Università degli Studi di Bologna

FACOLTÀ DI INGEGNERIA

Corso di Laurea in Ingegneria Elettronica

Insegnamento: Elettronica Applicata II

SVILUPPO DI UN AMBIENTE DI SIMULAZIONE PER ARCHITETTURE MULTIPROCESSORI SU SINGOLO CHIP

Tesi di Laurea di:

Tito Costa

Relatore :

Chiar. mo Prof. Ing. **Luca Benini**

Correlatori:

Chiar. mo Prof. Ing. **Giovanni De Micheli**

Chiar. mo Prof. Ing. **Bruno Riccó**

Sessione III

Anno Accademico 2001-2002

A Lucia, Neria, Paolo

Parole chiave

SoC, System-On-Chip, NoC, Network-On-Chip, MPSoC, Multiprocessor System-On-Chip, parallel architectures, distributed memory, micronetwork, switched network, RTOS, embedded operating system, system software, middleware, MPI, Message-Passing Interface, IP, Intellectual Propriety, NI, Network Interface, routing layer, switching layer, physical layer, switch, channel, flow control, buffer

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Introduzione

Secondo le previsioni dell'*International Technology Roadmap for Semicondutors* per la fine del decennio i processi tecnologici saranno in grado di realizzare transistor con lunghezze di canale di 50nm, portando a 4 miliardi il numero di transistori contenuti in un unico chip, funzionanti ad una frequenza massima di 10GHz [2, 12]. La crescita esponenziale nella disponibilità di transistori, stimata al 58% annuo, è solo in parte compensata da un aumento di produttività degli strumenti di progettazione assistita dal calcolatore, ferma al 21% annuo [30]. Si è cioè venuto a creare un "divario di produttività" che rende ancora più critico affrontare un mercato molto competitivo, caratterizzato da requisiti di *time-to-market* stringenti e dall'esigenza di massimo ritorno sugli ingenti investimenti in tecnologia e ricerca e sviluppo [47, 34, 42].
I recenti sviluppi nel campo delle nanotecnologie inoltre lasciano prevedere un ancor più sostanziale aumento nella disponibilità su singolo chip di componenti logici elementari (*carbon nanotube* e *nanowire*) nei prossimi dieci anni [3].
Il paradigma di progettazione *Systems-On-Chip* (SoC) rende possibile l'implementazione di diverse unità funzionali (CPU, DSP, blocchi hardware dedicati, memorie, dispositivi di I/O) nello stesso circuito integrato. Tuttavia, per mantenere costante la produttività di progettazione SoC sarebbe necessario aumentare la percentuale dell'area della fetta di silicio dedicata a memoria *cache* dal 40-50% attuale ad oltre il 90% nel corso del prossimo

decennio [31], mentre una porzione sempre più marginale sarebbe destinata ai moduli che realizzano funzioni logiche.
Una innovazione metodologica nel paradigma di progettazione è rappresentata dai sistemi multiprocessore su singolo chip (MPSoC). Lo stesso blocco funzionale viene replicato più volte a creare un'architettura parallela; si può prevedere che i moduli elementari o *processing element* per la realizzazione dei dispositivi integrati di prossima generazione saranno rappresentati da processori. I sistemi multiprocessore nei prossimi anni potranno integrare fino a decine o centinaia di *processing element* sulla stessa fetta di silicio. Questa tecnologia è già oggi utilizzata per sistemi *embedded* ad alte prestazioni per applicazioni di telecomunicazioni e processori paralleli multimediali [51].
Il paradigma MPSoC propone nuove opportunità e problematiche ad ogni livello della progettazione del sistema, sia hardware che software: dal *layout* fisico ottimizzato per "strisce" di macroblocchi che si ripetono regolarmente ad un *design* modulare basato sul riuso di componenti o *intellectual property* (IP); dal progetto di un sistema di interconnessione efficace e scalabile tra i vari processori ad un livello di interfaccia tra questa infrastruttura di comunicazione e moduli hardware standard disponibili *off-the-shelf*.
Un elemento cruciale è rappresentato dall'architettura di comunicazione, sempre più vero "collo di bottiglia" dal punto di vista prestazionale dei sistemi su chip. Tradizionalmente i diversi componenti sono allacciati ad un bus condiviso; questo approccio, che si distingue per la sua semplicità e diffusione tra i progettisti, presenta alcuni limiti, destinati ad acuirsi con le tecnologie di prossima generazione.
In primo luogo, il bus condiviso soffre di una modesta scalabilità che ne limita l'uso al collegamento di pochi processori. Essendo una risorsa in grado di gestire una sola comunicazione per volta, il bus serializza le transazioni ponendo di fatto una seria limitazione al parallelismo della computazione e aumentando la latenza, ossia il tempo che intercorre tra la richiesta di accesso al bus e l'istante in cui esso viene concesso.
Un maggior numero di processori collegati ad un bus si traduce in una su-

periore capacità di carico sulla linea e di conseguenza in una ridotta efficienza in termini di consumo di potenza. Allo stesso tempo le dimensioni delle sezioni delle piste di collegamento fra i vari transistor e le distanze fra le piste stesse diminuiranno sempre più, comportando un aumento delle resistenze e delle capacità associate. Le tecniche attuali per ridurre il carico capacitivo e ridurre il consumo di potenza prevedono strutture a bus gerarchico, ma richiedono notevole sforzo progettuale ed hanno comportamenti elettrici e dinamici difficilmente predicibili [15, 2].
Problemi di *clock-skew* [1] e di interferenze elettromagnetiche saranno sempre più rilevanti, al punto da mettere in discussione due cardini della progettazione elettronica tradizionale: la presenza di un unico segnale di clock e linee di trasmissione senza errori. Su silicio si integreranno diverse "isole" di componenti sincroni fra loro ma senza una nozione di segnale temporale valido per tutto il chip, definendo cioè sistemi globalmente asincroni e localmente sincroni (GALS) [1]. Protocolli di sincronizzazione e di correzione d'errore saranno allora necessari per gestire comunicazioni tra diverse isole su linee di collegamento non affidabili.
Una architettura MPSoC richiede inoltre un supporto dedicato da parte del software di sistema: il sistema operativo *embedded* deve fornire *driver* di dispositivo e primitive che consentano al software applicativo di utilizzare le risorse di computazione parallela disponibili sul chip.
È necessario poi un ulteriore strato di software o *middleware* che implementi una interfaccia di comunicazione tra i processi utente distribuiti sui diversi processori. Sistema operativo e *middleware* saranno altamente configurabili per consentire di ottimizzare prestazioni e consumo di potenza per una particolare applicazione *embedded* [2].

Le attuali metodologie di approccio alla realizzazione di sistemi su chip (SoC) non permettono di risolvere efficacemente le problematiche esposte. In particolare non sarà più sufficiente usufruire della riusabilità di moduli hardware (*cores*), ma risulterà indispensabile potere disporre di strutture di connessione riconfigurabili e architetture parallele su singolo chip, allo scopo di ridurre gli sforzi di progetto.

Una nuova metodologia è costituita dall'utilizzo di microreti di interconnessione (*network-on-chip, NoC*) come architetture di comunicazione su chip contenenti numerose unità funzionali o *processing element* [2]. La rete di interconnessione su chip è una alternativa al bus condiviso, scalabile, modulare e riconfigurabile. La microrete contiene parametri da adattare sia al grado di integrazione del sistema, sia al tipo di applicazione che si vuole implementare.
I componenti fondamentali di una *network-on-chip* sono: la *network interface* ossia l'interfaccia tra unità funzionale (*IP core , Intellectual Property*) e rete, gli *switch* e i *link di collegamento* tra switch e network interface.
Per realizzare l'instradamento dei messaggi sulla rete tra i diversi nodi e garantire trasferimenti affidabili su linee imperfette, l'informazione viene pacchettizzata ed è implementato su chip uno *stack* multilivello di protocolli ispirato al modello OSI.
Una maggiore flessibilità e modularità, oltre al miglioramento delle prestazioni dovuto alla possibilità di condurre transazioni in parallelo, che il paradigma NoC può offrire, presentano però un costo: protocolli di rilevazione di errore e *handshake* causano un *overhead* di diversi cicli di *clock* sulla trasmissione di ogni messaggio tra un blocco funzionale e un altro, così come l'instradamento dei pacchetti da un nodo all'altro attraverso gli *switch* della rete.
Tuttavia limitare il confronto tra microrete e bus al livello hardware trascurerebbe una parte importante dell'interazione distribuita tra i *processing element* integrati sul chip. È opportuno invece estendere l'analisi all'intero sistema inteso come architettura parallela, dall'infrastruttura hardware ai servizi di comunicazione offerti da sistema operativo e *middleware*.
In questa tesi si presenta un ambiente di simulazione che si propone di analizzare i sistemi multiprocessore su chip attraverso tutti i livelli di astrazione, da quello applicativo allo strato hardware, attraverso il software di sistema. Si è prestata particolare attenzione all'interfaccia tra l'infrastruttura di microrete e i vari livelli software – sistema operativo, *middleware* e applicativo – e il relativo costo in termini di prestazioni.
L'obiettivo è quello di contribuire all'analisi delle architetture *network-on-*

chip proiettando le differenze dell'infrastruttura di comunicazione rispetto al tradizionale bus condiviso su un'ottica più ampia, che tenga conto anche del necessario supporto e *overhead* da parte del software di sistema.
Il software di analisi che è stato sviluppato nasce come un'estensione di *ML-RSIM*, simulatore che descrive in dettaglio il comportamento di nodi costituiti da un bus su cui sono collegati una CPU, memoria *cache* primaria e secondaria, RAM e dispositivi di I/O [40, 41]. Il processo di sviluppo si è svolto in due fasi: la definizione di modelli hardware che rappresentassero l'architettura su chip a rete di interconnessione e l'implementazione di uno strato di software che fornisse primitive di comunicazione distribuita.
Il simulatore è stato ampliato per poter simulare diversi nodi che interagiscano tra loro; ogni processore è dotato di memoria privata e l'interazione tra i diversi processi si realizza attraverso lo scambio di messaggi sulla rete a *switch* che collega i nodi. Non ci sono banchi di memoria condivisi e accessibili da più processori.
L'architettura parallela modellata dal simulatore rientra nella categoria dei sistemi multiprocessore a memoria distribuita [10]. L'integrazione dei processori e della rete che li collega su singolo chip presenta caratteristiche nuove in termini di latenza e prestazioni rispetto ad architetture parallele esistenti; queste peculiarità vanno tenute in considerazione in fase di sviluppo del software di sistema e delle funzioni di comunicazione.
Sono stati definiti modelli di microrete su chip e di dispositivi *network interface* che consentono ai nodi di allacciarsi alla rete. Il modello di microrete simula una interconnessione a *switch* con *wormhole routing* [15]; il simulatore fornisce parametri per descrivere il comportamento della rete quali larghezza di canale, latenza, *throughput*, dimensione dei buffer, lunghezza della pipeline di comunicazione.
Il dispositivo *network interface* rappresenta il punto di connessione tra il bus privato di ogni nodo e la microrete. È un modulo hardware mappato nello spazio di I/O del processore di ogni nodo che si occupa di spedire e ricevere messaggi con la rete, gestendo la pacchettizzazione, il controllo di flusso e un *pool* di buffer.
I modelli sono di tipo *behavioral*, descritti con linguaggi ad alto livello (C e

C++), senza scendere nel dettaglio di linguaggi *register transfer level* come Verilog o VHDL, per venire incontro all'esigenza di disporre di un simulatore veloce che consentisse di sviluppare e testare applicazioni parallele. Ultimata la definizione di una architettura hardware, si è studiato il sistema operativo fornito con *ML-RSIM* per estenderne le funzionalità ad un supporto multiprocessore. Il sistema operativo è basato su *NetBSD* e in misura minore su *Linux* [41, 26].
Per poter utilizzare l'interfaccia di microrete, il *kernel* del sistema operativo è stato integrato con un *driver* di dispositivo e chiamate di sistema per la trasmissione e ricezione di messaggi tra processori diversi. È stata sviluppata una versione del *driver* che si occupa direttamente del trasferimento dati e una versione che sfrutta la tecnologia *direct memory access* (DMA).
Si è poi realizzato uno strato software di *middleware* che si colloca tra il sistema operativo e le applicazioni utente e gestisce la comunicazione tra i diversi processori. Esso rende disponibile alle applicazioni un paradigma di programmazione a passaggio di messaggi implementando il nucleo dell'interfaccia MPI (*Message-Passing Interface*) che si è imposta come standard *de facto* nello sviluppo di applicazioni parallele [20, 19, 22]. Il *middleware* consiste di una libreria di primitive di comunicazione e di un sistema di gestione dei protocolli di *handshake*, della semantica di sincronizzazione e dei buffer software [10].
Si sono quindi adattate alcune applicazioni parallele tratte dalla letteratura [36] ed estratto statistiche sul comportamento del sistema, variando alcuni parametri quali la latenza dei pacchetti sulla rete, la larghezza dei canali di rete, la larghezza del bus, le politiche di *scheduling* del sistema operativo rispetto alle funzioni di comunicazione. Questi dati hanno consentito di estrarre informazioni utili per riconoscere i punti in cui si concentra l'*overhead* della comunicazione distribuita e identificare possibili strategie di ottimizzazione nell'architettura hardware e nel software di sistema.

Capitolo 2

Reti di interconnessione

Un sistema multiprocessore su chip (*MPSoC*) comprende CPU, memorie, moduli hardware dedicati e dispositivi di I/O su un unico circuito integrato. È costituito da tre blocchi funzionali:

- **IP (Intellectual Property)**: processori, memorie, moduli (DSP, *decoder/encoder* multimediali, funzionalità specifiche), dispositivi di Input/Output.

- **Infrastruttura di comunicazione**: permette lo scambio di *messaggi* fra i diversi IP; può essere realizzata come architettura a bus condiviso su cui si affacciano tutti i componenti oppure come rete di interconnessione – Interconnection Network o NoC ("Network-on-Chip")

- **NI (Network Interface)**: interfaccia fra il microprocessore e l'infrastruttura di comunicazione, la cui logica sia tale da trasformare i segnali di comunicazione del microprocessore (bus dati, bus indirizzi, segnali lettura/scrittura e così via, definiti da standard quali *AMBA bus* o *OCP*) in segnali utilizzabili dalla rete. [10]

L'architettura modellata dall'ambiente di simulazione presentato adotta una infrastruttura di comunicazione basata su reti di interconnessione, che presenta doti di scalabilità e riconfigurabilità superiori rispetto al

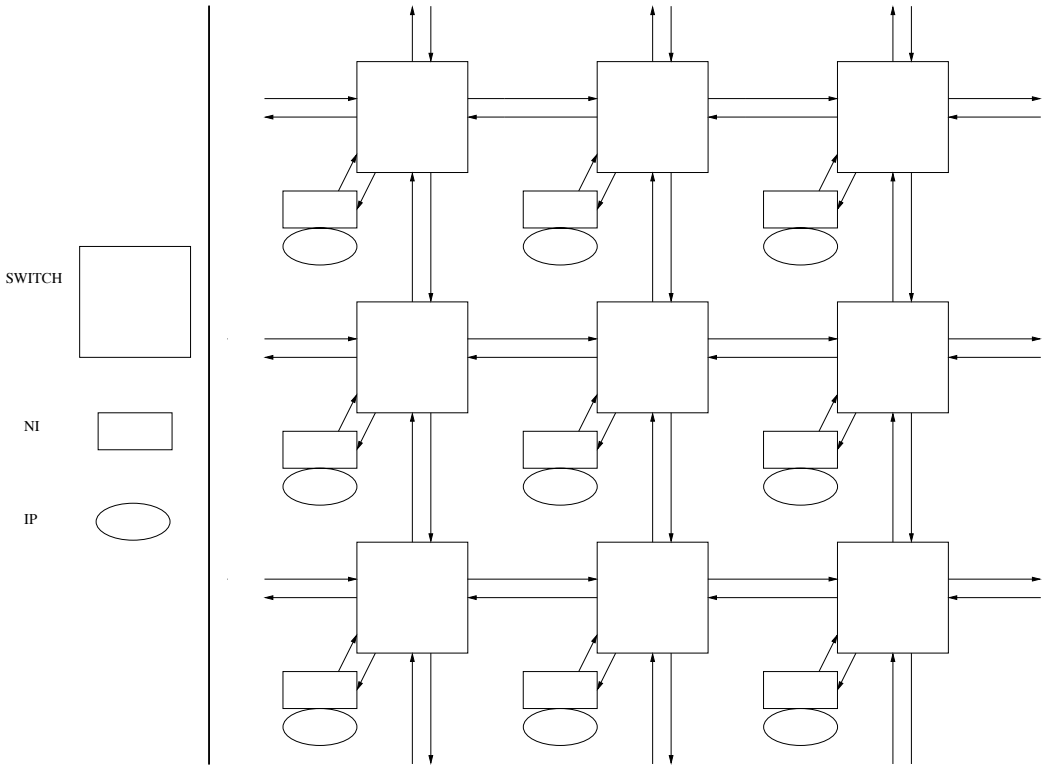


Figura 2.1: Soc: struttura mesh-2D.

bus condiviso [2]. Trattandosi di una rete che collega componenti funzionali all'interno di un solo circuito integrato, il sistema prende il nome di *Network-on-Chip* (NoC).

2.1 Reti di interconnessione

Le reti di interconnessione [15] si prestano a classificazioni diverse, basate sul modo di operare – sincrone, asincrone –, sul controllo di funzionamento – centralizzato, distribuito – o sulla topologia di collegamento – *direct networks*, *indirect networks*, topologie ibride.

2.1.1 Direct Network

Rete di collegamento costituita da un insieme di nodi, ognuno formato da processore, memoria locale e altre periferiche. I nodi sono collegati tra lo-

ro tramite *link* fisici, *channel* e hanno la funzionalità di veri e propri router. Questi ultimi possono essere connessi a quelli vicini formando così una maglia di collegamento. I canali di collegamento possono essere unidirezinali o bidirezionali. All'interno di ogni singolo nodo si possono distinguere canali interni, cioè quei canali che permettono la connessione diretta alla propria rete locale, e collegamenti esterni, che permettono il collegamento a nodi adiacenti. Ogni nodo può supportare un certo numero di canali di ingresso e di uscita.

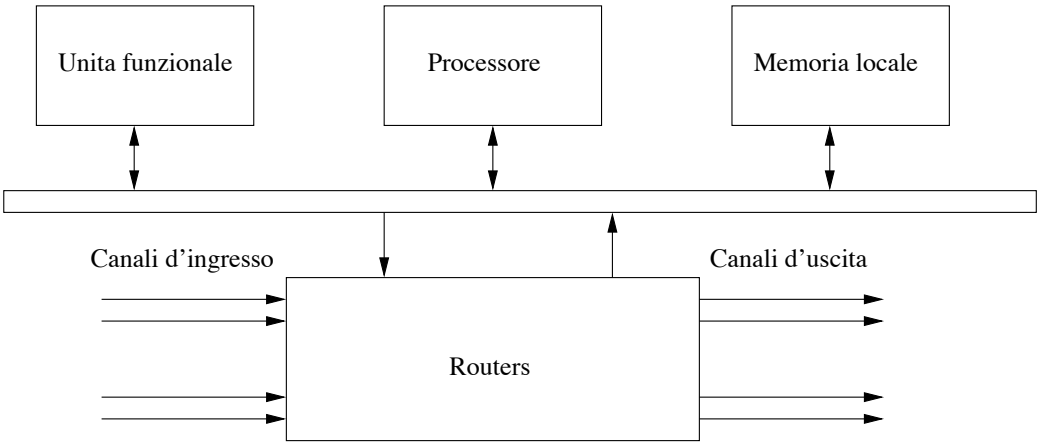


Figura 2.2: Architettura di un generico nodo.

Caratterizzazione delle Direct Network

- **Node degree:** rappresenta il numero di canali attraverso i quali il nodo è collegato a quelli vicini.
- **Diametro:** la massima distanza tra due nodi adiacenti.
- **Regolarità:** una network è regolare quando tutti i nodi hanno la stesso node degree.
- **Simmetria:** una network è simmetrica quando ogni nodo è uguale al vicino.

Una Direct Network è principalmente caratterizzata da tre fattori: topologia, routing e switching. La topologia definisce come i nodi sono connessi

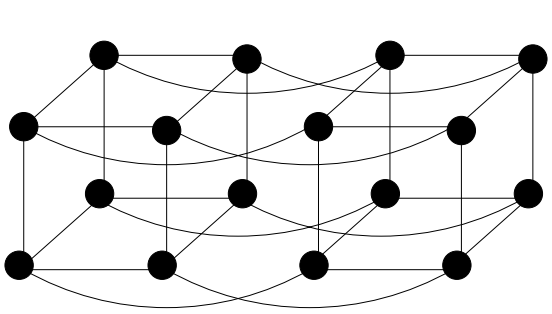
dai canali ed è normalmente interpretabile tramite grafi. Al crescere del numero di nodi che si collegano alla rete, i tempi di ritardo per attraversare la rete, l'area richiesta e di conseguenza il costo aumentano. L'unità di informazione scambiata è il *messaggio*, il quale può essere scomposto in più *pacchetti*. Questi pacchetti sono l'unità di informazione elementare che i diversi nodi possono scambiare. La politica di instradamento dei pacchetti (*routing*) che definisce come raggiungere il nodo destinazione dato quello corrente, determina le prestazioni del sistema. Il meccanismo di *switching* che si occupa dell'allocazione dei canali ed è implementato all'interno di ogni singolo router o *switch*, è a sua volta determinante nello studio delle prestazioni del sistema. È evidente che ogni switch necessita di memoria interna per non perdere informazioni utili. Si definisce controllo di flusso il meccanismo con cui switch adiacenti si scambiano informazioni. Ecco che per ogni tipo di rete vi sono politiche di routing, switching e controllo di flusso differenti, proprio per meglio addattare le caratteristiche del sistema alla trasmissione e ricezione dei pacchetti.

