

3. **DMA System**: vengono eseguiti i test del benchmark *Mibench* sulla CPU 1 ed in parallelo sulla CPU 0 dei thread effettuano, tramite motore DMA, delle copie in memoria.

I test del *MiBench* utilizzati dal *Micro-Benchmark Cache Pollution* sono mostrati in tabella 6.1; i valori della colonna **L2 Ref.** rappresentano il numero accessi in cache L2 effettuato da ogni test.

Name	**Test Mibench**	**Input**	**L2 Ref.**
susan[s]_small	Susan Smoothing	immagine (~ 7 KB)	1431
susan[e]_small	Susan Edge	immagine (~ 7 KB)	1986
susan[c]_small	Susan Corners	immagine (~ 7 KB)	1652
susan[s]_large	Susan Smoothing	immagine (~ 110 KB)	7776
susan[e]_large	Susan Edge	immagine (~ 110 KB)	39514
susan[c]_large	Susan Corners	immagine (~ 110 KB)	20955
susan[s]_huge	Susan Smoothing	immagine (~ 1 MB)	88063
susan[e]_huge	Susan Edge	immagine (~ 1 MB)	304382
susan[c]_huge	Susan Corners	immagine (~ 1 MB)	108839
qsort_small	Qsort	10^4 parole	157450
qsort_large	Qsort	$5 \cdot 10^4$ punti (x,y,z)	519904

Tabella 6.1: Test del Mibench effettuati dal Micro-Benchmark Cache Pollution

Inoltre al fine di rendere questa analisi significativa, sono stati utilizzati i dati forniti dal *Sistema di Profiling* per individuare le tuple <*funzione , blocco di memoria di grandi dimensioni*> invocate più spesso, che nel nostro caso sono:

- `copy_from_user()` su blocchi di memoria da 16064 byte;
- `copy_to_user()` su blocchi di memoria da 4096 byte.

La Figura 6.13 mostra i *local miss rate* relativi ai test del *Mibench* mentre in parallelo sulla CPU 0 sono in esecuzioni le seguenti operazioni:

- **Ideal System**: nessuna operazione;
- **CPU System**: `copy_from_user()` su blocchi di dati da 16064 byte;
- **DMA System**: `_dma_copy_from_user_sh()` su blocchi di dati da 16064 byte.

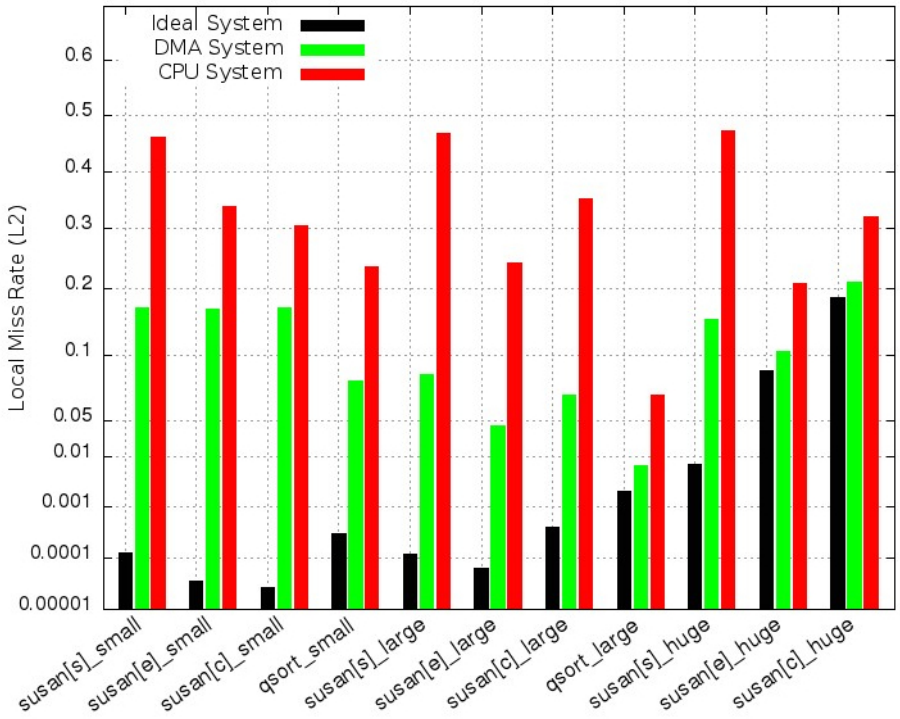

Figura 6.13: Local miss rate (L2) con trasferimenti di 16064B dalla memoria utente a quella kernel