Topologie di Direct Network

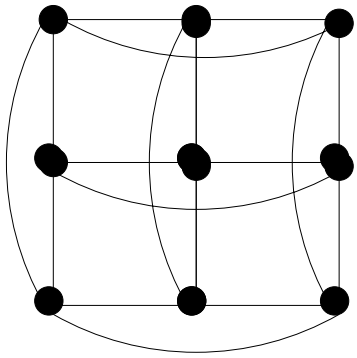
La topologia di una Direct Network si dice *ortogonale* se e solo se la rete di collegamento appartiene ad uno spazio n-dimensionale, ogni link di collegamento fra nodi appartiene ad una sola direzione e le direzioni possibili sono ortogonali tra loro. Le topologie ortogonali si possono poi ulteriormente suddividere in *strettamente ortogonali*, cioè dove ogni nodo è attraversato almeno da un link per ogni direzione dello spazio n-dimensionale e *debolmente ortogonali*, in cui possono mancare alcuni di questi link.
Altre topologie possono essere ad "albero binario" e ad "albero binario bilanciato".

2.1.2 Indirect Network

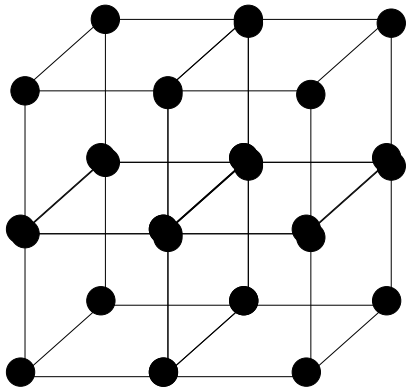
Le reti in cui ogni nodo rappresenta uno switch si dicono *indirect network* o *switch-based network*. L'informazione per raggiungere la propria destina-



(a) 2–ary 4cube (hypercube)

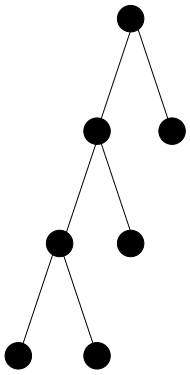


(b) 3–ary 2–cube

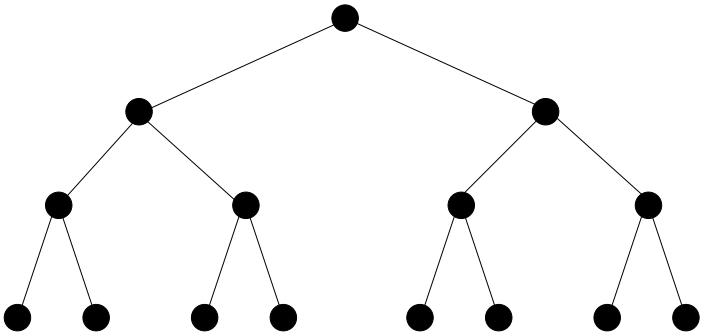


(c) 3–ary 3–D mesh

Figura 2.3: Strutture ortogonali.



(a)albero binario



(b) albero binario bilanciato

Figura 2.4: Strutture ad albero.

zione può attraversare diversi switch. Ognuno di essi ha un certo numero di *porte* attraverso le quali ha accesso al mondo esterno. Queste porte possono essere connesse, tramite *canali* o *link*, a switch vicini o a dispositivi di

interfaccia di rete (*network interface*). Le topologie possono essere *regolari* o irregolari.

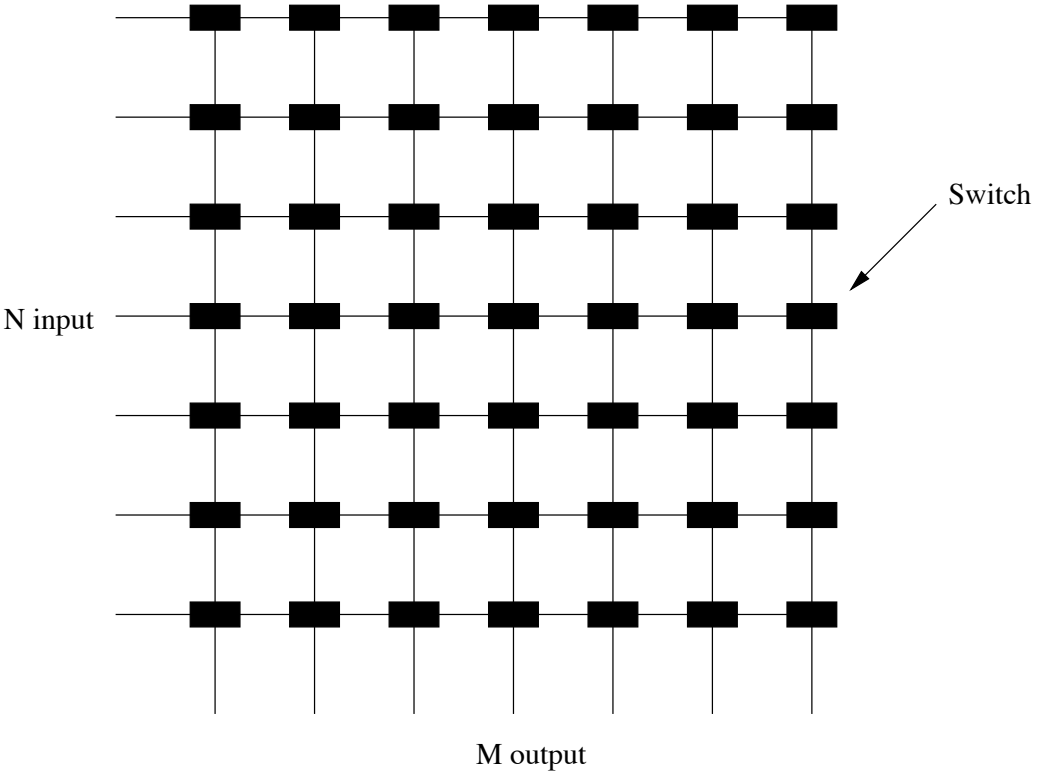


Figura 2.5: Struttura a crossbar.

Un esempio di topologia regolare è rappresentato dalla *crossbar*, costituita da uno switch con N input e M output (Figura 2.5). Si possono realizzare N,M interconnessioni senza conflitti. Normalmente N coincide con M. È chiaro che lo switch necessita di una unità *arbitro* che può essere distribuita all'interno dello switch e che permette di evitare collisioni di pacchetti (Figura 2.6).

In figura 2.7 sono riportati altri tipi di topologie di indirect network.

2.1.3 Topologie ibride

In figura 2.8 vengono mostrate due diverse tipologie ibride: ″a bus multiplo″ e ″Network a cluster″.

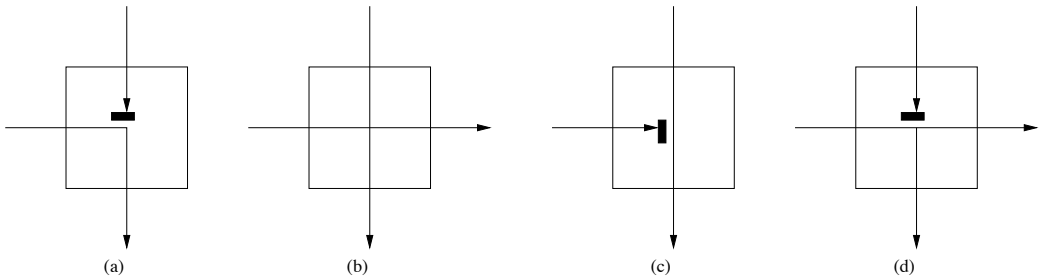


Figura 2.6: Combinazioni possibili ingresso-uscita.

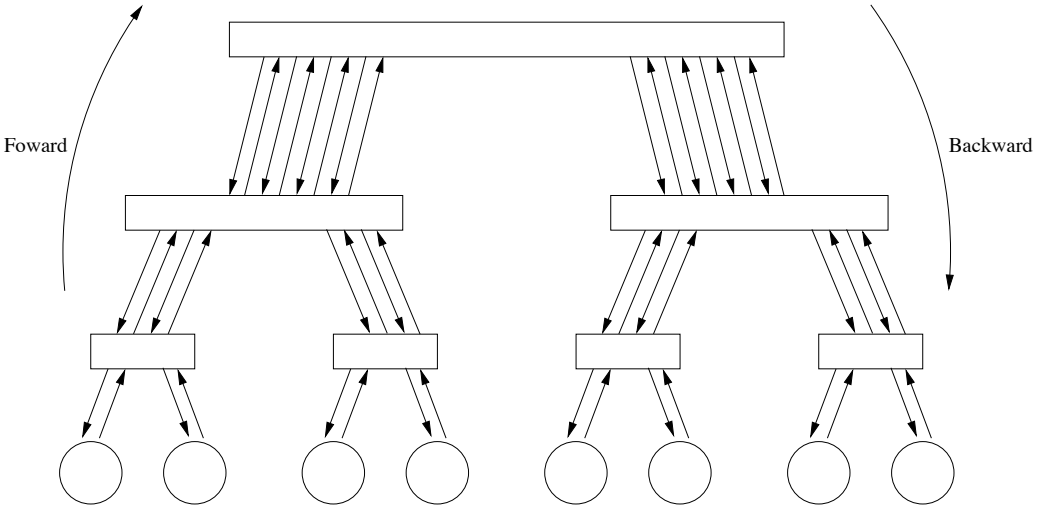
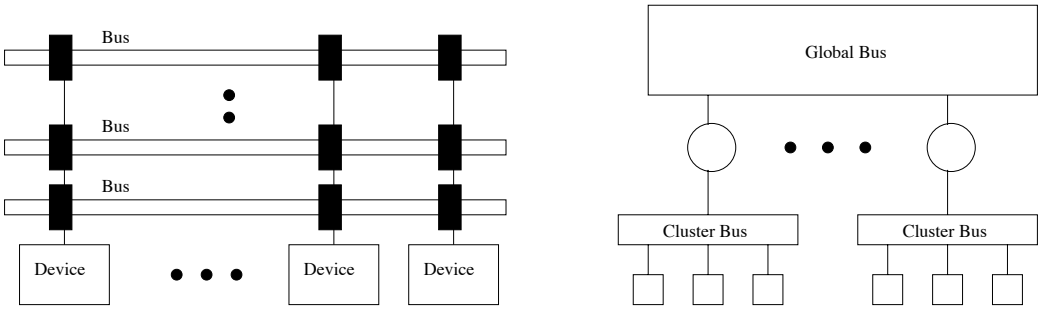


Figura 2.7: Struttura fat-tree.

2.2 Message switching layer

La comunicazione fra processori attraversa diversi *layer*: *routing layer*, *switching layer*, *physical layer*. Il physical layer è quel protocollo a livello fisico che trasferisce le informazioni attraverso i canali a switch adiacenti. Lo switching layer utilizza il physical layer per implementare la politica di spedizione dell'informazione attraverso la rete. Infine il routing layer determina il percorso che l'informazione dovrà seguire per raggiungere la destinazione. Le tecniche di *switching* determinano quali ingressi andranno in uscita. Queste tecniche sono strettamente legate al *flow control*, cioè al controllo di flusso delle informazioni scambiate fra switch adiacenti, e alla *gestione dei buffer*. Infatti ogni switch necessita di memoria locale per immagazzinare informazione che altrimenti andrebbero perse, ad esempio



(a) Network a bus multiplo

(b)Network a bus gerarchico a due livelli

Figura 2.8: Strutture Ibride.

perché il canale d'uscita è già stato impegnato.

2.2.1 Architettura di un generico switch

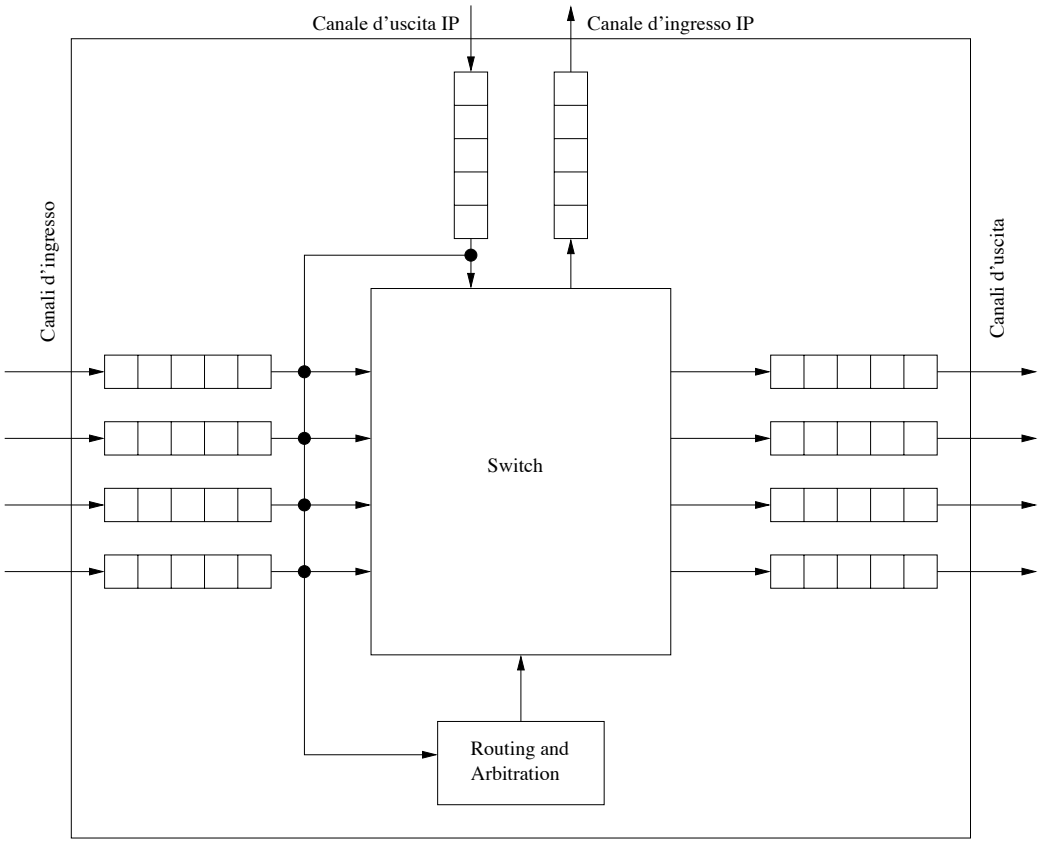


Figura 2.9: Architettura di un generico switch.

Come da figura 2.9, l'architettura di un generico switch è composta da

diversi elementi, quali:

- **Buffer:** possono essere FIFO (first input first output); se associati ad ogni ingresso, si parla di *input buffering*, se associati ad ogni uscita, di *output buffering*. Le loro dimensioni influiscono notevolmente sul costo di ogni singolo switch e sulle prestazioni del sistema.

- **Switch:** connette gli ingressi alle uscite. Può essere ad esempio una matrice crossbar a piena connettività che consente la connessione di ogni ingresso con ogni possibile uscita.

- **Routing e unità arbitro:** se allo stesso istante di tempo più ingressi richiedono la stessa uscita, quest'unità si incarica di scegliere quale ingresso potrà passare e quale invece dovrà attendere. Quest'ultimo può essere memorizzato internamente, nel buffer d'ingresso di quel canale, o dovrà essere rispedito, in base alla strategia adottata.

Si può identificare come *routing delay* il tempo impiegato dallo switch per instradare nella giusta uscita il dato arrivato in ingresso. L'*internal flow control latency* è invece il tempo di ritardo con cui il dato è disponibile in uscita a meno del ritardo di routing. Infine l'*external flow control latency* è il tempo di ritardo dei link fisici.

2.2.2 Messaggi, pacchetti, flit, phit

In generale ogni messaggio può essere diviso in pacchetti; questi sono divisi in *flit* che a loro volta sono costituiti da uno o più *phit*. Un phit è l'unità fisica di informazione che può viaggiare attraverso il link. Un flit invece rappresenta l'unità logica minima di controllo di flusso. In molti casi il phit coincide con il flit, convenzione adottata anche in questo lavoro. La figura 2.10 è un esempio di sincronizzazione fra due switch.

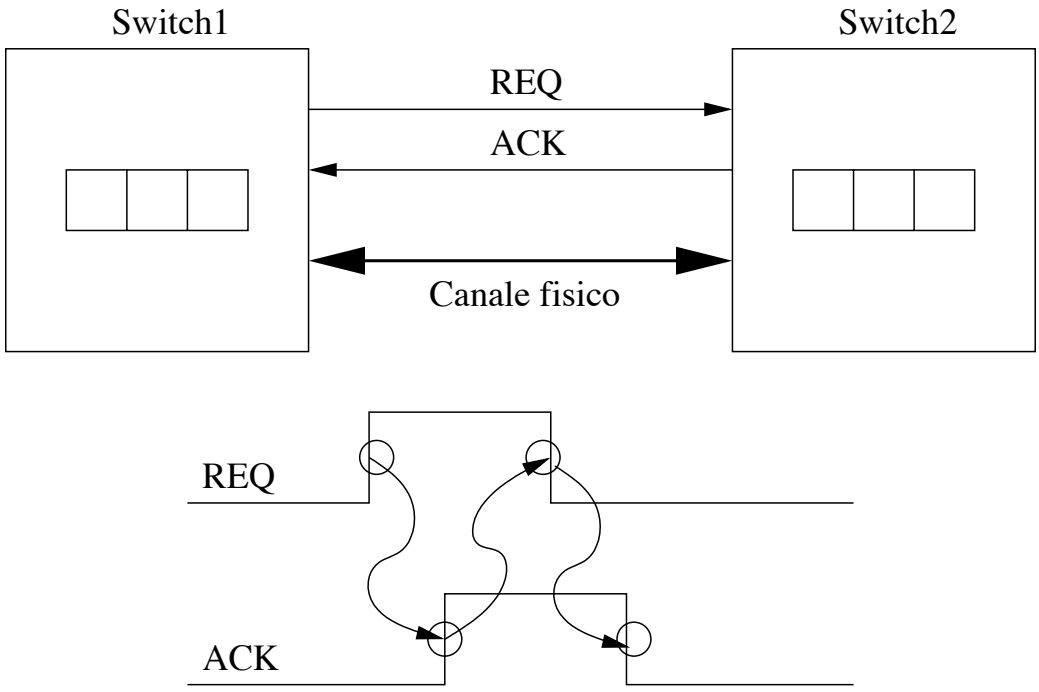


Figura 2.10: Esempio di flow control fra due switch.

2.2.3 Tecniche base di switching

Esistono diversi modi di gestire la comunicazione fra switch, ognuno con caratteristiche e proprietà differenti.

- **Circuit Switching:** il primo flit del primo messaggio, contenente le informazioni di routing, viene inviato alla rete. Ogni volta che il flit attraversa uno switch prenota il percorso fino ad arrivare alla destinazione. Un segnale di acknowledgment viene trasmesso alla sorgente e quando viene ricevuto, la sorgente spedisce a piena banda tutti gli altri flit lungo il medesimo circuito, che viene liberato solo dopo la conclusione del messaggio.

- **Packet Switching o Store-and-Forward:** il messaggio è diviso in più pacchetti, che vengono memorizzati completamente in ogni switch prima di essere ritrasmessi a quello seguente. La divisione del messaggio in pacchetti introduce *overhead* in termini di tempo e di spazio rispetto al circuit switching, in quanto ogni pacchetto viene instradato indipendentemente.

- **Virtual Cut Through switching:** mentre per il packet switching ogni nodo intermedio deve attendere l'arrivo di tutto il pacchetto prima di poter analizzare l'*header*, con questa tecnica i flit appena arrivati possono essere subito analizzati e se le risorse di comunicazione a valle sono libere, si può consentire l'attraversamento diretto degli ingressi in uscita.
- **Wormole Switching:** a differenza del virtual cut through switching, dove la bufferizzazione del pacchetto è ancora comunque necessaria, la bufferizzazione avviene attraverso stadi di pipeline nel link di collegamento fra switch.
- **Mad Postman Switching:** dato uno spazio n-dimensionale, i pacchetti vengono inoltrati secondo le diverse direzioni in maniera ortogonale, cioè prima percorrono tutta la prima dimensione, poi la seconda e così fino all'n-esima. Ad ogni passaggio di dimensione viene tralasciato il relativo flit di header.

2.2.4 Virtual channel

Assumendo che ad ogni canale di uscita sia associato un buffer FIFO, solo un flusso di dati può essere destinato a quel canale. La tecnica del *virtual channel* prevede che su ogni canale siano multiplexati diversi flussi di dati logici, ognuno dei quali gode di un buffer riservato, allo scopo di ridurre la contenzione delle risorse di rete. Come si può notare da figura 2.12, sia il flit A che il flit B devono passare attraverso la stessa porta d'uscita dello *switch 2*. Anche se il flit A risulta bloccato, B è libero di passare senza subire rallentamenti.

2.2.5 Tecniche ibride

Queste tecniche sono motivate dal desiderio di combinare i vantaggi delle diverse strategie di switching al virtual channel.

- **Pipeline Circuit Switching:** nel wormhole switching subito dopo l'header seguono i flit di *payload*; nel caso in cui l'header si dovesse

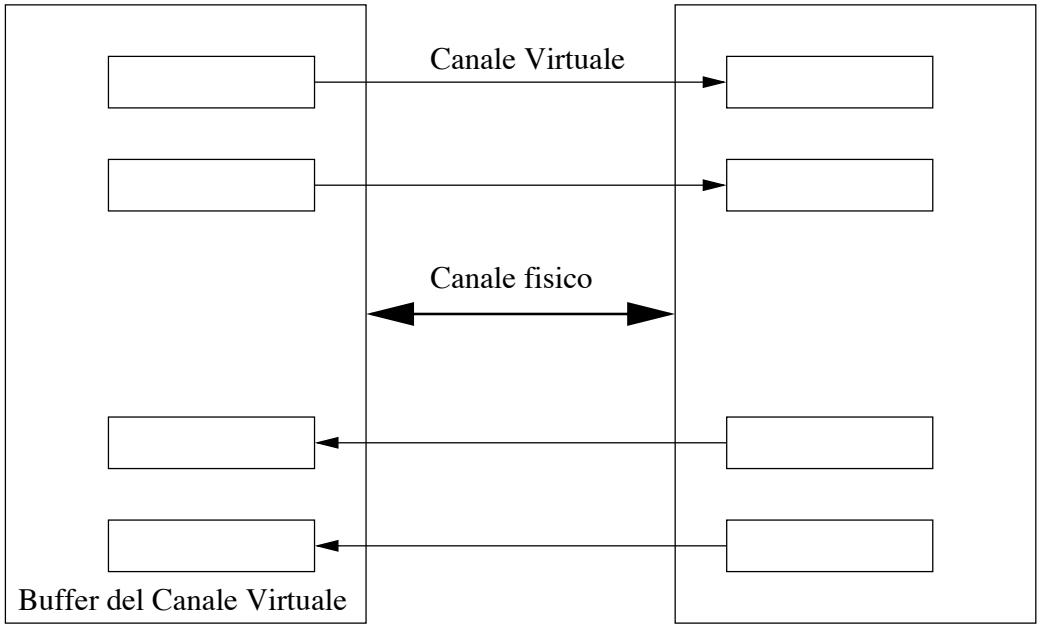


Figura 2.11: Canale fisico - Canale virtuale .

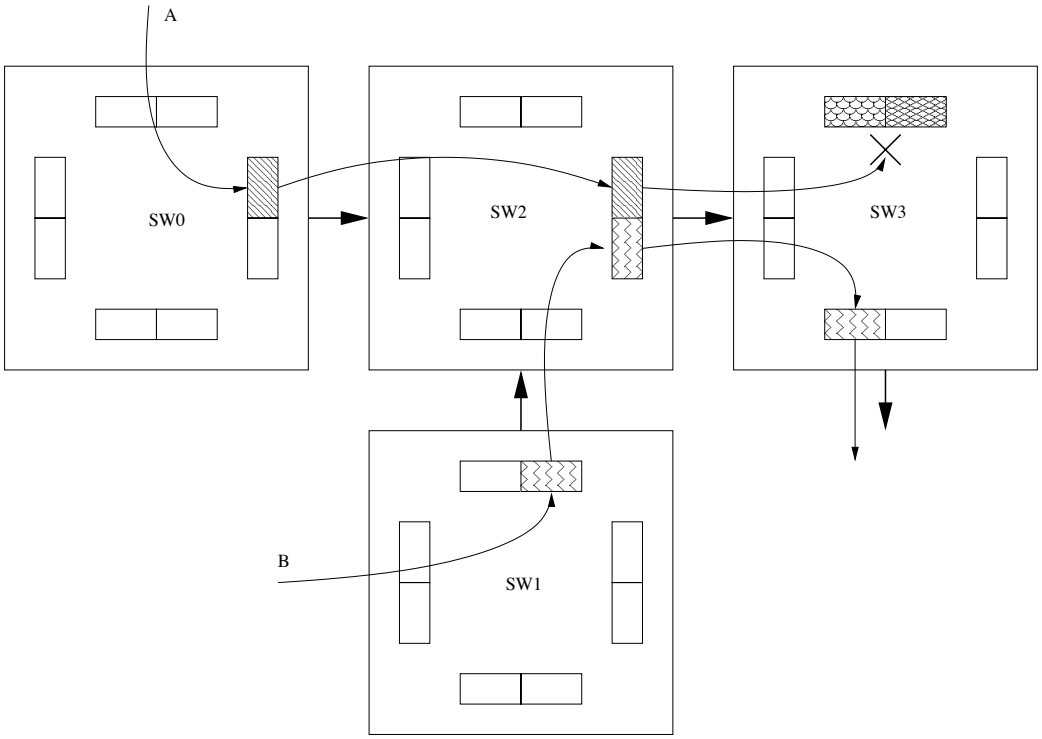


Figura 2.12: Esempio di funzionamento con Canale virtuale.

bloccare, anche tutti i flit al seguito si bloccherebbero lungo il canale *pipelined*, aumentando così il pericolo di *deadlock* sulla rete. Questo metodo serve a rendere meno probabile questa eventualità. Esso sfrutta le potenzialità del virtual channel, utilizzando la politica circuit switching. La sorgente invia il primo flit del pacchetto per riservare il circuito; i flit successivi vengono spediti solo dopo aver ricevuto un messaggio di *acknowledge* dalla destinazione. In questo caso le probabilità che il primo flit trovi il percorso libero aumentano, vista la disponibilità di più buffer per ogni canale fisico.

- **Scouting Swithing** A differenza del pipeline circuit switching i flit di payload seguono l'header a *distanza* K, con $K \in \mathbf{N}$ e $K \neq 0$, dove per distanza si intende la differenza fra il numero di switch attraversati dal flit di header e dal primo flit di payload. Con $K = 0$ si ricade nel pipeline circuit switching.

2.2.6 Considerazioni

Le tecniche di switching hanno un forte impatto sulle performance e sul comportamento della rete, in misura superiore della topologia e dell'algoritmo di routing. Infatti proprio dalle tecniche di switching dipendono sia la struttura interna dello switch che l'eventuale possibilità di prevedere la correzione d'errore e rendere affidabile il sistema di comunicazione.

2.3 Cenni su deadlock, livelock e starvation

Si verifica un *deadlock* quando un pacchetto non può più avanzare lungo il suo cammino, perché i buffer richiesti sono occupati e si è instaurato un ciclo di attesa tra pacchetti e buffer. Questo può accadere quando il nodo destinazione non consuma mai un pacchetto, ma in questo lavoro assumiamo che i pacchetti vengano sempre letti in un tempo finito. Sotto questa ipotesi il problema dell'eliminazione del *deadlock* va affrontato in sede di progettazione degli algoritmi di *routing*.

Una situazione diversa si verifica se alcuni pacchetti non sono in grado di raggiungere la destinazione, pur non essendo bloccati permanentemente. È possibile che un pacchetto giri intorno al proprio nodo destinazione senza mai raggiungerlo in quanto i canali necessari sono riservati da altri pacchetti, creando una situazione di *livelock*. Questa eventualità si può realizzare solo se i pacchetti possono eseguire percorsi non minimi.
Infine si verifica *starvation* quando un pacchetto rimane bloccato in modo permanente se il traffico è intenso e le risorse necessarie sono sempre assegnate ad altri pacchetti. La *starvation* può essere evitata semplicemente con uno schema corretto di assegnamento delle risorse.
Il problema di più difficile soluzione è di gran lunga il *deadlock*, che può essere affrontato con tre strategie. La *deadlock prevention* mira ad evitare il deadlock riservando tutte le risorse necessarie prima di iniziare la trasmissione di un pacchetto. La *deadlock avoidance* è meno restrittiva e cerca di riservare una risorsa alla volta man mano che i pacchetti avanzano nella rete, purché lo stato globale che ne risulta sia sicuro. Infine le tecniche di *deadlock recovery* assumono che l'evenienza di un *deadlock* sia molto rara e che le sue conseguenze siano accettabili, mirando a risolvere la situazione di stallo solo dopo che essa si è verificata.

2.4 Algoritmi di routing

Gli algoritmi di routing determinano quale dei possibili percorsi tra sorgente e destinazione viene utilizzato per instradare i pacchetti. Molte proprietà della rete dipendono dalla scelta dell'algoritmo:

- **Connettività**: capacità di instradare i pacchetti da un determinato nodo sorgente a un qualsiasi altro nodo destinazione.
- **Adattatività**: capacità di instradare i pacchetti per vie alternative in presenza di congestione del traffico.
- **Libertà da deadlock**: garantire che i pacchetti non si blocchino nella rete per un tempo indefinito.

- **Fault tolerance**: abilità di indirizzare pacchetti in presenza di errori.

Gli algoritmi di routing possono essere descritti in base a come e quando le decisioni di routing sono prese [11].

Come le decisioni di routing sono prese:

- *Algoritmo di routing deterministico:* data la coppia indirizzo sorgente, indirizzo destinazione, viene generato il percorso di routing, senza prendere in considerazione nessuna informazione relativa al traffico della rete. È consigliabile questa scelta quando la topologia utilizzata è di tipo *mesh* ($XY, XYZ, e-cube$), dove è semplice realizzare un routing *deadlock-free*. Topologie non uniformi invece richiedono un certo grado di adattività.

- *Algoritmo di routing semi-deterministico:* come il deterministico anche il semi-deterministico non tiene in considerazione nessuna informazione riguardante lo stato della rete e quindi del traffico. Si dice semi-deterministico perché per ogni destinazione sono possibili più percorsi da scegliere casualmente o in "maniera ciclica".

- *Algoritmo di routing adattativo:* vengono utilizzate le informazioni relative al traffico ed allo stato della rete per determinare il percorso di routing, allo scopo di evitare congestioni e/o errori di trasmissione dovuti a guasti nella rete.

Quando le decisioni di routing sono prese:

- *Source Routing:* il percorso che dovrà seguire un pacchetto è determinato al nodo sorgente e il compito dei diversi switch è solo quello di leggere nell'header di ogni pacchetto in quale uscita devono essere smistati i diversi pacchetti. Quindi se K è il numero delle uscite di ogni switch ed n è il numero di switch da attraversare , allora il numero di bit richiesti nell'header è almeno $n \log_2 K$. Questa tecnica riduce il numero di decisioni di routing per ogni pacchetto ad uno e rende particolarmente veloce l'instradamento dell'informazione nella rete minimizzando la latenza.

- *Routing distribuito:* la funzione di routing è calcolata in ogni switch per ogni pacchetto che lo attraversa. L'header flit contiene solo l'informazione dell'indirizzo della destinazione finale e spetta allo switch stesso scegliere verso quale uscita indirizzarlo.
  Una volta realizzata la rete, la topologia è fissata e ogni switch è a conoscenza degli switch adiacenti. Questo tipo di routing è favorito quando la topologia è simmetrica e regolare, in quanto gli switch adottano lo stesso algoritmo.

- *Routing ibrido:* a monte si calcolano solo alcune destinazioni parziali, più ovviamente quella finale. Gli switch provvedono nel dare la giusta direzione ai pacchetti che transitano in zone intermedie.

- *Routing centralizzato:* in questo caso la funzione di routing è stabilita da un controllore centralizzato.

Implementazione della funzione di routing

Ci sono due differenti approcci:

- **Macchina a stati finiti:** viene eseguito in software o hardware l'algoritmo mediante un automa a stati finiti. Quando la topologia di rete è configurabile dall'utente (mediante blocchi riconfigurabili) è possibile avere topologie irregolari; in questi casi è difficile derivare un algoritmo basato su una macchina a stati finiti ed è preferibile affidarsi ad una tabella di lookup.

- **Tabella di lookup:** il nodo sorgente o gli switch (a seconda che si tratti di source routing o routing distribuito) contengono una tabella di lookup in cui sono contenute tutte le informazioni di routing. Le lookup table alla sorgente contengono l'intero percorso che un pacchetto dovrà seguire e di conseguenza la dimensione della tabella dipende linearmente dalla dimensione della rete. Nel caso di routing distribuito le lookup table sono integrate nello switch e devono semplicemente contenere una memoria con un campo per ogni destinazione di uscita corrispondente ad ogni possibile ingresso.

2.5 Bandwitdh e latenza

Se ogni buffer ha capienza w flit e *signal rate* $\mathrm{f} = \frac{1}{\tau}$ dove τ è il periodo di clock, si ottiene una banda di canale *channel bandwitdh* $b = wf$. Se h rappresentano i flit di header e n quelli di payload si ha un'*occupazione di canale* $\frac{h+n}{b}$. Intuitivamente si può pensare alla banda effettiva come: $\frac{bn}{h+n}$. In realtà però bisogna anche tener conto del tempo di decisione dello switch e questo comporta la perdita di alcuni cicli temporali $w\Delta$. Quindi l'effettiva bandwith risulta essere: $\frac{b*n}{h+n+w\Delta}$. Più comunemente come parametro di confronto si utilizza la *bisection bandwitdh*, che rappresenta la somma delle bandwitdh di canale che se rimosse partizionerebbero la rete in due porzioni con uguale numero di nodi. La *latenza* è definta come il tempo

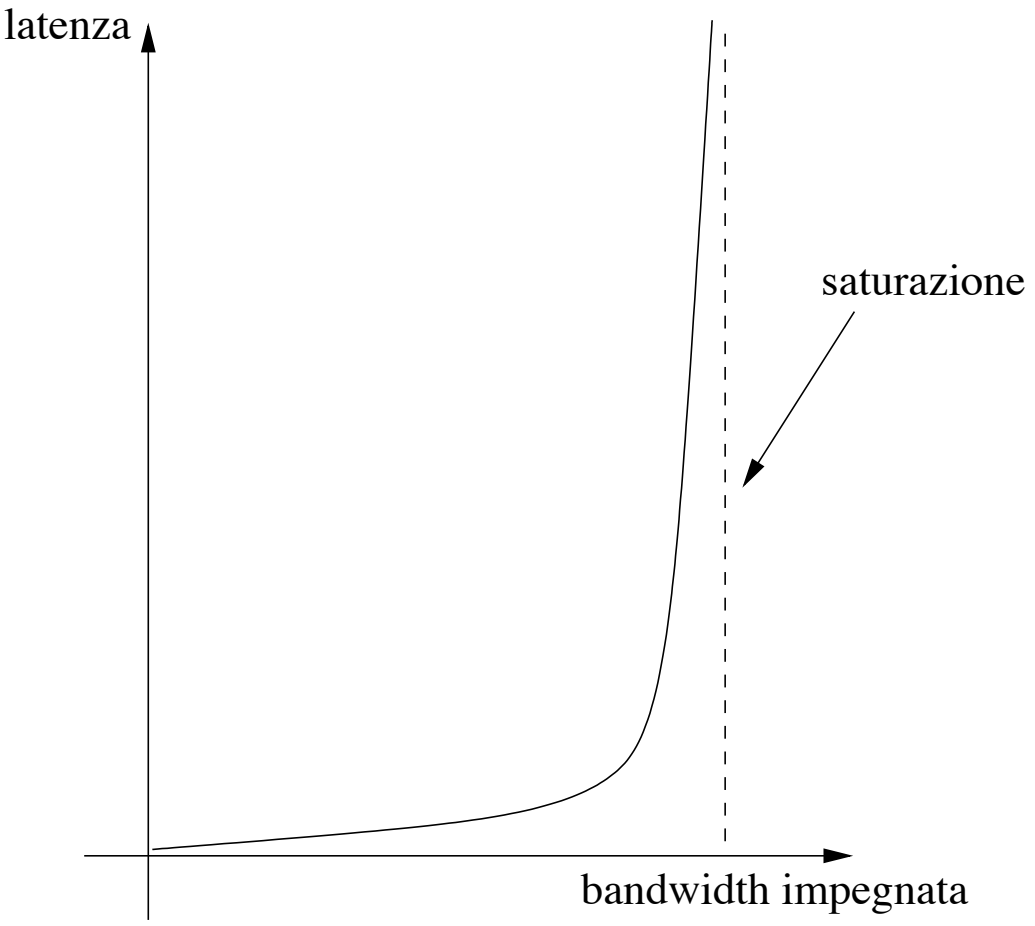


Figura 2.13: Bandwidth impegnata e latenza dei pacchetti.

impiegato da un pacchetto per raggiungere la destinazione. Questo tempo è legato alla percentuale di bandwith utilizzata: come si deduce dalla fig 2.13, all'aumentare della bandwitdh occupata la latenza dei pacchetti aumenta ripidamente.

Capitolo 3

Architetture e Paradigmi di Programmazione Paralleli

3.1 Introduzione

I sistemi multiprocessore su singolo chip dispongono di numerosi *processing element*, ognuno dei quali in grado di eseguire un flusso di istruzioni in parallelo agli altri. L'architettura hardware deve fornire una infrastruttura di comunicazione che consenta ai diversi processori di scambiare informazioni ed eseguire algoritmi distribuiti.
Inoltre è necessario definire un paradigma che descriva una metodologia di programmazione parallela, in particolare per specificare le modalità con cui a livello applicativo si possano definire le interazioni tra i processori.
Tra paradigma di programmazione software e infrastruttura hardware che collega i diversi nodi, si colloca uno strato software, detto *middleware*, che offre una prima astrazione dell' architettura sottostante, focalizzandosi sui servizi di comunicazione.
Esiste una vasta letteratura sulle architetture parallele, che pur disponendo di una piattaforma hardware diversa da quella qui presentata, presentano numerosi concetti che rimangono validi anche in questo contesto [10].
In questo capitolo si presenta una breve introduzione ad architetture (Sezione 3.2) e paradigmi di programmazione paralleli (Sezione 3.3), si citano alcuni esempi di soluzioni esistenti nel mondo dell'industria e accademi-

co e si posiziona la piattaforma multiprocessore su singolo chip sviluppata nell'ambito di questa categorizzazione (Sezione 3.4).

3.2 Architetture Parallele

In questa sezione ci occupiamo dell'infrastruttura hardware che mette in comunicazione i diversi processori (o *nodi*). In generale le architetture parallele si suddividono in base al sistema di memoria e a come esso è collegato ed accessibile ai processori.
Si distinguono architetture a **memoria condivisa**, **memoria distribuita** e **memoria condivisa distribuita**.

3.2.1 Memoria condivisa

Le architetture parallele a memoria condivisa sono caratterizzate dal fatto che tutti i processori hanno accesso ai banchi di memoria, che sono quindi condivisi.
La comunicazione avviene implicitamente come risultato di normali istruzioni di accesso alla memoria. Tutta la memoria disponibile nel sistema è mappata nello spazio di indirizzamento di ogni nodo che può accedervi direttamente grazie ad una infrastruttura di comunicazione.
Cooperazione e coordinazione tra i flussi di esecuzione si ottengono con letture e scritture di variabili condivise e puntatori che si riferiscono ad indirizzi condivisi.

Possibili infrastrutture di comunicazione sono una rete di interconnessione (in figura 3.1) o un bus comune (in figura 3.2). Utilizzando un bus condiviso, ogni processore è equidistante da ogni banco di memoria, quindi ogni processore è caratterizzato dallo stesso tempo di accesso, o latenza, ad una locazione di memoria.
Questa configurazione è chiamata *symmetric multiprocessor* (SMP) ed è impiegata in diversi prodotti commerciali, quali l'HP Netserver LX Series e il SUN Enterprise Server [10].

In presenza di memoria *cache* o tampone nei vari nodi, è necessario

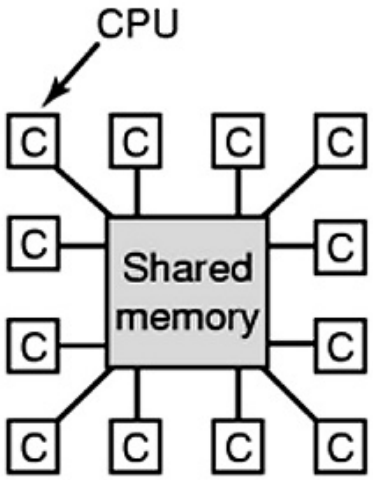


Figura 3.1: Memoria condivisa con rete di interconnessione

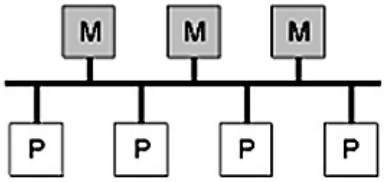


Figura 3.2: Memoria condivisa con bus comune

definire un protocollo di accesso alla memoria che garantisca la *coerenza* dei dati nel sistema (Sezione 3.2.5).

3.2.2 Memoria distribuita

Le architetture parallele a memoria distribuita, o *message passing*, presentano N nodi, ognuno dei quali dotato di processore, memoria *cache* e RAM privata del nodo e un sistema di I/O. Non esistono locazioni di memoria condivise; al contrario ogni banco è mappato nello spazio di indirizzamento di un solo processore.
La comunicazione tra processori avviene come operazioni di I/O esplicite. Questo stile di progetto presenta analogie con le *network of workstations* (NOW) o *clusters*, in quanto pone in collegamento computer completi, ma

la scala di integrazione è più compatta.

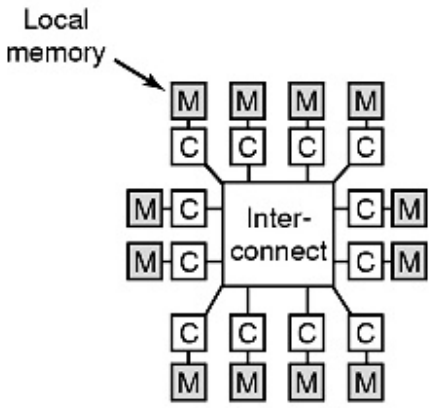


Figura 3.3: Memoria distribuita con rete di interconnessione

Anche in questo caso è possibile fare uso di diversi schemi di interconnessione, basati su rete (figura 3.3) oppure su bus comune (figura 3.4). In generale un dispositivo *direct memory access* (DMA) gestisce i trasferimenti tra il dispositivo di I/O collegato all'infrastruttura di comunicazione e le memorie private dei diversi processori.