Per quantificare la *Cache Pollution* generata dai trasferimenti DMA e CPU, si confronta il *local miss rate* relativo ai due scenari *DMA System* e *CPU System* con quello dell'*Ideal System*, che rappresenta l'ottimo in termini di *cache miss*.
Come è possibile notare lo scenario *CPU System* presenta le prestazioni peggiori in termini di *local miss rate* e ciò è riscontrabile su ogni test del *Mibench* sia quelli relativi all'elaborazione delle immagini (*susan*) che quelli sull'ordinamento (*qsort*). Mentre le prestazioni dello scenario *DMA System*,

come era lecito aspettarsi, sono inferiori a quelle dell'*Ideal System*, ma sicuramente migliori del *CPU System*. Inoltre analizzando i *test susan[e]_huge* e *susan[c]_huge* si riscontra che le prestazioni dello scenario *DMA System* sono di poco inferiori a quelle dell'*Ideal System*. Questo fenomeno è spiegabile osservando l'input di dimensioni considerevoli (vedi di tabella 6.1) sottomesso ai due test, che anche in un contesto isolato, quale l'*Ideal System*, presentano comunque un alto valore di *local miss rate*.

La Figura 6.14 mostra i *local miss rate* relativi ai test del *Mibench* mentre in parallelo sulla CPU 0 sono in esecuzioni le seguenti operazioni:

- **Ideal System**: nessuna operazione;
- **CPU System**: `copy_to_user()` su blocchi di dati da 4096 byte;
- **DMA System**: `_dma_copy_to_user_sh()` su blocchi di dati da 4096 byte.

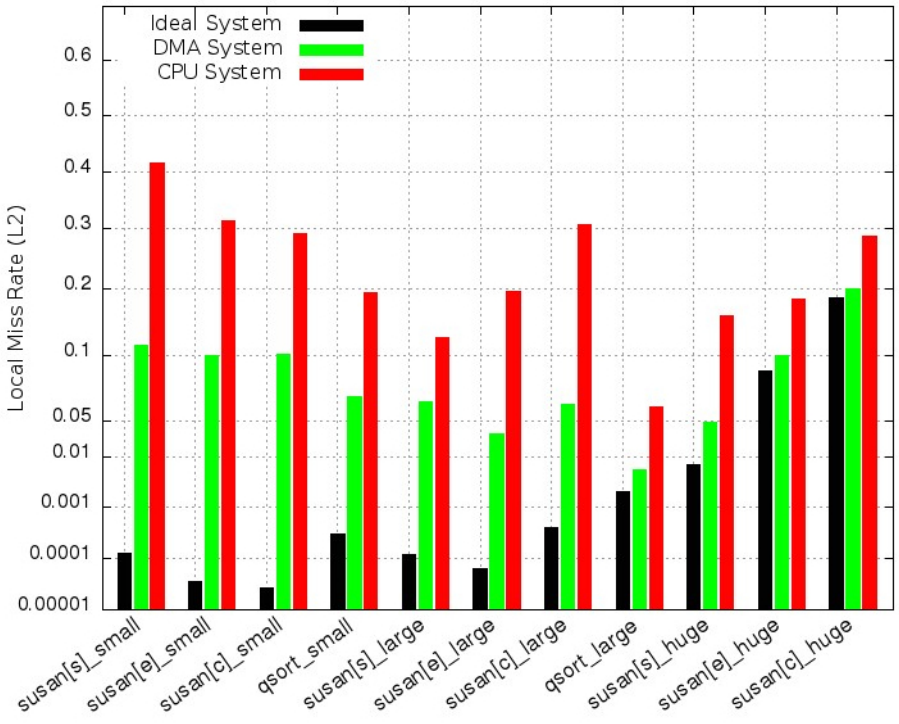

Figura 6.14: Local miss rate (L2) con trasferimenti di 4096B dalla memoria kernel a quella utente

Le considerazioni sulla Figura 6.14 sono analoghe a quelle della Figura 6.13.

Dopo questa breve analisi è possibile concludere che il fenomeno della *Cache Pollution* relativo ai trasferimenti DMA è circoscritto alle operazioni di pre-trasferimento, ossia tutte le operazioni eseguite dal micro-benchmark che esulano dal trasferimento effettivo.