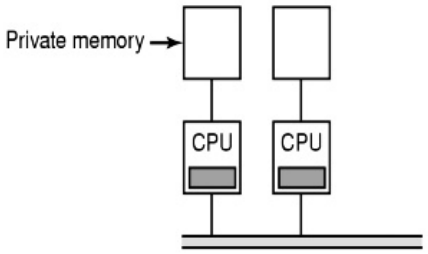


Figura 3.4: Memoria distribuita con bus comune

I processori appartenenti a nodi diversi comunicano scambiandosi messaggi e trasferendo dati sull' infrastruttura di comunicazione.
Soluzioni commerciali basate su architetture a memoria distribuita sono l'IBM SP-2 e l'Intel Paragon.

3.2.3 Memoria condivisa distribuita

Si tratta di una architettura parallela simile a quelle con memoria distribuita, in cui sono presenti banchi di memoria RAM privata in ogni nodo. In questo caso però uno strato hardware si occupa del passaggio di messaggi tra i nodi, offrendo ai livelli superiori una astrazione a memoria condivisa, in cui la comunicazione è integrata nel sistema di memoria, piuttosto che nel sistema I/O come nei sistemi a memoria distribuita.
Ogni nodo contiene una *directory* che associa indirizzi di memoria ai processori. Se l'indirizzo di memoria corrisponde a un banco di RAM locale, il trasferimento avviene all'interno del nodo; se invece l'indirizzo si riferisce a memoria appartenente ad un altro nodo, l'infrastruttura attiva transazioni per accedere trasparentemente al dato in questione.
Questa modalità ànche nota come *non-uniform memory access* (NUMA) in quanto una istruzione di accesso alla memoria può comportare sia un veloce trasferimento locale al nodo che iniziare una transazione sulla rete di interconnessione tra i nodi [10].
Esempio commerciali di questa architettura sono rappresentati dal Cray T3E e l'SGI Origin.

3.2.4 Network of Workstations

Invece di utilizzare costosi componenti dedicati e architetture proprietarie, un sistema *network of workstations* (NoW) consiste di una collezione di *workstation* o *personal computer* collegati in rete. NoW prevede l'impiego di soli componenti *off-the-shelf* per le unità di computazione (normali PC) e la rete di interconnessione (Ethernet o Myrinet) per massimizzare il rapporto prezzo/prestazioni.
Dal punto di vista del modello di memoria, le NoW rientrano chiaramente nella categoria delle architetture a memoria distribuita, in quanto interconnettono computer indipendenti, ognuno dei quali dotato della propria memoria RAM.

3.2.5 Coerenza della memoria cache

Il modello di memoria più intuitivo deve garantire che una lettura ad una locazione restituisca il valore più recente scritto in quella locazione. Questa è la proprietà fondamentale di una astrazione di memoria utilizzata in programmi sequenziali, nei quali si usa la memoria per comunicare un valore da un punto nel programma in cui esso è calcolato ad altri punti in cui esso è utilizzato. La stessa proprietà è utilizzata in una architettura con spazio di indirizzamento condiviso per comunicare dati tra processi in esecuzione sullo stesso processore. In questo caso il *caching* non interferisce perché tutti i processi accedono alla memoria attraverso la stessa gerarchia di *cache*.
Sarebbe auspicabile poter contare sulla stessa proprietà anche nel caso in cui più processi siano in esecuzione su due processori diversi che condividono una memoria. Tuttavia quando due processi vedono la memoria condivisa attraverso *cache* diverse, c'è il rischio che uno veda il nuovo valore nella propria *cache* mentre l'altro vede ancora il vecchio valore.

Il comportamento descritto in figura 3.5 è in contrasto con il modello di memoria delineato in precedenza. In realtà problemi di coerenza della *cache* emergono anche in sistemi uniprocessore in corrispondenza di transazioni I/O. La maggior parte di operazioni I/O avviene con accessi diretti alla memoria (DMA) da parte di dispositivi che trasferiscono dati tra la memoria e le periferiche senza l'intervento del processore. Quando un dispositivo DMA scrive in una locazione di memoria, il processore potrebbe continuare a vedere il valore precedente memorizzato nella sua cache, a meno che non si provveda con azioni particolari. In presenza di *cache write-back*, un dispositivo DMA potrebbe leggere una locazione non valida dalla memoria di sistema perchè il valore più aggiornato per quella locazione si trova nella cache del processore. In questo caso per sistemi uniprocessore si sono spesso adottate soluzioni drastiche grazie ad una minore frequenza delle operazioni di I/O rispetto agli accessi in memoria: ad esempio si possono marcare i segmenti di memoria destinati all'I/O come *uncacheable*, o scaricare le cache in memoria esplicitamente attraverso

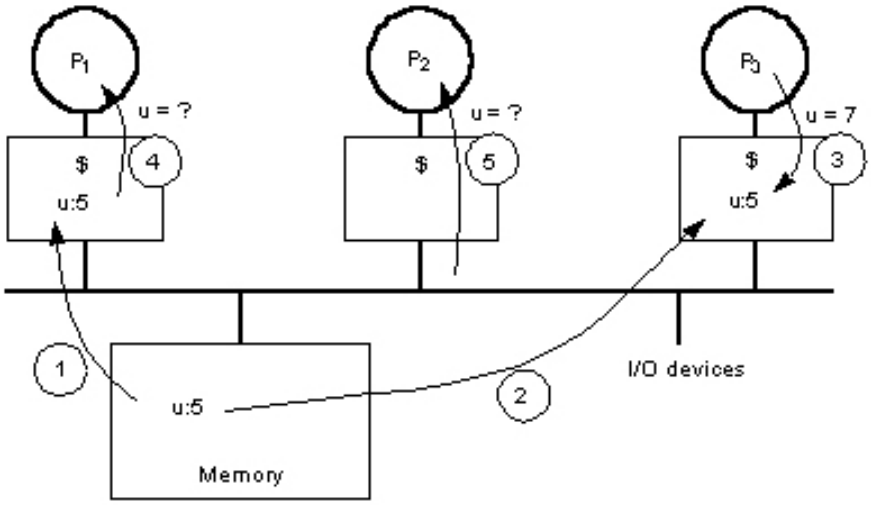


Figura 3.5: La figura mostra tre processori connessi da un bus alla memoria principale. u è una locazione in memoria il cui contenuto viene scritto e letto dai processori. La sequenza delle operazioni è indicata dai numeri cerchiati. È evidente che senza opportune precauzioni quando P_3 aggiorna il valore di u a 7, P_1 continuerà a leggere il valore scaduto dalla propria *cache*, così come P_2.

il supporto del sistema operativo prima di iniziare una transazione I/O.
In sistemi multiprocessore a spazio di indirizzamento condiviso, la lettura e scrittura di variabili condivise da processori diversi sono operazioni frequenti in quanto questo rappresenta il supporto di intercomunicazione per applicazioni parallele. È quindi necessario affrontare la coerenza delle cache come un problema a livello di progettazione del sistema hardware. Nel caso di architetture basate su bus condiviso, una tecnica per implementare in hardware un protocollo di coerenza della cache è il *bus snooping*: grazie al fatto che il bus è accessibile e visibile a tutti i componenti, ogni controllore di cache è in grado di monitorare tutte le transazioni sul bus, riconoscere gli accessi alla memoria che coinvolgono una locazione presente in cache e prendere opportuni provvedimenti, come invalidare o aggiornare la linea di cache.
Se invece la memoria è distribuita nel sistema, come nel caso delle macchine NUMA, l'approccio più comune si chiama *directory-based cache coherence*. Lo stato dei blocchi di cache non può essere determinato impli-

citamente tramite *bus snooping*, ma va mantenuto esplicitamente in una struttura dati, la *directory*, all'interno di ogni nodo. Il protocollo di coerenza per mantenere aggiornate le *directory* si realizza con transazioni sulla rete di interconnessione con messaggi di richiesta, risposta, invalidazione, aggiornamento, *acknowledge*.

3.3 Paradigmi di programmazione

Ad un più alto livello di astrazione si colloca la definizione di un paradigma di programmazione, che esprime come sviluppare applicazioni parallele utilizzando le risorse di computazione e comunicazione della piattaforma hardware.
I paradigmi più comuni sono *memoria condivisa* e *message passing*.
Anche se la scelta più naturale è quella di utilizzare un paradigma a memoria condivisa su una architettura a memoria condivisa (o memoria condivisa distribuita) e un paradigma a passaggio di messaggi su una piattaforma a memoria distribuita, è possibile ipotizzare soluzioni diverse.
Uno strato di *middleware* adatterà il paradigma di programmazione alle caratteristiche ed esigenze della piattaforma.

3.3.1 Memoria condivisa

Le applicazioni parallele dichiarano locazioni di memoria condivisa tra i diversi processori con chiamate di sistema quali *shared malloc* (`shmalloc()`) [37]. Gli algoritmi in questo caso si basano sulla definizione di variabili globali condivise nel sistema e su meccanismi di sincronizzazione per garantire una interazione coerente ed evitare corruzione di dati.
È necessario dunque gestire meccanismi di mutua esclusione e *locking* distribuito su diversi processori per permettere ad una sola applicazione per volta di accedere alle locazioni condivise. In presenza di risorse che vengono riservate da diversi processi si presenta l'eventualità di fenomeni di *deadlock*, che vanno accuratemente studiati per essere prevenuti, evitati o eliminati a posteriori dal software di sistema.

A livello applicativo, l'accesso alla memoria locale o al banco associato ad un diverso nodo non presenta alcuna differenza: in ogni caso il programmatore utilizza una variabile dichiarata "globale" [7]. La trasparenza è garantita da *middleware* e infrastruttura hardware (ad esempio nel caso di architettura NUMA).
Se da un lato la programmazione a memoria condivisa rende molto intuitiva e trasparente l'accesso a variabili globali da diversi processori, dall'altro ha un comportamento più difficilmente predicibile: la stessa operazione di accesso alla memoria può infatti usufruire di un dato conservato in *cache* oppure generare transazioni verso un nodo remoto, dando origine a latenze e prestazioni molto differenti.

3.3.2 Message Passing

Le operazioni a livello applicativo più comuni sono varianti di `send` e `receive`. Una `send` specifica un buffer di dati locale che deve essere trasmesso e un processo, tipicamente in esecuzione su un processore diverso, destinato a riceverlo. Una `receive` specifica un processo mittente e un buffer di dati locale su cui collocare il messaggio trasmesso. `send` e `receive` corrispondenti provocano un trasferimento dati da un processo all'altro.
Generalmente una `send` consente anche di specificare un `tag` e un `datatype` per ogni messaggio, a cui confrontare le richieste di `receive` [20]. La combinazione di `send` e `receive` produce inoltre un evento di sincronizzazione tra i due processi e una copia da memoria a memoria, in cui ogni *endpoint* di comunicazione specifica il suo indirizzo locale [10].
Si possono specificare diversi tipi di semantiche di sincronizzazione, in base al comportamento di queste funzioni rispetto alla bufferizzazione e al trasferimento dati sulla rete. La `send` si dice *sincrona* se completa quando la `receive` è stata eseguita e un messaggio di *acknowledge* è stato ricevuto dal mittente; in questo caso è implicito un effetto di sincronizzazione tra i processi sorgente e destinazione, anche noto come *rendez-vous*. La `send` si dice *bloccante* se ritorna solo dopo che una richiesta di `receive` cor-

rispondente è stata inviata dal processo destinazione; se invece la `send` completa al termine della copia in un buffer di sistema del messaggio in spazio utente, essa viene definita *bufferizzata* (o *buffered*).
`send` e `receive` si dicono poi *non bloccanti* se queste ritornano subito dopo aver iniziato il trasferimento da o verso i buffer di sistema, affidandolo in generale ad un disposivo hardware dedicato quale un controllore DMA. Per verificare l'avanzamento del trasferimento sono disponibili funzioni di `probe` [10].
Nell'ambito del modello di programmazione a passaggio di messaggi si è imposto uno standard che definisce sintassi e semantica di una collezione di funzioni di comunicazione: *message-passing interface* (MPI) [20, 22]. Si vedano Sezione 3.4.1 per un'introduzione allo standard e Sezioni B.2.1, B.2.2 per una descrizione di come MPI è stata utilizzata nel nostro modello di MPSoC.

3.4 Architetture Multiprocessore su Singolo Chip

Questa tesi presenta un simulatore per sistemi multiprocessori su singolo chip (MPSoC). Nello stesso circuito sono integrati N nodi, ognuno dei quali presenta al proprio interno un bus locale al quale sono collegati un processore con memoria cache di primo e secondo livello, banchi di memoria RAM e un dispositivo I/O di *network interface*. I vari nodi sono collegati da una microrete di interconnessione, come mostrato in figura 3.6.

L'architettura simulata rientra dunque nella categoria a *memoria distribuita* e la comunicazione tra processori avviene attraverso scambi di messaggi gestiti della *network interface* di nodo sorgente e destinazione.
Il simulatore sviluppato è basato su *ML-RSIM* (Sezione 4.5) a cui sono state apportate significative modifiche per modellare rete di interconnessione e interfacce di rete (Sezione 4.6). Dal punto di vista del paradigma di programmazione, si è adottato il modello *message passing*, che si adatta naturalmente alla infrastruttura hardware a memoria distribuita.
Il *middleware* che si è sviluppato per l'architettura multiprocessore su chip

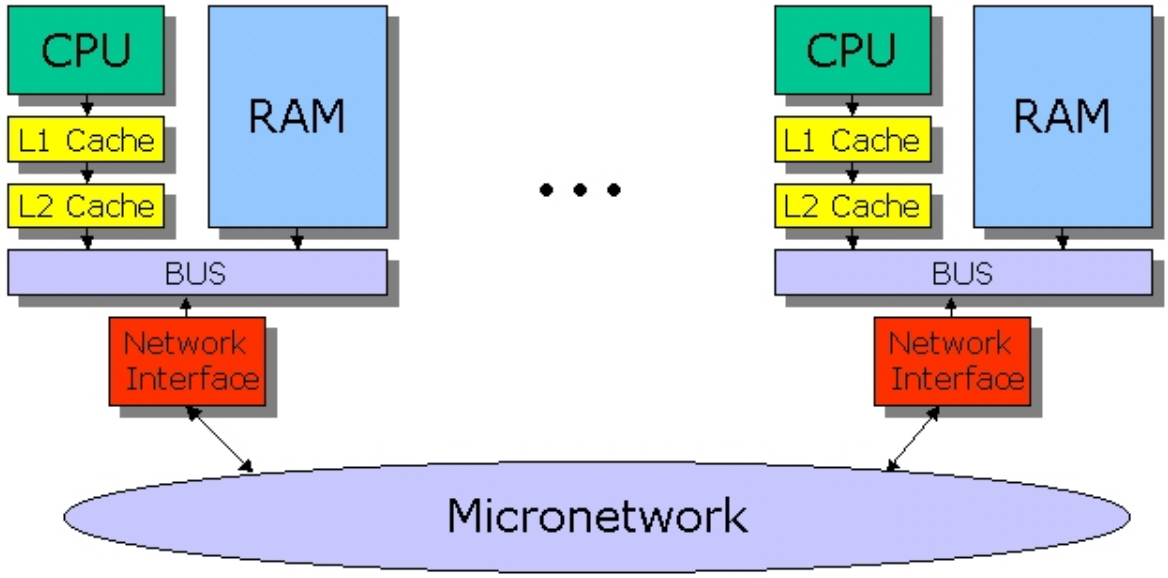


Figura 3.6: Modello di architettura sviluppato

utilizza un *subset* di funzioni dell'interfaccia standard MPI, come spiegato nelle sezioni B.2.1 e B.2.2. Si veda la sezione 5.3 per strategie e dettagli implementativi del *middleware*.

3.4.1 Message-Passing Interface

Message-Passing Interface (MPI) è una interfaccia standard per macchine parallele con modello di programmazione a scambio di messaggi. La prima bozza, noto come MPI-1, fu presentata nel 1992 dall'MPI Forum – un consorzio delle maggiori aziende del settore, centri di ricerca e università –, seguito da MPI-1.1 nel 1995 e MPI-2.0 nel 1997.
Il principale obiettivo di progetto di MPI risiede nell'aumento delle prestazioni attraverso la sovrapposizione di computazione e comunicazione. MPI definisce un insieme di *routines* e astrazioni di alto livello – o *Application Programming Interface* (API) – allo scopo di massimizzare la portabilità e la facilità di utilizzo di programmi paralleli tra diverse piattaforme hardware e linguaggi di programmazione. L'interfaccia è poi implementata utilizzando algoritmi di passaggio di messaggi *machine-dependent* a più basso livello su una varietà di architettue parallele come nCUBE Vertex,

p4 and PARMACS, Intel NX/2, Cray T3D and T3E, IBM SP2, Thinking Machines CM-5, NEC Cenju-3 [20, 19]. Nel nostro caso, la piattaforma hardware sottostante è un sistema multiprocessore a memoria distribuita, come modellato dal simulatore sviluppato (Sezione 4.6).
Lo standard MPI definisce primitive di comunicazione *point-to-point* e collettiva (*broadcast, gather, all-to-all*), gruppi di processi, contesti di comunicazione, topologie virtuali di processi, *bindings* per C e Fortran 77, gestione e scoperta dell'ambiente MPI e una interfaccia per il *profiling*.
Le diverse semantiche esposte in sezione 3.3.2 sono definite dall'interfaccia; lo sviluppatore può inoltre richiedere un controllo più preciso della gestione delle risorse di *buffering*, specificando dimensioni delle code e locazioni in memoria.
MPI include un sistema di tipi di dato indipendente dal linguaggio utilizzato, con regole di conversione e *casting* tra tipi diversi; è inoltre possibile definire tipi complessi come *array* e strutture. Con MPI-2 sono stati introdotti supporto per C++ e Fortran 90, creazione e gestione dinamica di processi, comunicazioni *one-sided* come le operazioni per memoria condivisa e I/O parallelo.

Capitolo 4

Simulatori multiprocessore

4.1 Introduzione

Lo sviluppo dell'ambiente di simulazione è iniziato con la ricerca di simulatori per architetture parallele già disponibili in ambito accademico o commerciale e con l'analisi delle loro caratteristiche di configurabilità per poter rappresentare un modello fedele di un sistema multiprocessore su chip.
In questo capitolo sono presentati i simulatori di multiprocessori più diffusi e utilizzati (Sezioni 4.2 - 4.5) e il simulatore per MPSoC che abbiamo sviluppato (Sezione 4.6).

4.2 RSIM

RSIM è un simulatore disponibile pubblicamente per sistemi a memoria condivisa costruiti con processori che sfruttano aggressivamente il parallelismo a livello di istruzioni (ILP), come MIPS R10000, Hewlett-Packard PA8000 e Intel Pentium Pro presentati all'inizio degli anni '90. Questi processori avevano per la prima volta la possibilità di ridurre gli stalli dovuti a letture in memoria sovrapponendo la latenza di lettura con altre operazioni, cambiando la natura dei colli di bottoglia nelle prestazioni del sistema. RSIM (*Rice simulator for ILP multiprocessors*) fu quindi sviluppato presso

Rice University per indagare l'impatto della microarchitettura del processore sulle prestazioni di sistemi *shared-memory*, cercando di mantenere le simulazioni efficienti e veloci. Il codice di RSIM è stato rilasciato nel 1997 e distribuito gratuitamente per utilizzo non commerciale.
Il modello di processore di RSIM è simile al MIPS R10000, mentre l'*instruction set* scelto è quello di Sparc V9. Il modello può essere configurato su diversi livelli di utilizzo di ILP: da *single-instruction issue, in-order instruction scheduling* e operazioni di memoria bloccanti a *multiple-instruction issue, out-of-order instruction scheduling* e operazioni di memoria non bloccanti. Per quanto riguarda il sistema di memoria e *cache*, RSIM supporta un gerarchia di *cache* a due livelli; il modello consente l'*interleaving* e *pipelined split transaction* sul bus.

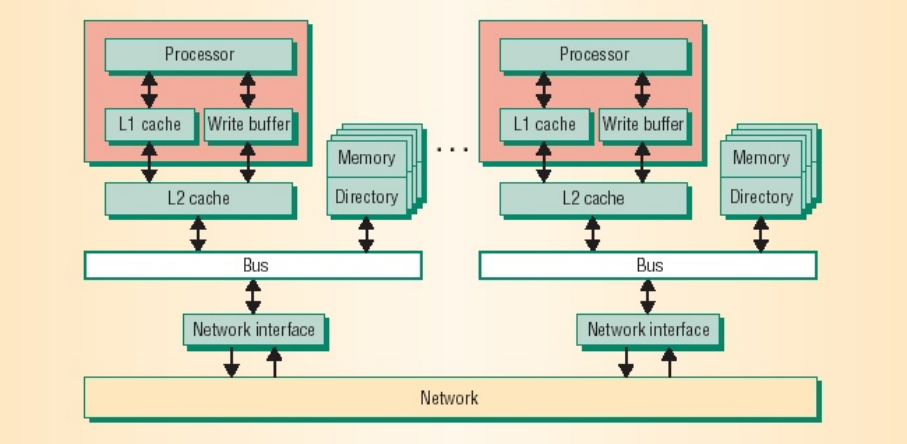


Figura 4.1: Modello di architettura in RSIM

Il sistema multiprocessore modellato è a memoria condivisa distribuita (NUMA) basato su *directory* con un protocollo di coerenza della cache. Come mostrato in figura 4.1, ogni nodo del sistema consiste di un processore, la gerarchia di cache L1 e L2 insieme ad una parte della memoria fisica, la sua *directory* e una *network interface*. Un bus a *split-transaction* connette la cache L2, la memoria, il modulo *directory* e la *network interface*.
Per la comunicazione tra i nodi, RSIM supporta una rete a *mesh* bidimensionale con *wormhole routing*. I protocolli di *cache coherence* supportati sono *modified, exclusive, shared, invalid* (MESI) e *modified, shared, invalid* (MSI).