Conclusioni

L'*Estensione DMA-CFTU*, sviluppata nel presente lavoro di tesi, fornisce un'interfaccia generica per effettuare le operazioni di copia tra memoria kernel e utente tramite motore DMA, inoltre mette a disposizione una serie di politiche volte alla gestione dei canali DMA. Una di queste, la *Priority Channel*, è stata sviluppata per gestire i trasferimenti di applicazioni real-time, infatti consente di assegnare il bus del motore DMA, per un certo intervallo di tempo, ai canali che gestiscono task ad alta priorità.

L'analisi dei tempi di esecuzione delle copie in memoria, eseguita tramite il *Micro-Benchmark Time Execution*, ha dimostrato che le prestazioni dei trasferimenti DMA sono migliori di quelle della CPU se i blocchi di dati coinvolti sono di dimensioni considerevoli, ossia maggiori di 2 MB. Inoltre nel corso della trattazione, è stato verificato che la CPU ha migliori prestazioni del motore DMA fino a quando una buona parte dei dati da trasferire risiede in memoria cache. Quindi il tempo di esecuzione delle operazioni di copia effettuate tramite CPU è fortemente legato allo stato della memoria cache, mentre il motore DMA, non utilizzando suddetta memoria, presenta un tempo di esecuzione che è direttamente proporzionale alla dimensione del blocco da trasferire. Così è possibile affermare che le operazioni di copia effettuate tramite motore DMA sono altamente predicibili.
I test effettuati tramite il *Micro-Benchmark Bandwidth* hanno confermato quanto appena asserito, ossia che la velocità di trasferimento dei canali DMA è costante, mentre quella delle CPU dipende dallo stato della memoria cache e quindi anche dalle dimensioni del blocco di dati da copiare. Inoltre è stata fatta un'analisi puntuale sulla velocità di trasferimento dei singoli canali DMA gestiti con politica *Priority Channel*, i risultati hanno dimostrato che la velocità di trasferimento del canale DMA ad alta priorità è costante e significativamente più elevata dei restanti canali a bassa priorità. Così si è concluso che è possibile utilizzare la politica *Priority Channel* per discrimi-

nare l'assegnazione delle risorse DMA in base alla criticità dell'operazione e quindi delegare al/ai canale/i ad alta priorità l'espletamento dei task real-time.
Infine è stato dimostrato che il fenomeno della *Cache Pollution* relativo ai trasferimenti DMA è circoscritto alle operazioni di pre-trasferimento, che nel presente lavoro sono identificate dall'insieme di istruzioni eseguite dal micro-benchmark prima del trasferimento effettivo.

In conclusione, l'utilizzo di un motore DMA, per effettuare le copie tra memoria kernel e utente, apporta notevoli benefici all'intero sistema in termini di ottimizzazione delle risorse e di carico di lavoro sulle CPU.

Sviluppi Futuri

Gli sviluppi futuri di questi lavoro possono essere molteplici:

- il mantenimento del codice seguendo gli aggiornamenti del kernel Linux
- lo sviluppo di ulteriori micro-benchmark per:
  - misurare il tempo effettivo dei trasferimenti del motore DMA discriminandolo da quello di setup
  - studiare il fenomeno della Cache Pollution misurando il *local miss rate* dei test del pacchetto *Office Automation* del benchmark *MiBench*
- introdurre nell'*Estensione DMA-CFTU* il supporto alla tecnica di *Scatter/Gather*
- implementare un'interfaccia che permetta le copia memoria-memoria in spazio utente tramite motore DMA
- il motore DMA, utilizzato per questo lavoro di tesi, dispone di una circuiteria dedicata per le operazioni di XOR e MEMSET, così sarebbe interessante integrare nell'interfaccia messa a disposizione dall'*Estensione DMA-CFTU* tali operazioni dedicate.
- utilizzare l'*Estensione DMA-CFTU* su SoC (System on Chip) di ultima generazione, che dispongono di motori DMA programmabili integrati all'interno del socket dei processori (*Processor DMA*).

Bibliografia

[1] Daniel P. Bovet, Marco Cesati. Understanding The Linux Kernel. O'Reilly, third edition, November 2005.

[2] DMA: Direct Memory Access. http://www.dizionarioinformatico.com

[3] Marco Mezzalama, Corso di Sistemi a Microprocessori 2009-10. http://www.cad.polito.it/~bernardi/04flycy

[4] Personal computer/Mapping/Interfacce. http://it.wikibooks.org

[5] Carl Hamacher, Zvonko Vranesic, Safwat Zaky, Naraig Manjikian. Introduzione all'architettura dei calcolatori. McGraw-Hill, third edition, February 2013.

[6] Marco Cesati. Corso di Architetture dei calcolatori 2010. http://ac10.sprg.uniroma2.it

[7] Emiliano Betti. Sviluppo di un driver per un controllore digitale di movimento, 2004-2005. www.ebetti.net/res/BettiE_MasterThesis.pdf

[8] A. F. Harvey, Data Acquisition Division Staff NATIONAL INSTRUMENTS. DMA Fundamentals on Various PC Platforms.

[9] Personal computer/Architettura/IO. http://it.wikibooks.org/wiki/Personal_computer/Architettura/IO

[10] 8259A Programmable Interrupt Controller (8259A/8259A-2). http://pdos.csail.mit.edu/6.828/2005/readings/hardware/8259A.pdf

[11] Introduzione all'8259. http://www.intel-assembler.it/

[12] IA-32 Intel Architecture Software Developer's Manual. Volume 3: System Programming Guide. http://www.scs.stanford.edu/nyu/04fa/lab/ia32/IA32-3.pdf

[13] Mike Rieker. Advanced Programmable Interrupt Controller. http://www.osdever.net/tutorials/pdf/apic.pdf

[14] FR Series MCUs with on-chip DMAC. Fujitsu Microelectronics America, Inc. http://www.fujitsu.com/downloads/MICRO/fma/pdfmcu/an-dma-on-chip-v1-0.pdf

[15] Giacomo Bucci. Architettura dei calcolatori elettronici. McGraw-Hill, 2001.

[16] PIC24F Reference Manuals, Section 54: Direct Memory Access Controller (DMA). http://ww1.microchip.com/downloads/en/DeviceDoc/39742B.pdf

[17] Introduction to the 16-bit PIC24F Microcontroller Family. http://www.elektor.nl/Uploads/Files/PIC24FIntro_5f082806.pdf

[18] Direct Memory Access (DMA). Information Tecnology, University of Babylon. http://www.uobabylon.edu.iq/eprints/paper_12_25785_124.pdf

[19] Intel I/O Acceleration Technology Helps Eliminate Network Bottlenecks. http://www.dell.com/downloads/global/power/ps2q07-50060435-Intel.pdf

[20] IOAT. http://it.wikipedia.org/wiki/IOAT

[21] Intel I/O Acceleration Technology. http://www.broadberry.co.uk/pdf/intel_10gbe/IOAT_tech_brief.pdf

[22] Accelerating High-Speed Networking with Intel I/O Acceleration Technology. http://download.intel.com/support/network/sb/98856.pdf

[23] Introduction to TCP Offload Engines. http://www.dell.com/downloads/global/power/1q04-her.pdf

[24] Large segment offload. http://en.wikipedia.org/wiki/Large_segment_offload

[25] Efficient data transfer through zero copy. http://www.ibm.com/developerworks/library/j-zerocopy/index.html

[26] Dragan Stancevic. Zero Copy I: User-Mode Perspective *linuxjournal, Jan 01 2003.*

[27] Zero-copy. http://en.wikipedia.org/wiki/Zero-copy

[28] Integrated Network Accelleration Features, Tecnical White Paper.

[29] Marco Mezzalama, Gianluca Oglietti. Acquisizione del traffico IP da una rete ad elevato throughput utilizzando un sistema Open Source: tecnologie e problematiche.

[30] Andrew Gover, Christopher Leech. Accelerating Network Receiver Processing, Intel I/O Acceleration Technology.

[31] Improving Network Performance in Multi-Core Systems, White Paper Intel Ethernet Controllers.

[32] Intel Gigabit ET, ET2, and EF Multi-Port Server Adapters, Product Brief.

[33] Documentazione Kernel Linux 3.9.2. Documentation/networking/multiqueue.txt

[34] Documentazione Kernel Linux 3.9.2. Documentation/networking/scaling.txt

[35] Karthikeyan Vaidyanathan, High Performance and Scalable Soft Shared State for Next-Generation Datacenters

[36] Li Zhao, Ravi Iyer, Srihari Makineni, Laxmi Bhuyan, Don Newell. Hardware Support for Bulk Data Movement in Server Platforms.