RSIM simula applicazioni compilate e collegate su piattaforma Sparc V9/Solaris, salvo il fatto che il processore modellato è a 32 bit e gli interrupt software o *trap* non sono standard UNIX. È quindi necessario utilizzare librerie proprietarie che implementano le funzioni più comuni della *C standard library*. Inoltre tutte le chiamate di sistema vengono semplicemente emulate, non simulate; in altre parole l'interazione tra applicazioni e software di sistema e tra software di sistema e piattaforma hardware viene ignorata, mentre il sistema di I/O non è suppportato [27, 37]. Si noti che è proprio questa l'interazione – a livello di sistema – su cui il nostro ambiente di simulazione vuole soffermare la propria analisi. RSIM è disponibile per sistema operativo Solaris su workstation Sun.

4.3 SimOS

SimOS è stato progettato presso Stanford University nel 1992 per l'analisi efficiente ed accurata di sistemi uniprocessore e multiprocessore; esso simula la piattaforma hardware con tale dettaglio da lanciare un intero sistema operativo ed applicazioni inalterate su di esso, ed offre una notevole flessibilità nel *tradeoff* tra velocità e precisione della simulazione.
Questo è possibile grazie ad una simulazione della piattaforma hardware estremamente veloce, che permette di eseguire applicazioni anche meno di dieci volte più lentamente rispetto all'esecuzione nativa. Inoltre è possibile variare in modo flessibile il dettaglio dei modelli dei diversi componenti hardware, anche durante la simulazione; ad esempio in fase di boot del sistema operativo si possono utilizzare modelli più veloci per poi passare ai modelli dettagliati durante l'esecuzione dell'applicazione sotto esame.
SimOS simula l'hardware di una workstation SGI (Silicon Graphics Inc.) con precisione sufficiente ad eseguire il sistema operativo Irix. Carichi di lavoro di applicazioni sviluppate su piattaforma SGI/Irix possono essere simulati senza alcuna modifica.
SimOS comprende inoltre modelli di altri componenti hardware quali dischi magnetici, interfaccia seriale, interfaccia Ethernet, timer e interprocessor interrupt controller. La architettura parallela simulata si basa sul

multiprocessore DASH di stanford, un sistema NUMA a memoria condivisa distribuita con protocollo di coerenza della cache basato su *directory*. La rete di interconnessione non è però simulata come in RSIM. SimOS era disponibile per piattaforma SGI, ma non sembra essere un progetto attivo. [38, 39, 32]

4.4 Simics

Simics è un simulatore di architetture di *instruction set* a livello di sistema, includendo cioè componenti hardware utilizzati tipicamente solo da sistemi operativi. Simics è un prodotto commerciale, ma disponibile gratuitamente per istituti di ricerca e università, realizzato da Virtutech, Inc. Il progetto è oggi attivo e supportato.
Simics è dotato di caratteristiche simili a SimOS, ma è in grado di simulare una vasta gamma di architetture (Alpha, SPARC, Intel, ARM, MIPS, PowerPC) in diverse configurazioni. Come SimOS, Simics può lanciare sistemi operativi completi (come SUN Solaris, Linux, Tru64, VxWorks, Windows NT ed XP) senza modificazioni garantendo compatibilità binaria con l'architettura simulata.
È anche possibile modificare modelli hardware e aggiungere moduli personalizzati; sono supportate architetture multiprocessore SMP, a *clusters* e reti di calcolatori. Simics fornisce una funzionalità di *checkpoint* che consente di salvare il suo stato su file per ricaricarlo successivamente e procedere la simulazione dallo stesso punto. Simics è disponibile su diversi sistemi *host* (Linux, Solaris, Windows) [49].

4.5 ML-RSIM

ML-RSIM è stato sviluppato dal Prof. Lambert Schaelicke (University of Notre Dame) e Mike Parker (University of Utah) e pubblicato in novembre 2002. Si tratta di un simulatore accurato al ciclo di clock che integra un modello dettagliato di processore (derivante da RSIM, con *instruction*

set SPARC), modelli di memoria e cache e un sottosistema I/O completo. L'architettura hardware di ML-RSIM, pur essendo rappresentativa di una serie di sistemi workstation e server moderni, non modella nessun particolare sistema ma racchiude caratteristiche comuni di diversi sistemi. In questo senso, il simulatore non garantisce compatibilità binaria con alcuna piattaforma reale e l'eventuale *porting* di un sistema operativo è operazione molto complessa perché sarebbe necessaria una conoscenza molto approfondita della struttura unica del simulatore.
Con ML-RSIM è già integrato un sistema operativo Unix compatibile chiamato Lamix, basato in parte sul codice sorgente di NetBSD v1.4.3b. ML-RSIM utilizza l'interfaccia Solaris per chiamate di sistema, quindi applicativi compilati staticamente su macchine *host* Sun Solaris possono eseguire senza modifiche sul simulatore. La principale caratteristica distintiva di ML-RSIM rispetto ad RSIM è quella di fornire una piattaforma completa di bus e sistema I/O in grado di simulare un sistema operativo.

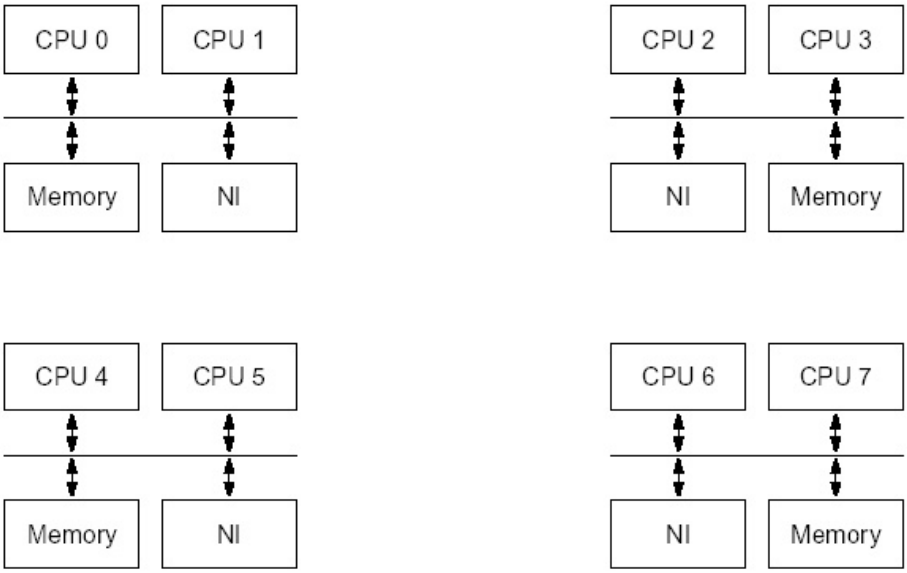


Figura 4.2: Modello di architettura in ML-RSIM

D'altra parte è stata eliminata dal modello la rete di interconnessione e i nodi sono indipendenti e isolati. Pur supportando N nodi, ognuno

dei quali con M processori al suo interno, ML-RSIM non è adatto alla simulazione di architetture parallele per l'assenza di una infrastruttura di comunicazione tra i vari nodi e l'assenza di supporto multiprocessore nel sistema operativo Lamix. ML-RSIM è disponibile con il codice sorgente su host Solaris, SGI e Linux [40, 41].

4.6 Simulatore per MPSoC

L'architettura hardware che si vuole modellare è un sistema multiprocessore su singolo chip (MPSoC). I diversi processori sono collegati da un microrete di interconnessione con *wormhole routing*. L'elevato grado di configurabilità e precisione – oltre all'esperienza raccolta con questo simulatore presso Stanford University in studi precedenti [51] – hanno fatto propendere inizialmente la scelta su RSIM. La piattaforma simulata, che include già un modello dettagliato di rete tra i nodi, presenta una notevole analogia con il modello *Network-On-Chip* (NOC) ed è facilmente riconfigurabile per rispecchiare i parametrici tipici dell'elevato livello di integrazione degli MPSoC.
L'ambiente RSIM originale non si occupa dell'analisi del software di sistema, limitando a semplice *emulazione* le funzioni del sistema operativo; ma l'obiettivo di questa tesi è L'idea iniziale è stata quindi quella di utilizzare un sistema operativo embedded multiprocessore e colmare il *gap* che RSIM lascia tra i programmi applicativi e il modello hardware. Si è allora deciso di effettuare il *porting* del sistema operativo RTEMS sull'architettura simulata da RSIM [7, 6, 8, 9] come descritto in appendice C.
Tuttavia la piattaforma RSIM non presenta compatibilità binaria con alcun sistema reale – pur implementando quasi completamente l'*instruction set* SPARC – e manca del sottosistema I/O e delle istruzioni di interruzione hardware (*interrupt* e software (*trap*). È chiaro dunque che RSIM non si presta per la simulazione "di sistema" dalla piattaforma hardware al livello applicativo senza tralasciare sistema operativo e *middleware* di comunicazione [27].
All'inizio di novembre 2002, è stato rilasciato un simulatore basato su

RSIM completo di codice sorgente, *ML-RSIM*. ML-RSIM introduce nel modello hardware le caratteristiche necessarie ad eseguire un sistema operativo completo e su di esso qualsiasi applicazione compatibile. Un sistema operativo UNIX-compatibile è già incluso nell'ambiente di simulazione ed è in grado si eseguire applicazioni compilate e *linkate* staticamente per Solaris/SPARC.
Se da un lato ML-RSIM ha il vantaggio decisivo di integrare un sistema operativo e di consentire la simulazione di transazioni con dispositivi di I/O sul bus, come accennato precedentemente (Sezione 4.5), esso non include alcun supporto di comunicazione tra i nodi, né a livello di modello hardware né di sistema operativo.
Il modello di piattaforma multiprocessore ML-RSIM è stato allora integrato per simulare una architettura MPSoC come rappresentato in figura 3.6. Il simulatore è stato sviluppato su piattaforma Solaris/SPARC. ML-RSIM è disponibile anche per LINUX e i moduli hardware aggiunti necessitano solo di essere ricompilati per portare tutto l'ambiente su Linux; tuttavia questa configurazione non è stata testata. Il simulatore richiede comunque che sistema operativo, *middleware* e applicazioni utente vengano compilati in ambiente Solaris con target SPARC.
Alla descrizione delle nuove caratteristiche implementate in ML-RSIM è dedicato l'appendice A. I componenti fondamentali che sono stati introdotti in ML-RSIM sono:

Rete di interconnessione (*micronetwork*) Modello di rete a *switch* integrata su chip con *wormhole routing*. La rete è completamente connessa e non sono dettagliate risorse e contenzione a livello di porte e buffer degli switch. La bassa utilizzazione effettiva della rete che si è riscontrata nella simulazione dell'intero sistema hardware e software giustifica l'adozione di questa ipotesi semplificativa (Sezione 6.7). Si veda la sezione A.3 per maggiori dettagli.

Interfaccia di rete (*network interface*) Componente hardware connesso al bus locale di ogni nodo, rappresenta il punto di raccordo tra i nodi individuali e la microrete di connessione. La *network interface* è un

dispositivo mappato nello spazio di indirizzamento I/O del processore e gestito da un *device driver* del sistema operativo. Si veda la sezione A.4 per maggiori dettagli sull'implementazione del modulo e dei suoi blocchi funzionali.

Il supporto di comunicazione tra nodi si basa su uno *stack* di protocolli ispirato ai livelli OSI; l'utilizzo di protocolli su più *layer* consente di gestire in modo configurabile e versatile la trasmissione fisica (*physical layer*), la gestione di errore sui canali non affidabili (*datalink layer*), il *routing* dei pacchetti sulla rete di *switch* da un nodo all'altro (*network layer*), il controllo di flusso e sincronizzazione (*transport layer*). A più alto livello (*system* e *application layer*), il software di comunicazione (*middleware*) gestisce protocolli di *handshake* e la trasmissione di dati utente. Si veda la sezione A.2 per ulteriori dettagli.

Capitolo 5

Software di sistema

5.1 Introduzione

I sistemi multiprocessore su singolo chip (MPSoC) costituiscono una classe nuova di architetture parallele, il cui alto grado di integrazione presenta opportunità e criticità nuove che suggeriscono un cambiamento di metodologia nella progettazione dei livelli software.
In questo capitolo si presenta la struttura del sistema operativo (in Sezione 5.2) e del *middleware* che offre i servizi di comunicazione (in Sezione 5.3).

5.2 Sistema operativo

Integrato al simulatore di architetture parallele *ML-RSIM* è fornito un sistema operativo basato su *NetBSD* e in misura minore su *Linux*, chiamato *Lamix* [41]. Tuttavia, a causa dell'interesse degli sviluppatori di ML-RSIM per l'analisi dei carichi di input/output piuttosto che per sistemi multiprocessore, Lamix non offriva alcun supporto per la gestione di più nodi o processori.
Per poter utilizzare il modello di MPSoC supportato dal simulatore che è stato sviluppato, il *kernel* del sistema operativo è stato ampliato come descritto in dettaglio in appendice A.5; in particolare le estensioni realiz-

zate si sono concentrate nella definizione di un *driver* di dispositivo e di chiamate di sistema per supporto multiprocessore.

5.2.1 *Driver* di dispositivo

I vari nodi del sistema MPSoC sono collegati alla microrete di interconnessione attraverso un dispositivo di I/O collegato al bus locale del nodo, detto *network interface*. È stato quindi necessario integrare nel sistema operativo un *driver* di dispositivo che consentisse agli strati software superiori di dialogare con l'interfaccia di rete. Il *device driver* si occupa semplicemente di trasferire dati dalla memoria ai buffer interni dell'interfaccia di rete in due modalità, non-DMA e DMA.
Nel primo caso è la CPU che trasferisce dati con coppie di istruzioni `LOAD` e `STORE`; nel secondo caso la *network interface* viene programmata con indirizzo fisico di memoria e lunghezza del messaggio ed è il dispositivo che si occupa del trasferimento mentre il processore è libero per proseguire il flusso di istruzioni. L'accesso al *device driver* è riservato allo strato di *middleware* ed avviene mediante chiamate di sistema UNIX.
I dettagli implementativi sono riportati in appendice A.5.1.

5.2.2 Supporto multiprocessore

Per consentire allo strato di *middleware* di conoscere il processore e il nodo sul quale un processo è in esecuzione sono state sviluppate due chiamate di sistema `getnode()` e `getprocessor()`.
`getnode()` riporta l'identificativo del nodo di esecuzione, un numero intero da 0 a $N-1$, dove N è il numero di nodi nel sistema. *ML-RSIM* consenta di instanziare M processori all'interno di ogni nodo; `getprocessor()` riporta l'identificativo della CPU di esecuzione nell'ambito di un nodo, un numero intero da 0 a $M-1$. Si noti che nella nostra configurazione, in cui è presente un solo processore in ogni nodo, unna chiamata a `getprocessor()` restituisce sempre 0.
Si veda l'appendice A.5.2 per informazioni più dettagliate sull' implementazione di queste primitive.

5.3 Middleware

Il *middleware* è uno strato di software che si colloca tra le applicazioni utente e il sistema operativo implementando le funzionalità di comunicazione tra processi in esecuzione sullo stesso processore o su processori distinti.
Nel primo caso, la comunicazione avviene secondo le modalità di *interprocess communication* in ambiente UNIX, attraverso locazioni di memoria condivise nello spazio di indirizzamento virtuale dei processi.
Nel secondo caso, in presenza di un modello di programmazione a scambio di messaggi (Sezione 3.4), vengono avviate transazione sulla rete di interconnessione attraverso il dispositivo di *network interface*. Il *middleware* accede alla *network interface* attraverso il *device driver* incluso nel sistema operativo (Sezione 5.2).
Lo strato di *middleware* rende disponibili i propri servizi alle applicazioni utente attraverso una interfaccia standard, *message-passing interface* (MPI) come descritto in appendice B.2.2.

5.3.1 Blocchi funzionali

L'architettura del *middleware* consiste di:

- strutture dati *mailbox* per messaggi in uscita dai diversi processi
- un buffer per messaggi in entrata dalla rete
- un processo che esegue in modalità *daemon* che si occupa di trovare richieste `send` e `receive` corrispondenti e di gestire il protocollo di comunicazione a livello applicativo.

Una descrizione dettagliata dei blocchi funzionali del *middleware* è disponibile in appendice B.4.

5.3.2 Accesso concorrente

Il sistema software progettato consiste di un ambiente multiprogrammato e multiprocessore; in altre parole ci sono numerosi processi in esecuzione

su ognuno degli N processori del MPSoC. Questa situazione rende necessaria una gestione attenta delle risorse di sistema (quali code e buffer di messaggi) per una esecuzione coerente del codice, garantendo un accesso *mutuamente esclusivo* tra i diversi flussi di esecuzione.
È stato sviluppato un sistema di *locking* di risorse basato su semplici *spin locks* che utilizzano operazioni di lettura e scrittura "atomiche" in memoria (Appendice B.8). In presenza di meccanismi di *locking* di risorse emerge il problema della gestione di eventuali *deadlock*, ovvero situazioni di attesa indefinita di processi per risorse occupate. Il *middleware* garantisce che solo una risorsa alla volta sia riservata per un processo, prevenendo la possibilità di cicli di attesa circolare (Appendice B.9).

5.3.3 Paradigma di programmazione

Il *middleware* implementa una libreria a scambio di messaggi, esportando al livello applicativo superiore il nucleo fondamentale dell'interfaccia *MPI* [20, 19, 22]. Lo sviluppatore di applicazioni parallele utilizza primitive `MPI_Send()` e `MPI_Recv()` per indicare esplicitamente una comunicazione tra processi; nell'identificativo di processo è inoltre indicato il nodo di esecuzione, quindi il progettista ha piena visibilità delle interazioni internodo. È stata implementata la `MPI_Send()` in semantica bufferizzata e sincrona e la `MPI_Recv()` bloccante (Appendice B.11).
Il *middleware* realizza la sincronizzazione sulla rete tra processi sorgente e destinazione utilizzando un protocollo di *handshake receiver-initiated* come dettagliato in appendice B.11.2.

Capitolo 6

Risultati

6.1 Introduzione

Sono state utilizzate quattro applicazioni parallele tratte dalla letteratura per testare il funzionamento dell'ambiente di simulazione [36]. Le applicazioni utilizzano primitive *MPI* implementate dal *middleware*.
Sono state raccolte statistiche sull'utilizzazione di rete e di bus e sulla latenza media dei pacchetti dovuta a contenzione di risorse [1].
Utilizzando il supporto per la raccolta di dati all'interno del kernel e delle applicazioni fornita dal simulatore *ML-RSIM*, sono stati valutati i costi della comunicazione legati ai vari strati di software coinvolti, dal livello applicativo, al *middleware*, al sistema operativo.

6.2 Applicazioni parallele

Utilizzando la libreria MPI implementata dallo strato *middleware*, si sono compilate due applicazioni parallele tratte da una collezione di *benchmark* MPI [36]: prodotto di matrici distribuito tra N processori e calcolo di π basato sull'algoritmo *dashboard*. Entrambi gli algoritmi prevedono un processore *master* che distribuisce il lavoro tra i vari processori *slave* e raccoglie i risultati a computazione terminata.

[1] In questo modello di rete, la contenzione si può verificare solo quando due nodi intendono comunicare con lo stesso nodo destinatario

Una terza applicazione prevede il passaggio di un messaggio in un circolo di processori per valutare il comportamento del sistema in presenza di messaggi lunghi (2 kB).
La quarta applicazione consiste in una catena di processori, ognuno dei quali esegue un prodotto tra la matrice in ingresso e una matrice generata internamente, passandola al processore successivo. Questa applicazione è stata eseguita anche in un sistema multiprocessore su chip basato su bus condiviso e ha consentito di confrontare le due architetture.