[37] Instruction window. http://en.wikipedia.org/wiki/Instruction_window

[38] Buffer di riordino.
http://it.wikipedia.org/wiki/Buffer_di_riordino

[39] i/oat, Linux Foundation. http://www.linuxfoundation.org/collaborate/workgroups/networking/i/oat

[40] Intel QuickData Technology Software Guide for Linux, White Paper Server Platform Group.

[41] Karthikeyan Vaidyanathan, Dhabaleswar K. Panda. Benefits of I/O Acceleration Technology (I/OAT) in Clusters.

[42] K. Vaidyanathan, W. Huang, L. Chai, D K. Panda. Designing Efficient Asynchronous Memory Operations Using Hardware Copy Engine: A Case Study with I/OAT.

[43] K. Vaidyanathan, L. Chai, W. Huang, D K. Panda. Efficient Asynchronous Memory Copy Operations on Multi-Core System and I/OAT.

[44] Wen Su, Ling Wang, Menghao Su, Su Liu. A Processor-DMA-Based Memory Copy Hardware Accelerator.

[45] Loongson. http://en.wikipedia.org/wiki/Loongson

[46] Speedup. http://en.wikipedia.org/wiki/Speedup

[47] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst,Todd M. Austin, Trevor Mudge, Richard B. Brown. MiBench: A free, commercially representative embedded benchmark suite.

[48] Valeria Cardellini. Corso di Architetture Avanzate dei Calcolatori 2007. http://www.ce.uniroma2.it

[49] Thomas Gleixner. Performance Counters for Linux. http://lwn.net

[50] PERF_EVENT_OPEN. Section: Linux Programmer's Manual. http://web.eece.maine.edu/

Ringraziamenti

Alla fine di un percorso si prova sempre un senso di nostalgia, volendo citare il mio amico Phil “alla fine di un ciclo, prima dell’alba, si intravede il tramonto di ghiaccio“. Il mio viaggio è durato diversi lustri nei quali ho vissuto esperienze paradossali e incontrato persone di ogni genere, che mi hanno profondamente cambiato portandomi a riconsiderare tutte le certezze su cui basavo il mio essere. Ho sempre considerato l’ambiente universitario come un piccolo Eden in cui sono raggruppate persone dotate di un alto senso civico nonché di notevole apertura mentale con cui è stato piacevole potersi confrontare. Grazie a queste persone ho potuto maturare e perfezionare il mio pensiero critico, prettamente relativista, che mi ha permesso di valutare ogni situazione, che mi si poneva dinanzi, libero dagli inutili orpelli del pensiero precostituito di questa società.

Un particolare ringraziamento va ai miei genitori che mi sono stati sempre vicino nei momenti di sconforto e mai si sono opposti alle mie scelte universitarie, anteponendo il mio benessere al loro.
Devo molto anche al mio relatore il Prof. Marco Cesati e al mio correlatore l’Ing. Emiliano Betti, grazie a loro ho ritrovato gli stimoli che avevo perduto nel corso della mia carriera universitaria. Li ringrazio per avermi dimostrato fiducia e per avermi fatto comprendere quale sia il percorso professionale e lavorativo che voglio intraprendere.

Durante il mio lavoro di tesi non sono mancati i momenti di sconforto, se sono riuscito a non abbattermi e a ritrovare l’ottimismo lo devo ai miei amici Carmen, Stefano, Daniele, Neve, Katia, Peppe, Alessandra, Giovanni che non mi hanno mai abbandonato in questi 10 mesi.
Oltre a loro voglio ringraziare tutti gli amici che ho avuto vicino durante la mia carriera universitaria: Domenico, Pino, Luca (che dubito riuscirà a svegliarsi in tempo per la mia laurea), Mauro, Gabriele, Michele, Anastasia,

Giovanna, Shaggy, Ramon, Paolo, Alessandro F., Giuseppe P., Alessandro P., Alessio T., Cristina, Noemi, Alessio P., Emanuele, Matteo, Francesco, Carlo e tutti quelli che non ho menzionato in queste pagine.

Voglio ringraziare mia sorella Valeria, Simone e Sofie per i bei momenti trascorsi insieme.

Un grazie anche ai miei zii e cugini per tutto l'affetto che mi hanno sempre riservato.