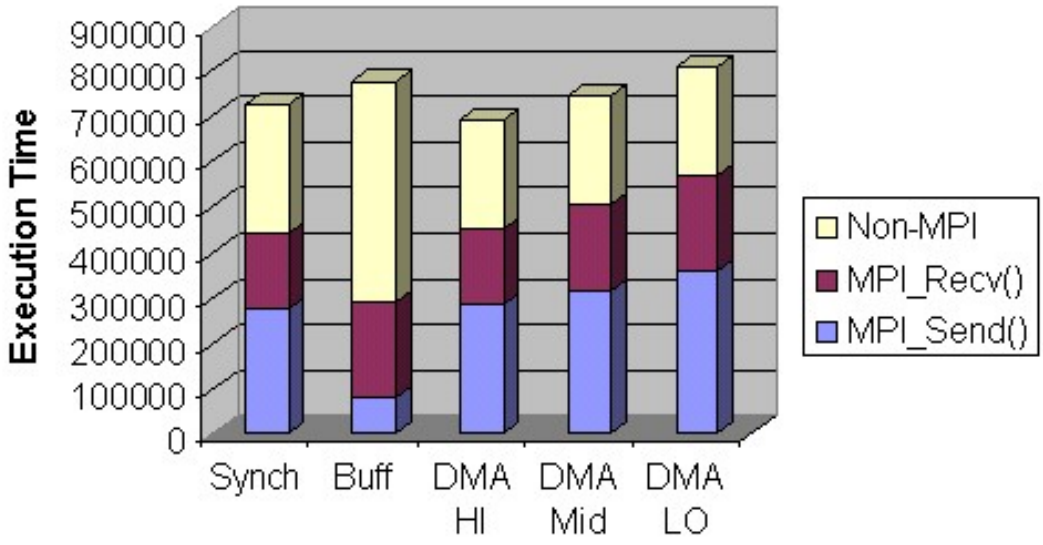


Figura 6.1: Matrix Multiply Performance

In figura 6.1 è rappresentata la prestazione complessiva in una configurazione a quattro processori, suddivisa in cicli macchina impiegati da chiamate `MPI_Send()`, `MPI_Recv()` e tempo impiegato per la computazione. Le cinque statistiche corrispondono a: `MPI_Send()` sincrone, `MPI_Send()` bufferizzata, `MPI_Send()` sincrone con trasferimenti DMA, utilizzando tre diverse politiche di *scheduling* per il processo di *middleware* (alta, media e bassa priorità).
Utilizzando la `MPI_Send()` bufferizzata – che consente al processo sorgente di proseguire nell'esecuzione senza fermarsi ad aspettare il messaggio `RECV_ACK` che segnala la completata ricezione da parte del processo destinatario – si osserva una riduzione sostanziale del tempo necessario

per la chiamata MPI di trasmissione; il motivo è da ricondursi esclusivamente ad una differente semantica di sincronizzazione. Il tempo risparmiato nella `MPI_Send()` viene in realtà sprecato in più numerosi cambi di contesto tra il middleware e il sistema operativo e cicli di attesa sulla `MPI_Recv()` bloccante.
Mentre il DMA riduce notevolmente il tempo di trasmissione dei singoli messaggi (Sezione 6.9), comporta ritardi aggiuntivi nella gestione del controllore DMA e dell'interfaccia di rete che ne limitano l'efficacia in termini assoluti.

6.3 Overhead software

Analizzando le statistiche emerge chiaramente che la trasmissione fisica sulla rete di un messaggio rappresenta solo una minima parte del processo di comunicazione tra due processi in esecuzione su due processori diversi, stimabile intorno al 5-10%.
Una chiamata `MPI_Send()` invocata dalla applicazione comporta in sequenza:

1. una copia in un buffer del *middleware*
2. la gestione del protocollo di *handshake*
3. l'attesa di un messaggio `RECV_READY` corrispondente
4. un passaggio di contesto al sistema operativo attraverso una chiamata di sistema e una *trap* software
5. il trasferimento del messaggio alla interfaccia di rete con `LOAD` e `STORE` da parte della CPU oppure con transazioni DMA
6. trasmissione del pacchetto sulla rete di interconnessione

Dualmente, in corrispondenza di una chiamata `MPI_Recv()`, il messaggio deve percorrere a ritroso la stessa sequenza di operazioni.
È stato valutato come si distribuisce tra i vari livelli software il tempo

per trasferire un messaggio da un nodo all'altro, concentrandosi sul trasferimento dallo spazio di indirizzamento virtuale del processo utente ai registri del dispositivo di interfaccia di rete (*network interface*).

6.4 Chiamate di sistema e cambi di contesto

La principale causa di ritardi nel software è riconducibile ai passaggi di contesto dallo spazio utente allo spazio privilegiato in cui esegue il sistema operativo. Ciò avviene in conseguenza di una chiamata di sistema, implementata come una *trap* software nei sistemi operativi UNIX.
La necessità di passare il controllo al *kernel* è dovuta all'attuale struttura del simulatore: l'interfaccia di rete è mappata nello spazio di I/O *non-cacheable* del processore di nodo. Nell'architettura SPARC simulata da ML-RSIM, lo spazio di I/O è accessibile solamente da istruzioni che eseguono in modo privilegiato.
Quindi ogni volta che si intende comunicare con l'interfaccia di rete – non solo per trasmettere e ricevere messaggi, ma anche per verificare lo stato dei buffer, la presenza di messaggi in arrivo dalla rete e lo stato di avanzamento della trasmissione di messaggi – è necessario effettuare una costosa chiamata di sistema.
Il *middleware* è eseguito in spazio utente per consentire una maggiore integrazione con i processi utente locali. D'altra parte questo causa un aumento nel numero di *context switch* rispetto ad uno scenario in cui il *middleware* esegue come processo di sistema all'interno del *kernel*.
Il supporto del *kernel* per i soli processi "pesanti" in stile UNIX rende più costosa l'operazione. L'utilizzo di processi "leggeri" o *thread* ridurrebbe considerevolmente questi ritardi.
In termini quantitativi, dalla chiamata di sistema in spazio utente alla prima istruzione del *system call handler* eseguita in modo privilegiato, trascorrono tra i 300 e i 600 cicli macchina. Un ritardo così sensibile è da imputare al costoso salvataggio del contesto di un processo UNIX e ad una serie di *cache miss* dovute alla rottura del principio di località in corrispondenza del cambio di contesto; infatti il nuovo codice in esecuzione si troverà

ad utilizzare dati non recentemente utilizzati e in generale non correlati a quelli precedenti, aumentando considerevolmente la probabilità che questi non siano stati inseriti nella memoria tampone.
Se considerati nell'ambito di una infrastruttura di rete su chip, caratterizzata da latenza molto bassa – nell'ordine di decine di cicli – e alto *throughput*, questi ritardi assumono un significato importante. Se sono trascurabili in architetture parallele con latenze dell'ordine di grandezza dei microsecondi o più [45, 4, 5], diventano decisivi in un contesto di *network-on-chip*.

6.5 Polling

L'interfaccia di rete è gestita in *polling* dal *middleware*, che periodicamente interroga i registri di stato del dispositivo per verificare la presenza di nuovi pacchetti in arrivo dalla rete. I messaggi sono suddivisi in pacchetti, che a loro volta sono inviati sulla rete come *flit* successivi di dimensione pari alla larghezza del canale di interconnessione tra i nodi.
In figura 6.2 sono riportate statistiche relative ai cicli macchina consumati in attesa di nuovi pacchetti durante la ricezione dei messaggi; una volta che è iniziata la trasmissione del messaggio, è possibile che i buffer dell'interfaccia di rete si svuotino, costringendo il software a cicli di attesa attiva. Dai dati raccolti emerge che questa evenienza è estremamente rara, intorno a 2000 cicli durante l'intera esecuzione, su alcune centinaia di migliaia di cicli spesi in attività di comunicazione; il motivo risiede nell'alta velocità della rete di interconnessione rispetto al tempo necessario al software di sistema per svuotare i suoi buffer di ingresso.
Affidando questo trasferimento ad un modulo DMA, l'eventualità di cicli di attesa in generale aumenta; in questo caso infatti la velocità con cui vengono svuotati i buffer della *network interface* è paragonabile o superiore al *transfer rate* in ingresso dalla rete di interconnessione su chip. Aumenta quindi la probabilità che il trasferimento DMA verso la memoria del processore debba attendere nuovi pacchetti in arrivo dalla rete.
È anche interessante notare come variando la priorità del processo di *midd-*

leware nella politica di *scheduling* del sistema operativo si ottengono risultati notevolmente diversi. Sembra quindi di estrema importanza una operazione di *fine tuning* del software di sistema per massimizzare le prestazioni dell'applicazione *embedded* che si intende implementare.

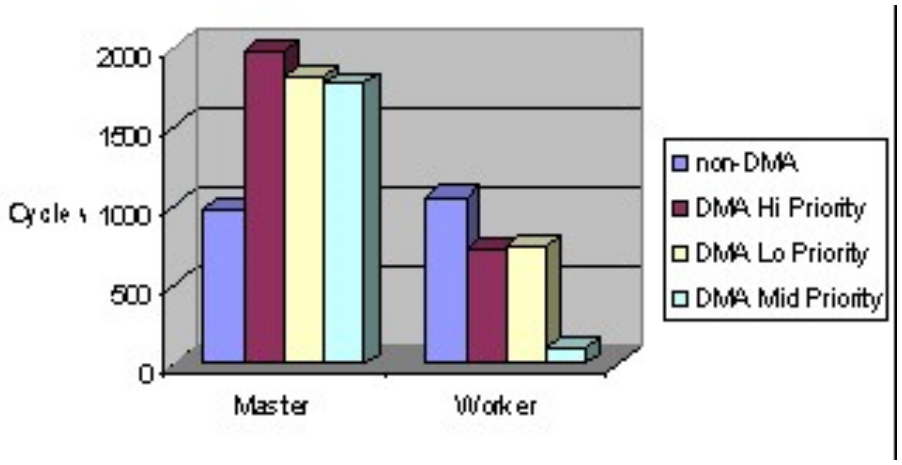


Figura 6.2: Polling for non-empty network interface

Dualmente durante la trasmissione di messaggi, il software di sistema (il *device driver* integrato nel sistema operativo) suddivide i messaggi in pacchetti di dimensione prefissata, per poi trasferirli all'interfaccia di rete. Quando il software di sistema richiede accesso all'interfaccia di rete per inviare un messaggio, deve verificare per ogni nuovo pacchetto la disponibilità del dispositivo a trasmettere sulla rete di interconnessione.
I buffer hardware della *network interface* si possono saturare in caso di congestione della rete dovuta a contenzione di risorse. Nel nostro modello si verifica contenzione nel caso in cui due nodi vogliano comunicare con lo stesso nodo destinazione; il secondo nodo sorgente dovrà allora attendere che il primo nodo abbia completato con successo la trasmissione dell'intero messaggio.
Una situazione diversa si verifica in modalità DMA: la suddivisione del messaggio in pacchetti non è effettuata in software, ma avviene un trasferimento *burst* nei registri della *network interface* dell'intero messaggio in presenza di spazio sufficiente. In modalità DMA si osserva una notevole diminuzione dell'*overhead* dovuto al *polling*. Equilibrare velocità

della rete di interconnessione e trasferimento dati dalla memoria utente all'interfaccia di rete consente una riduzione dei tempi di attesa.

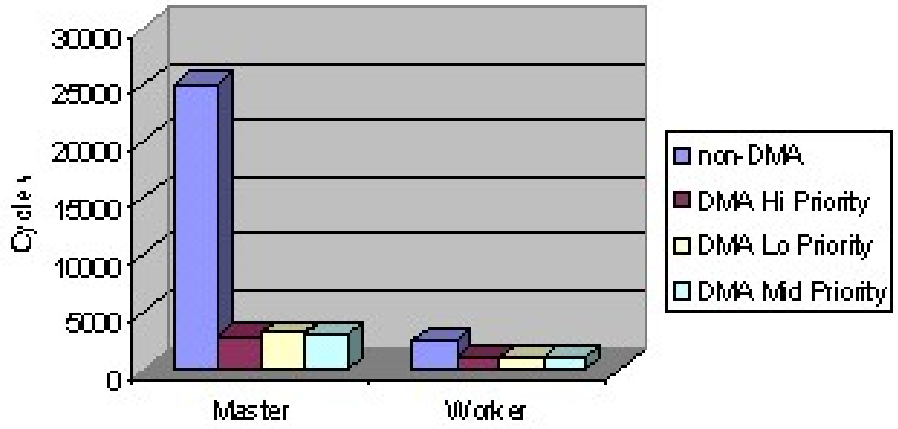


Figura 6.3: Polling for ready network interface

6.6 Sistema operativo e Middleware

Si è analizzata l'interazione tra *middleware*, sistema operativo e *network interface* nel processo di trasmissione e ricezione di messaggi. Nell'architettura software implementata, la libreria MPI a livello applicativo si occupa semplicemente di trasferire i contenuti dei messaggi utente da e verso i buffer gestiti dal *middleware* e di gestire le semantiche di sincronizzazione delle funzioni di comunicazione.
L'effettiva comunicazione dei messaggi verso nodi sulla rete viene effettuata dal *middleware*, che invoca le chiamate di sistema `NI_Send()` e `NI_Receive()` per trasferire dati dai buffer di sistema ai registri interni dell'interfaccia di rete.
I servizi `NI_Send()` e `NI_Receive()` fanno parte del *device driver* e sono inclusi nel *kernel* del sistema operativo; trasferire il controllo al sistema operativo causa un cambio di contesto, rappresentato in figura 6.4 dalle voci *NI_Send() overhead* e *NI_Receive() overhead*. Si osserva l'alta incidenza di *NI_Receive() overhead* a causa delle chiamate in *polling* per verificare la

presenza di pacchetti dalla rete di interconnessione.

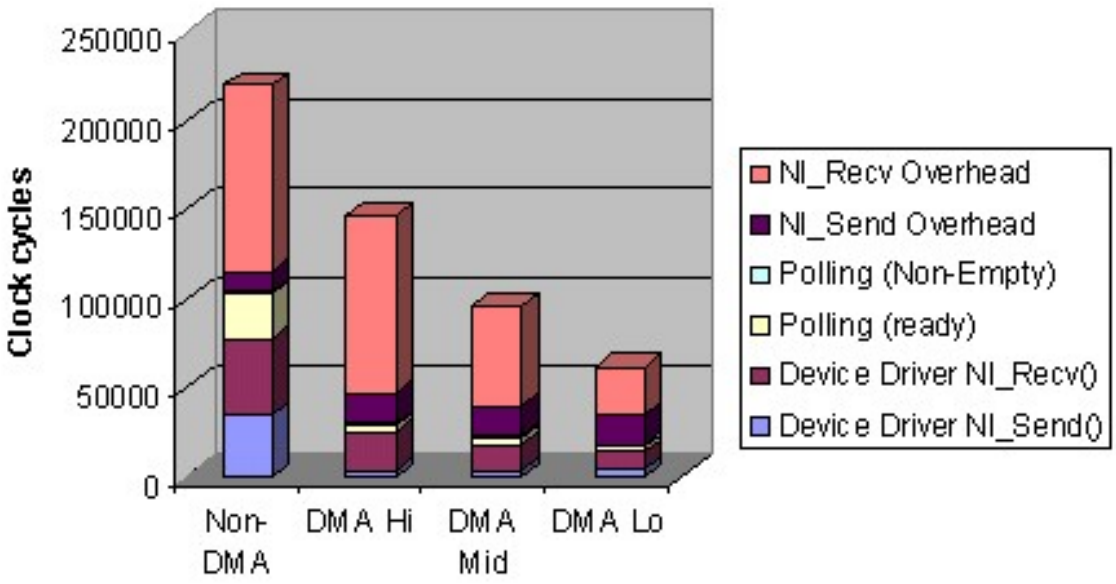


Figura 6.4: Middleware and Operating System

Introducendo la modalità DMA, si riduce il peso di una gestione a *polling* dell'interfaccia di rete, che si evidenzia nella diminuzione dei cicli di *NI_Receive() overhead*. Evidentemente al diminuire della priorità del *middleware*, si riducono anche le chiamate "a vuoto" sulla *network interface*. Inoltre i trasferimenti DMA alleggeriscono significativamente la funzione `NI_Send()` del *device driver*, eliminando le sequenze di `LOAD` e `STORE` software eseguite dalla CPU in modo privilegiato.

6.7 Utilizzazione della rete di interconnessione

Nei diversi *benchmark* utilizzati, l'utilizzazione di rete si è mantenuta a livelli bassi, tra il 5% e il 10%, nonostante le applicazioni fossero relativamente *communication-intensive*. Una bassa utilizzazione di rete ci ha inoltre consentito di adottare un modello semplificato della rete, che non simula contenzione a livello di risorse quali buffer e porte degli *switch*.

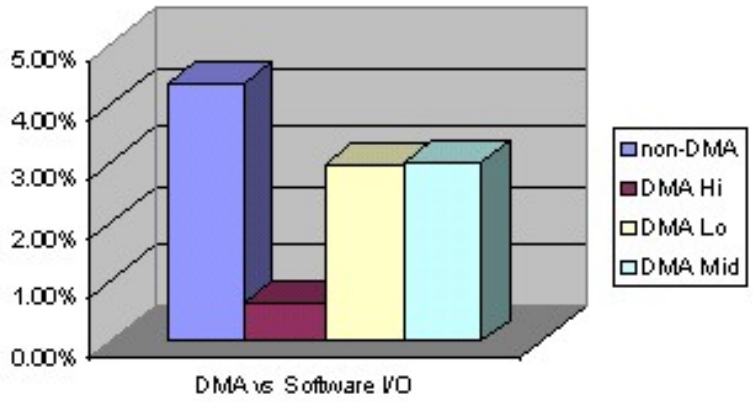


Figura 6.5: Network Utilization

L'utilizzo della modalità di trasferimento DMA ha mostrato di avere un impatto sull'utilizzazione di rete in quanto rende più fluido e veloce lo svuotamento dei buffer della *network interface* contentente pacchetti in arrivo dalla rete. Maggiore spazio disponibile nei buffer si traduce in minore congestione di rete dovuta a *overflow* presso il nodo destinatario.
Anche la priorità del processo *middleware* nelle politiche di *scheduling* del sistema operativo – contrassegnato da *DMA Hi, Low, Mid* in figura 6.5 – presenta una correlazione con la congestione di rete. Il motivo è ancora legato alla frequenza con cui vengono aggiornati lo stato e i buffer dell'interfaccia di rete, in questo caso da parte del software: una minore priorità – e quindi una minore frequenza di aggiornamento – aumenta la probabilità di congestione sulla rete.

6.8 Utilizzazione di Cache e Bus

L'incidenza di *cache miss* dati e istruzioni è rimasta pressoché costante variando diversi parametri di simulazione.
Cache hit rate per le istruzioni si è mantenuta a livelli molto alti superiori al 98% in ogni contesto. Per quanto riguarda la *cache* dati, il tasso di *hit* si è collocato tra l'80% e l'85%; la semantica bufferizzata delle operazioni di `MPI_Send()` ha causato un aumento dei casi di *cache miss* di un paio di punti percentuali nei diversi esperimenti.

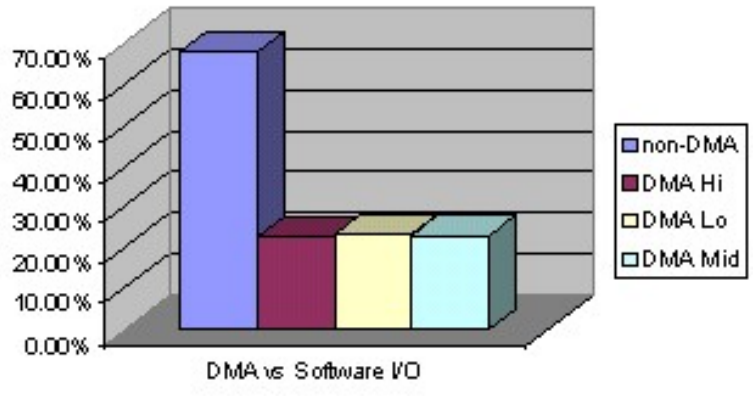


Figura 6.6: Bus Utilization

Analogamente il tasso di utilizzazione del bus all'interno dei diversi nodi non ha presentato dipendenza dalle diverse applicazioni parallele eseguite. In modalità DMA, l'utilizzazione del bus da parte della CPU, che evidentemente si concentrava nel trasferimento dei messaggi dalla memoria alla *network interface*, subisce un calo dall'70% al 20%. La porzione mancante in questo caso è gestita direttamente dall'interfaccia di rete e dal modulo DMA.

6.9 Direct Memory Access

Il software di sistema ha dimostrato di non poter tenere il passo con il flusso di messaggi dalla rete di interconnessione, specialmente per dimensione di messaggi più alta, causando anche congestione della rete in alcuni casi. Aumentare anche considerevolmente la dimensione dei buffer di ingresso del modulo di interfaccia di rete non è stato sufficiente ad eliminare questo problema.
È necessario un approccio diverso per sfruttare al massimo il *throughput* offerto dalla rete. I dati dei messaggi non sono copiati nei registri della *network interface* da trasferimenti I/O controllati dal *device driver* del sistema operativo; la *network interface* è in grado di spostare e recuperare dati direttamente senza l'intervento da parte della CPU. Questo meccanismo è noto come *direct memory access* (DMA).

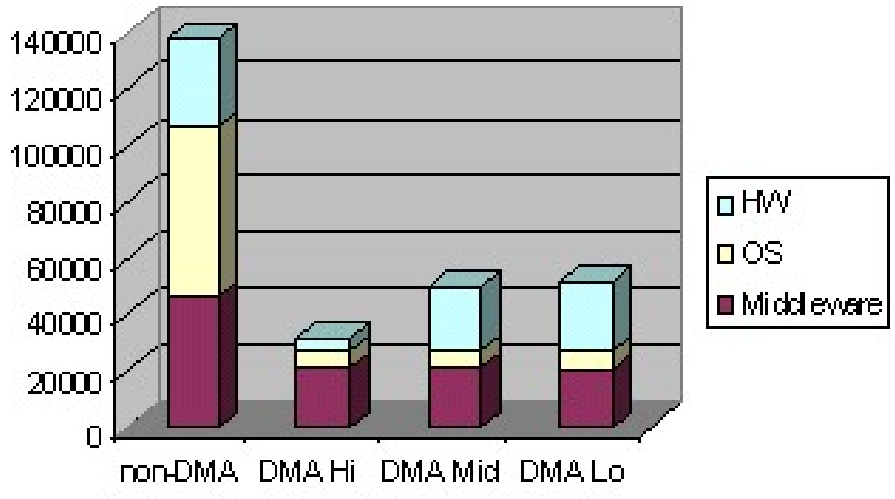


Figura 6.7: DMA con messaggi corti

In figura 6.7 è rappresentato il miglioramento ottenuto con trasferimenti DMA, al variare della priorità del *middleware*. Il vantaggio sostanziale appare al livello del sistema operativo in quanto il *device driver* non si occupa più di spostare i dati dalla memoria all'interfaccia di rete ma solamente di programmare il dispositivo con l'indirizzo in memoria dei dati e il numero di byte da trasferire (durante le operazioni `NI_Send()` e `NI_Receive()` citate in precedenza). Come atteso, percentualmente il risparmio è superiore in presenza di messaggi lunghi: in figura 6.8 si mostrano i dati relativi al passaggio di un *token* della lunghezza di 2 kilobyte in un circolo di processori.

Si può notare inoltre come l'utilizzo del DMA vada ad influire positivamente sulla velocità di trasferimento sulla rete di interconnessione (contrassegnata in figura 6.7 e 6.8 da *HW*). Il DMA consente infatti di mantenere i buffer della *network interface* liberi grazie a trasferimenti più veloci verso la memoria, riducendo la probabilità di congestione e contenzione di risorse. Il meccanismo di controllo di flusso si trova quindi più raramente nella condizione di sospendere i trasferimenti in attesa che i buffer alla destinazione dispongano di spazio sufficiente a memorizzare nuovi pacchetti.

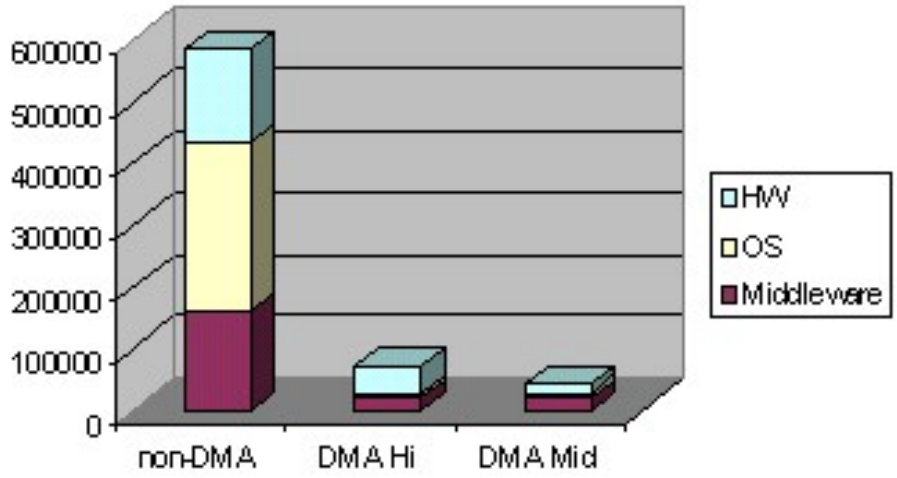


Figura 6.8: DMA con messaggi lunghi

6.10 Algoritmi paralleli

I *benchmark* utilizzati non si sono dimostrati sufficientemente impegnativi dal punto di vista computazionale e non hanno consentito di verificare appieno una sovrapposizione di comunicazione e computazione. Tuttavia in modalità DMA, si è osservata la possibilità di eseguire alcune operazioni di controllo da parte del *middleware* in parallelo al trasferimento dati eseguito indipendentemente in hardware, utilizzando primitive di *probe* fornite dal *driver* dell'interfaccia di rete come punti di sincronizzazione.
Gli algoritmi paralleli utilizzati adottano una tecnica di suddivisione del lavoro da parte di un processore *master*, che successivamente raccoglie i risultati dagli altri processori (detti *workers*). Questo metodo tende a serializzare l'esecuzione delle operazioni e causa un carico di comunicazione localizzato in entrata ed uscita dal nodo *master*. L'architettura di comunicazione fornita dalla rete di interconnessione non sembra essere sfruttata al massimo nella sua caratteristica di poter gestire numerosi flussi di comunicazione contemporaneamente.

Capitolo 7

Conclusioni

7.1 Introduzione

I dati raccolti hanno consentito di valutare la ripartizione del carico di comunicazione tra i vari livelli hardware e software dell'architettura (Sezione 7.2). Si è valutata l'influenza di parametri software quali la politica di *scheduling* del sistema operativo sulle prestazioni dell'intero sistema (Sezione 7.3) e l'importanza di un supporto hardware alla comunicazione (Sezione 7.4). Si sono inoltre delineate alcune strategie per migliorare e ottimizzare l'ambiente di simulazione in futuro (Sezione 7.5).

7.2 Comunicazione Hardware e Software

Eseguendo le applicazioni parallele sul simulatore è stato possibile stimare come si ripartisce il carico della trasmissione di messaggi da un processore all'altro tra i livelli hardware e software della architettura *network-on-chip* presentata. Il percorso di un messaggio è stato seguito *end-to-end*, dallo spazio di indirizzamento di un processo utente su una CPU fino alla memorizzazione in un buffer del processo utente destinatario, in esecuzione su un processore diverso.
I livelli di astrazione che un messaggio attraversa nel suo percorso sono i seguenti:

- Trasmissione dei pacchetti sulla rete, gestita in hardware dai dispositivi di *network interface* al nodo sorgente e al nodo destinatario.

- Sistema operativo. Il *device driver* si occupa del trasferimento dati dai buffer di comunicazione all'interfaccia di rete in due modalità, con e senza *direct memory access* (DMA). In modalità DMA, il sistema operativo programma semplicemente l'interfaccia di rete con l'indirizzo fisico del buffer contenente il messaggio e con la sua lunghezza; una opportuna chiamata di *probe* verifica l'avanzamento del trasferimento. Senza DMA, il sistema operativo trasferisce il messaggio dai buffer software ai registri dell'interfaccia di rete con un ciclo di istruzioni `LOAD` e `STORE` suddividendolo in diversi pacchetti.

- Middleware. Questo strato di software si colloca tra il sistema operativo e le applicazioni utente; implementa la libreria *message-passing*, la gestione dei buffer e i protocolli di *handshake*

- Livello applicativo. Un processo utente che intenda partecipare come *endpoint* di comunicazione, utilizza le chiamate di interfaccia MPI; una libreria collegata al processo utente agisce da *frontend* per i servizi di comunicazione forniti dal *middleware* con la gestione dei trasferimenti tra lo spazio di indirizzamento utente e i buffer di comunicazione e il controllo della semantica delle funzioni di `send` e `receive` (sincrona, bloccante, non-bloccante).

Sono stati variati diversi parametri nell'architettura simulata di sistema multiprocessore su chip: la latenza di rete, la larghezza del canale sulla rete di interconnessione, la larghezza di bus locale ad ogni nodo, la lunghezza della *pipeline* di comunicazione, la dimensione dei buffer hardware e software, le semantiche a livello applicativo.
Per le diverse applicazioni parallele eseguite, il peso dei tre strati di software – applicativo, *middleware*, sistema operativo – si è attestato tra il 90% e il 95% dei cicli macchina.

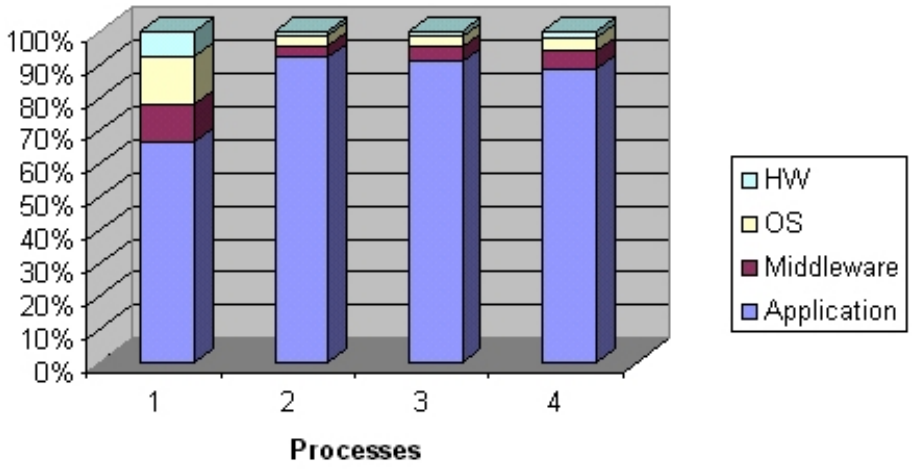


Figura 7.1: Ripartizione della comunicazione tra i diversi strati

7.3 Politiche di *scheduling*

Lo strato software di *middleware* comprende un processo che gestisce in *polling* l'interfaccia di rete, aggiorna i buffer di comunicazione ed associa i messaggi in entrata e in uscita sulla rete con i processi locali interagendo attraverso l'interfaccia MPI.
Il sistema multiprocessore è stato simulato attribuendo al processo di *middleware* tre diversi livelli di priorità, alta, media e bassa; un processo ad alta priorità giova di uno *slot* di tempo più ampio per la propria esecuzione secondo la gestione dello *scheduler* del sistema operativo.
È emerso che un *middleware* a bassa priorità, cioè particolarmente attento a cedere volontariamente il controllo del processore ad altri *task*, riduce nel complesso il numero di cambi di contesto e migliora in generale la gestione a polling dell'interfaccia di rete perché il numero di accessi al dispositivo in assenza di nuovi pacchetti dalla rete diminuisce.

Tuttavia la prestazione nella trasmissione e ricezione di pacchetti sulla rete ha dimostrato una forte dipendenza dalla particolare applicazione. In figura 7.2 sono rappresentati i cicli di clock spesi in trasmissione nell'algoritmo di moltiplicazione di matrici distribuito, caratterizzato da uno scambio frequente di messaggi brevi; in questo caso una priorità più alta garantisce una prestazione migliore. La figura 7.3 si riferisce invece al pas-

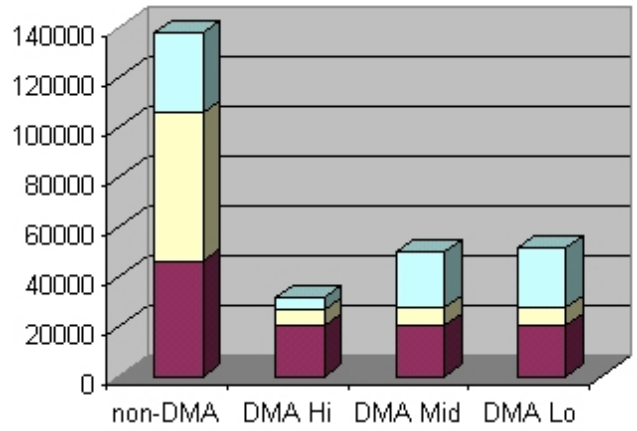


Figura 7.2: Politica di *scheduling* con messaggi corti

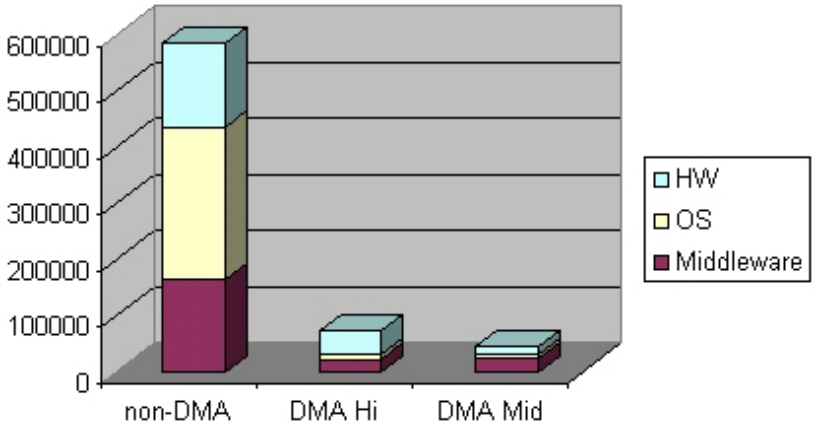


Figura 7.3: Politica di *scheduling* con messaggi lunghi

saggio di dieci messaggi di 2 kilobyte in un circolo di processori. Si osserva come una priorità ridotta consenta di migliorare i tempi di trasmissione.
È interessante notare come una diversa politica di *scheduling* vada ad influenzare l'utilizzazione di rete a livello hardware (contrassegnato in figura da *HW*), a causa di una diversa gestione dei buffer di comunicazione nel *middleware* e nel dispositivo di interfaccia, che a sua volta si ripercuote sul controllo di flusso sulla rete.
Se non si restringe l'analisi alla comunicazione vera e propria, la prestazione complessiva dei vari *benchmark* si è dimostrata superiore mantenendo una priorità più alta per il processo di *middleware*. Probabilmente appli-

cazioni caratterizzate da un maggiore carico computazionale trarrebbero vantaggio da un *middleware* che concedesse più tempo alla CPU per eseguire compiti non legati alla comunicazione con altri processori, riducendo la propria priorità.
Una fase di *tuning* del software di sistema per migliorare le prestazioni di una particolare applicazione *embedded* dovrebbe quindi considerare come il *middleware* di comunicazione viene posto in esecuzione dallo *scheduler* del sistema operativo.

7.4 Supporto hardware

Gestire le primitive di comunicazione tra nodi completamente in software non consente di ottimizzare il funzionamento dell'infrastruttura hardware; la ragione risiede nel fatto che senza un supporto hardware la veloce rete di interconnessione tende a saturare i buffer presenti nei dispositivi di *network interface* e creare condizioni di congestione. In presenza di rete *wormhole routing*, che prevede la trasmissione in modalità *pipelined*, i pacchetti che non possono essere ricevuti completamente dal nodo destinazione a causa di *buffer overflow* tengono impegnate risorse lungo il percorso, impedendo ad altri pacchetti di transitare sulla rete e peggiorando ulteriormente la contenzione.
Assumendo una larghezza di canale di 64 bit per la microrete di interconnessione, nel solo tempo necessario al sistema operativo per attivare il processo di *middleware* che gestisce la comunicazione (Sezione 6.4), possono essere consegnati all'interfaccia di rete della destinazione 3-4 kilobyte di dati. Scegliere una soluzione software per la consegna dei messaggi ai processi utente costringe il progettista a riservare buffer hardware di dimensione considerevole in input ad ogni nodo dell'architettura parallela. L'implementazione di un dispositivo *network interface* con supporto DMA ha mostrato un miglioramento nell'utilizzo dei buffer e una conseguente ridotta contenzione sulla rete.
Un ulteriore miglioramento è prevedibile associando al messaggio un indirizzo di memoria in spazio utente presso il nodo destinazione, con un

approccio simile alla soluzione *active messages* [16], come delineato in sezione 7.5.3.

7.5 Strategie di ottimizzazione

Questa sezione presenta alcune strategie per ottimizzare in futuro il sistema multiprocessore presentato, sia per quanto riguarda la piattaforma hardware che gli strati software.
Nelle sezioni 7.5.1 e 7.5.2 sono presentate due soluzioni alternative per limitare l'*overhead* legato ai passaggi di contesto tra i processi e tra codice utente e privilegiato.
Nelle sezioni 7.5.3 e 7.5.4 si propone una estensione del supporto hardware alla comunicazione per eliminare la necessità di copie di messaggi da memoria a memoria ed alleggerire la struttura del middleware di comunicazione allo scopo di ridurre il peso dominante del software nella latenza di trasmissione dei messaggi.
Infine la sezione 7.5.5 suggerisce alcune linee guida a livello applicativo per lo sviluppo di algoritmi paralleli che sfruttino le caratteristiche distintive delle architetture *network-on-chip*.

7.5.1 Integrazione di sistema operativo e *middleware*

La struttura attuale del middleware prevede un processo UNIX in *background* che comunica con l'interfaccia di rete attraverso il meccanismo delle chiamate di sistema. Il processo di *middleware* esegue in spazio utente nel tentativo di ottimizzare l'interazione con i processi che utilizzano funzioni di comunicazione MPI. D'altra parte questa scelta pone l'interfaccia tra modo utente a modo privilegiato tra il middleware e il sistema operativo, la cui interazione si è dimostrata molto intensa (Sezione os-middleware).
L'integrazione del middleware come processo di sistema all'interno del *kernel* ridurrebbe l'*overhead* dovuto alle istruzioni *trap* necessarie al passaggio da codice utente a codice privilegiato secondo la convenzione UNIX

delle *system call*. In questa configurazione sarebbero allora le chiamate MPI ad essere implementate come chiamate di sistema.

7.5.2 Processi leggeri

Il sistema operativo utilizzato nel simulatore *ML-RSIM* è basato su *NetBSD* e fornisce il supporto per processi "pesanti" UNIX [41, 26]. Il salvataggio del contesto di un processo UNIX è un'operazione molto dispendiosa, che impiega la CPU per alcune centinaia di cicli.
Questo aspetto ha un peso nella struttura software presentata perché il *middleware* di comunicazione comprende un processo di gestione dei buffer di ingresso e uscita che esegue in *background*. Almeno un passaggio da un processo utente al processo di *middleware* è necessario per ogni chiamata MPI a livello applicativo; centinaia di cicli di clock assumono un particolare significato in una architettura multiprocessore su chip, caratterizzata da latenze nell'ordine delle poche decine di cicli e alto *throughput*.
Una possibile soluzione a questo problema è offerta dai processi "leggeri" o *thread*. In questo caso diversi *thread* condividono lo stesso spazio di indirizzamento e in generale un cambio di contesto comporta il salvataggio di un numero molto inferiore di informazioni e strutture dati.
È possibile ipotizzare di sviluppare uno strato software di comunicazione in spazio utente come processo leggero, che interagisca con i processi MPI appartenenti allo stesso nodo – cioè in esecuzione sullo stesso processore – condividendo lo spazio di indirizzamento e riducendo l'*overhead* di passaggio da un flusso di esecuzione all'altro. Una simile strategia è comune per sistemi operativi *embedded* e *real-time* quali RTEMS [7].
Tuttavia con questo approccio, rimarrebbe l'*overhead* sostanziale del passaggio alla modalità di esecuzione privilegiata attraverso chiamate di sistema al driver di dispositivo per comunicare con l'interfaccia di rete (Sezione 6.6). Mappare l'interfaccia di rete nello spazio di indirizzamento dei processi utente – in sostituzione alla scelta tradizionale di mappare i dispositivi in uno spazio di indirizzamento I/O privilegiato – sarebbe allora decisivo per trarre massimo vantaggio dalla soluzione *multithreaded*;

sono chiari in questo caso i rischi dal punto di vista di protezione e sicurezza, da affrontare attraverso la definizione di una interfaccia di accesso "trusted", implementata da una libreria di middleware. È inoltre da ricordare il contesto di applicazioni *embedded* nel quale si collocano i sistemi multiprocessore su chip, in cui le applicazioni utente sono definite dal progettista ed eventuali applicazioni caricate dinamicamente possono essere poste in esecuzione in un modello a *sandbox*.

7.5.3 Copie da memoria a memoria

Nella struttura attuale del software è necessaria una sola copia da memoria a memoria per ogni messaggio in uscita o entrata, dallo spazio utente ai buffer di comunicazione allocati dal *middleware* e viceversa. Una copia da memoria a memoria rappresenta in generale un *overhead* del software di comunicazione ed aumenta la latenza totale nella trasmissione del messaggio.
In trasmissione (`MPI_Send()`), la libreria copia il messaggio dall'indirizzo specificato dal processo utente a buffer di sistema; questo trasferimento non è necessariamente un aspetto negativo, in quanto può essere utile allo sviluppatore per realizzare la semantica bufferizzata e, ad esempio, riutilizzare immediatamente la locazione di memoria in spazio utente. Se questo non è il caso, può essere utile sfruttare il supporto hardware alle comunicazioni internodo, programmando un trasferimento DMA direttamente dallo spazio di indirizzamento del processo utente al dispositivo di *network interface*; lo standard MPI consente di definire una `send` non bloccante per avviare il trasferimento e una funzione `probe` per verificarne l'avanzamento.
In ricezione (`MPI_Recv()`), ogni messaggio in arrivo dalla rete viene memorizzato in unico *input buffer* di sistema; gli *header* dei messaggi vengono confrontati con le richieste pendenti tra i processi locali e associati al *communication endpoint* corrispondente. Saranno poi trasferiti allo spazio utente dalla libreria MPI; per eliminare quest'ultima copia da memoria a memoria, sarebbe necessario saltare la fase di *matching* tra le intestazioni a

livello del *middleware* e consentire all'interfaccia di rete di conoscere l'indirizzo fisico di destinazione del messaggio nel buffer utente.
L'indirizzo dovrebbe essere incluso nel messaggio inviato sulla rete da parte del mittente. Se l'ambiente adotta l'ipotesi semplificativa di avere lo stesso programma in esecuzione su tutti i nodi (*Single Program Multiple Data* o SPMD), il mittente è in grado di conoscere l'indirizzo fisico sul destinatario, altrimenti è necessario apportare una modifica al protocollo di trasporto come descritto in sezione 7.5.4.

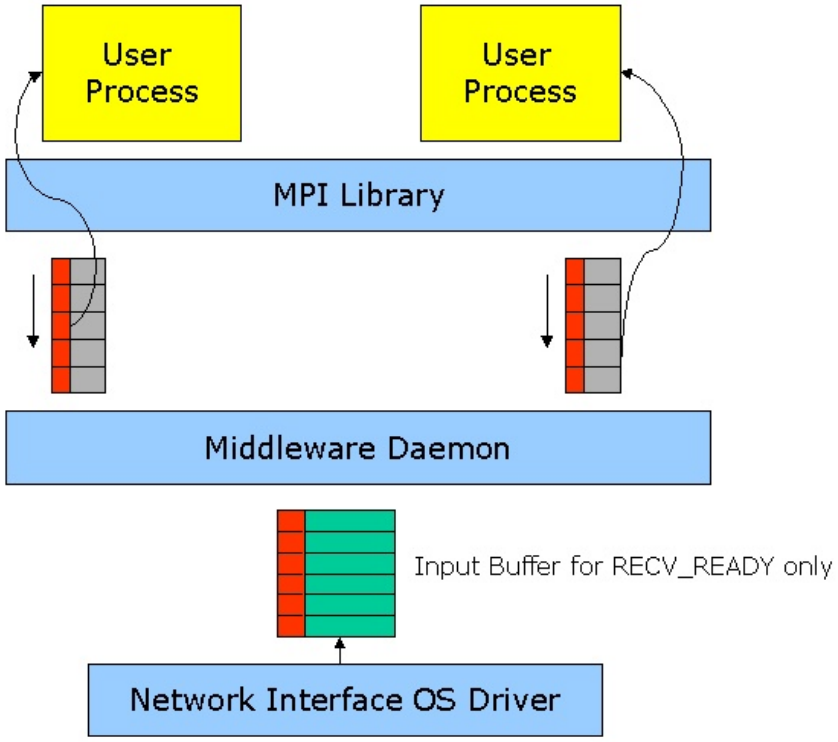


Figura 7.4: Middleware leggero

Eliminata l'esigenza del confronto tra le intestazioni dei messaggi in ingresso dalla rete e le richieste in sospeso, il processo di *middleware* può essere notevolmente alleggerito e le risorse di buffering ridotte sostanzialmente. Un buffer di ingresso è mantenuto per memorizzare i `ready-to-receive` e avviare le transazioni corrispondenti, secondo il protocollo "iniziato dal ricevente" in uso nel sistema progettato (Sezione B.11.2).

7.5.4 Protocollo di trasporto

Allo scopo di eliminare la copia da memoria a memoria necessaria per trasferire i messaggi dall'input buffer di sistema allo spazio utente, è possibile sfruttare il supporto hardware offerto dall'interfaccia di rete, a patto che il messaggio in trasmissione sulla rete sia provvisto di indirizzo fisico di destinazione; a tale scopo proponiamo una integrazione al livello di trasporto del protocollo di comunicazione (Figura 7.5).
La sincronizzazione nella comunicazione sulla microrete di interconnessione è realizzata con un protocollo di *handshake receiver-initiated*. La *network interface* sul nodo ricevente spedisce il `ready-to-receive` insieme all'indirizzo e alla lunghezza del buffer nel quale si intende memorizzare il messaggio da ricevere. La *network interface* sul nodo sorgente allega al messaggio gli stessi dati forniti dalla destinazione, che potrà trasferire direttamente il messaggio in spazio utente senza l'intervento del software.

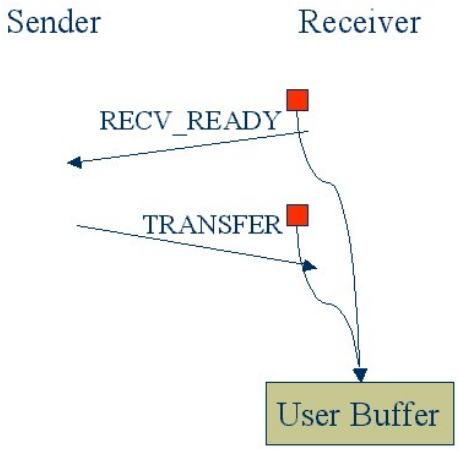


Figura 7.5: Protocollo di trasporto

7.5.5 Algoritmi paralleli

Allo scopo di utilizzare al meglio le intrinseche caratteristiche di parallelismo dell'architettura *Network-on-Chip* rispetto al bus condiviso, gli stessi algoritmi multiprocessore a livello applicativo giocano un ruolo fondamentale. La metodologia a *master - slave* causa in generale una serializza-

zione della comunicazione intorno al nodo master che distribuisce i dati e raccoglie i risultati, rendendo la comunicazione di fatto non dissimile a quella che si può ottenere con una piattaforma *shared bus*.
In presenza di decine o centinaia di processori su singolo chip nei prossimi anni, è possibile ipotizzare una suddivisione del lavoro gerarchica, ad albero o a *cluster* di processori. Così facendo, mappando opportunamente i nodi logici sui nodi fisici tenendo conto della loro distanza in *hop* o *switch*, processori appartenenti a diversi rami o gruppi possono scambiarsi informazioni in parallelo utilizzando link fisici differenti.
L'utilizzo di librerie message passing come MPI affida allo sviluppatore di applicazioni la responsabilità di ottimizzare gli algoritmi paralleli per la particolare topologia di rete di interconnessione. MPI supporta topologie virtuali e gruppi di processi di cui è opportuno valutare l'efficacia ed utilità anche nel contesto dei sistemi multiprocessore su chip.

Bibliografia

[1] W.J. Bainbridge and S.B. Furber. Delay Insensitive System-on-Chip Interconnect using 1-of-4 Data Encoding. 2001.

[2] Luca Benini and Giovanni De Micheli. Networks on Chips: A New SoC Paradigm. *IEEE Computer*, February 2002.

[3] M.T. Bohr. Nanotechnology goals and challenges in electronic applications. *IEEE Transactions on Nanotechnology*, 1(1):56–62, March 2002.

[4] H. Chen and P. Wyckoff. Performance evaluation of a Gigabit Ethernet switch and Myrinet using real application cores. *http://www.osc.edu/ pw/avici/hoti8.pdf*.

[5] H. Chen, P. Wyckoff, and K. Moor. Cost/Performance evaluation of Gigabit Ethernet and Myrinet as Cluster Interconnect. *http://www.osc.edu/ pw/avici/opnet2000.pdf*.

[6] OAR Corp. *RTEMS BSP and Device Driver Development Guide*. http://www.rtems.com/RTEMS/rtems.html.

[7] OAR Corp. *RTEMS On-Line Library*. http://www.rtems.com/RTEMS/rtems.html.

[8] OAR Corp. *RTEMS Porting Guide*. http://www.rtems.com/RTEMS/rtems.html.

[9] OAR Corp. *RTEMS SPARC Applications Supplement*. http://www.rtems.com/RTEMS/rtems.html.

[10] D.E. Culler, J.P. Singh, and A. Gupta. *Parallel Computer Architecture, A Hardware/Software Approach*. Morgan Kaufmann Publishers, Inc., 1999.

[11] W.J. Dally. Lecture 6, Interconnection Networks Architecture and Design. *Stanford University, Class material*.

[12] W.J. Dally and S. Lacy. VLSI Architecture: Past, Present, and Future. *20th Anniversary Conference on Advanced Research in VLSI. Proceedings.*, pages 232–241, 1999.

[13] W.J. Dally and C.L. Seitz. The Torus Routing Chip. *J. Distributed Computing*, 1(3):187–196, 1986.

[14] W.J. Dally and B. Towles. Route Packets, Not Wires: On-Chip Interconnection Networks. *Design Automation Conference. Proceedings.*, pages 684–689, 2001.

[15] J. Duato, S. Yalamanchili, and L. Ni. *Interconnection Networks, an Engineering Approach*. IEEE Computer Society Press, 1997.

[16] T.v. Eicken, D.E. Culler, S.C. Goldstein, and K.E. Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. *Computer Architecture*, pages 256 –266, 1992.

[17] D.R. Engler, M.F. Kaashoek, and J. O'Toole Jr. Exokernel: An Operating System Architecture for Application-Level Resource Management. *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles, Operating Systems Review*, 29(5):251–266, December 1995.

[18] M. J. Flynn, P. Hung, and K. W. Rudd. Deep-submicron microprocessor design issues. *IEEE*, 1999.

[19] MPI Forum. *MPI-2: Extensions to the Message-Passing Interface*. http://www.mpi-forum.org.

[20] MPI Forum. *MPI: A Message-Passing Interface Standard*. http://www.mpi-forum.org.

[21] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.

[22] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, 1999.

[23] William D. Gropp and Ewing Lusk. *User's Guide for* `mpich`, *a Portable Implementation of MPI*. Mathematics and Computer Science Division, Argonne National Laboratory, 1996. ANL-96/6.

[24] T. Grötker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Kluer Academic Publishers, 2002.

[25] P. Guerrier and A. Grenier. A Generic Architecture for On-Chip Packet-Switched Interconnections. *Proc. IEEE Design Automation and Test in Europe (DATE 2000), IEEE Press*, pages 250–256, 2000.

[26] http://www.netbsd.org. *NetBSD.org*. The NetBSD Project.

[27] C.J. Hughes, V.S. Pai, P. Ranganathan, and S.V. Adve. Rsim: Simulating Shared-Memory Multiprocessors with ILP Processors. *IEEE Computer*, February 2002.

[28] J. Robert Jump. *Netsim Reference Manual*. Rice University, 1993.

[29] J. Robert Jump. *Yacsim Reference Manual*. Rice University, 1993.

[30] A.B. Kahng. Design technology productivity in the DSM era. *Design Automation Conference. Proceedings.*, 2001.

[31] A.B. Kahng. Finding and sharing brick walls. *Computer-Aided Network Design Committee (CANDE)*, September 2001.

[32] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford FLASH Multiprocessor. *Proceedings of the 21st International Symposium on Computer Architecture, Chicago, IL*, pages 302–313, April 1994.

[33] S.S. Lumetta and D.E. Culler. Managing concurrent access for shared memory active messages. *Parallel Processing Symposium. IPPS/SPDP*, pages 272–278, 1998.

[34] M.Birnbaum and H.Sachs. How VSIA Answers the SOC Dilemma. *IEEE Computer*, June 1999.

[35] L.M. Ni and P.K. McKinley. A Survey of Wormhole Routing Techniques in Direct Networks. *IEEE Computer*, pages 62–76, February 1993.

[36] University of New Mexico. *High Performance Computing, Education and Research Center (HPCERC)*. http://www.hpcerc.unm.edu.

[37] Vijay S. Pai, Parthasarathy Ranganathan, and Sarita V. Adve. *RSIM Reference Manual*. http://rsim.cs.uiuc.edu/rsim, 1997.

[38] M. Rosenblum, E. Bugnion, S. Devine, and S. A. Herrod. Using the SimOS Machine Simulator to Study Complex Computer Systems. *ACM Transactions on Modeling and Computer Simulation*, 7(1):78–103, January 1997.

[39] M. Rosenblum, S.A. Herrod, E. Witchel, and A. Gupta. Complete computer system simulation: the SimOS approach. *IEEE Parallel & Distributed Technology: Systems & Applications*, 3(4):34–43, 1995.

[40] Lambert Schaelicke and Mike Parker. L-RSIM: A Simulation Environment for I/O Intensive Workloads. In *Proceedings of the 3rd Annual IEEE Workshop on Workload Characterization*, pages 83–89, 2000.

[41] Lambert Schaelicke and Mike Parker. *ML-RSIM Reference Manual, Tech report 02-10*. Department of Computer Science and Engineering, Univ. of Notre Dame, Notre Dame, Ind., 2002.

[42] SIA. *Semiconductor Industry Association*. http://www.sia-online.org.

[43] SPARC International, Inc. *The SPARC Architecture Manual Version 8*. http://www.sparc.org.

[44] SPARC International, Inc. *The SPARC Architecture Manual Version 9*. http://www.sparc.org.

[45] A.S. Tanenbaum. From Commercial Supercomputer to Homebrew Supercomputer. *http://www.cs.vu.nl/ ast/talks/asci-97/*, 1997.

[46] Indiana University. *Local Area Multicomputer MPI*. http://www.lam-mpi.org.

[47] H. Veendrick. The future of Semiconductors; Moore or less. *The IEEE International Symposium on Circuits and Systems*, 2001.

[48] ERC32 Project website. *ftp://ftp.estec.esa.nl/pub/ws/wsd/erc32*. European Space Agency.

[49] Simics website. *http://www.simics.com*. Virtutech, Inc.

[50] E. Witchel and M. Rosenblum. Embra: Fast and Flexible Machine Simulation. *Sigmetrics*, 1996.

[51] Terry Tao Ye, Luca Benini, and Giovanni De Micheli. Packetized On-Chip Interconnect Communication Analysis for MPSoC. *Design, Automation and Test in Europe. Proceedings.*, March 2003.

Appendice A

Architecture model

A.1 Introduction

ML-RSIM provides a flexible and parameterized model of a parallel machine with N nodes. Each node contains several modules connected to a detailed model of a PCI bus: M CPUs with L1 and L2 caches, RAM banks, a RAM controller, a SCSI controller and a real-time clock device. Unlike RSIM, ML-RSIM does not simulate any interconnection network and nodes are independent and isolated. We first considered the feasibility of simulating a single node bundling several CPUs, in a configuration known as *Shared-Memory Multiprocessor*: in this scenario, a shared bus connects multiple CPUs and memory. The typical programming paradigm for this architecture, often referred to as *uniform memory access* (UMA), is the shared-memory model, since all processors share the same physical address space [15].
Unfortunately the kernel running on top of ML-RSIM, based on NetBSD and Linux, didn't provide at the time of this writing any multiprocessor support, preventing us from running any parallel application with such a configuration. As a matter of fact, although ML-RSIM inherits from RSIM a multi-processor framework, it focuses on a very detailed system-level simulation of I/O intensive workloads in single-processor, single-node machines [40]. We decided to leverage ML-RSIM's multiprocessor-ready simulation environment and enhanced it with key features, necessary in or-

der to run parallel applications. The following features have been added to ML-RSIM and its companion operating system Lamix:

- inter-node communication built on a **multi-layered micronetwork protocol stack** (section A.2)
- a model of an **interconnection network** (section A.3)
- a model of a **network interface** connected to the bus of each node (section A.4)
- enhancements to the **operating system** to add kernel support to the network interface module (section A.5.1) and basic multiprocessor features in the kernel (section A.5.2)

We extended ML-RSIM and implemented a model of *distributed-memory multiprocessor*, also known in literature as *multicomputer* [15]. In this configuration we have several nodes interconnected by a switched network; each node has a PCI bus connecting one CPU with its own L1 and L2 caches, memory and a network interface. Processors on different nodes don't share any memory therefore the only way they have to communicate is by passing messages across the interconnection network through nodes' network interfaces.
To complete an overview of the system across all hardware and software layers, on top of our extended ML-RSIM simulation engine we developed a *middleware* software layer from scratch (chapter B) and ran parallel computing benchmarks (chapter 7) to gain insights about the overall and layer-by-layer performance of the platform.

A.2 Layered protocol stack

The micronetwork features a stack of layers inspired by the OSI Reference Model [2]. The parameters of the network, such as width of the data channel, packet latency and throughput, pipeline stages, reflect typical values of Multiprocessor Systems-on-Chips (MPSOCs), in which several processors are integrated in the same chip. *Physical* through *transport* layers are

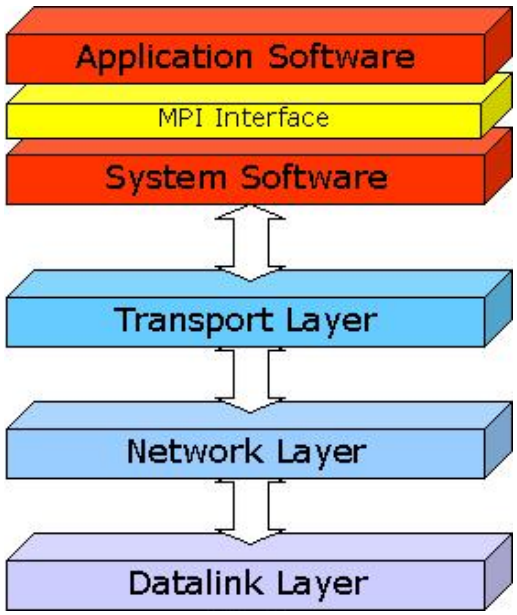


Figura A.1: Micronetwork protocol stack

implemented in hardware, the *system* layer is made of software routines included in the operating system and in the middleware and the *application* layer refers to user applications. Starting from the lower level of abstraction, the protocol stack can be described as follows.

Physical layer Nodes are connected by a switch fabric. Links can be noisy and non-reliable, but these effects are not currently simulated by the model. Transfers at this layer use *physical channel flow control* units or *phits*; a phit is the unit of information that can be transferred across the channel in one step or clock cycle. The size of a phit, which is equal to the width of the data channel, is a parameter of our model.

Datalink layer The packets are split into *flits* or units of flow control by the network interface for error detection and recovery. This layer manages hop-to-hop communication and resolves contention of network resources such as links, switches, internal buffers. In this architecture, the flit size is equal to the phit size and the width of the data channel.

Network layer This layer deals with end-to-end routing of packets across

the network switches. Source and destination node identifiers are included in the packet header delivered by the *Communicator* block of the network interface to the network (see section A.4.2). An alternative approach is represented by source routing: the "directions" for a packet are defined hop-by-hop at the source node [15] and delivered within the packet header along with a hop counter; at each hop, the counter is incremented and a switch connects the appropriate output port as described in the header. Following this technique, the network interface at the source node retrieves the routing path from a lookup table given the destination identifier. The table is initialized and possibly updated by the middleware, in accordance to traffic conditions or failures. However, although changes to the routing tables can occur at run-time, this approach is to be considered among deterministic routing algorithms, since routing path can't be changed adaptively once a packet has left the source node.

Transport layer A message is divided into packets for the purpose of flow control, error detection and buffer management by the *Controller* block of the network interface (see section A.4.1). In order to resolve contention in the network and to avoid congestion in relation to buffering resources, a *flow control* mechanism is required to manage information transmission. Flow control can be performed at two different levels: channels or switches may embed hop-by-hop flow control mechanisms at the datalink or network layer, managing data transfers with granularity of a flit or a packet; communication endpoints can similarly synchronize end-to-end message transfers at a higher abstraction layer. The latter approach is pursued in our model. The transport layer deals with error detection/correction and faulty data retransmission, providing upper layers with a reliable channel abstraction. While in macroscopic networks the same physical link is shared by data transfers and control signals, on-chip networks can take advantage of extended wiring capabilities and utilize dedicated control lines to evaluate the traffic on the network [51]. In our mo-

del flow control information don't share the same medium with data transfers.

System layer This layer is made of two software blocks: a *device driver* embedded in the operating system and a *middleware* implementing communication services among processes running on different processors.

The device driver is a kernel module which makes all services provided by the network interface accessible and usable by software applications (see section A.5.1 below for details).

The middleware provides the application layer with a common framework for inter-process and inter-node communication leveraging operating system and device driver services. The framework we chose adopts a message-passing programming paradigm as defined in the *Message Passing Interface* (MPI), which is the *de facto* standard in the industry and academia (section B.2.1). MPI defines communication end-points, message tags and datatypes and synchronization behaviors. At this layer source and destination identifiers are not nodes but distributed processes running on multiple processors. Several synchronization semantics of *send* and *receive* operations – such as synchronous, asynchronous, blocking, non-blocking, ready *MPI communication modes* [20] – call for handshaking and rendezvous protocols, tightly coupled with different buffer management policies [10]. Chapter B discusses further aspects of the tasks and implementation details of the middleware layer.

Application layer The user's software application makes use of communication services provided by lower layers through a well-defined *Application Programming Interface* or API exported by the middleware. We ran some benchmarks on our simulated architecture drawn from parallel computing literature [36], such as parallel matrix multiply and PI calculation using dashboard algorithm, using the core subset of point-to-point MPI primitives.

Tabella A.1: Micronetwork protocol stack

Layer	HW/SW	Data Unit	Logic block
Application	SW	MPI Message	MPI Application
System	SW	OS Message	Middleware - OS
Transport	HW	Message	Network Interface
Network	HW	Packet	Switches
Datalink	HW	Flit	Links
Physical	HW	Phit	Physical channel

A.3 Interconnection Network

Nodes are connected through a *wormhole-routed* switched network. In wormhole routing, proposed by Dally and Seitz [13], a packet is broken up into *flits*, which are the units of message flow control. The size of a flit depends on system parameters, in particular the channel width. Wormhole routing uses a cut-through approach to switching. The packet header containing routing information is examined as soon as it is received; in our architecture the packet header fits in just one flit. Switches can start forwarding flits to the destination just after taking routing decisions based on header's contents, therefore packets are pipelined through the network at the flit level.

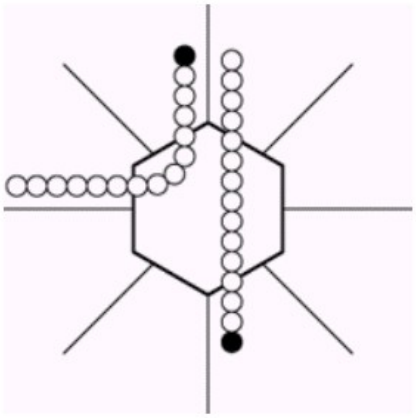


Figura A.2: Wormhole Routing

The average number of pipeline stages along communciation paths in

the network is a parameter of our model. If the header flit encounters a channel already in use, it is blocked by the flow control mechanism until it becomes available while the trailing flits remain in flit buffers along the path.
The overall packet latency due to buffering and routing in the network is simulated using a programmable delay for the first flit, while subsequent flits are delivered to the destination one per clock cycle. Additional flit delay can be caused by contention and congestion at the destination node, through a back-propagating flow control mechanism that uses dedicated control lines.
The presented architecture adopts a deterministic routing since the same path is followed for any given pair of source and destination, whithout taking into account network traffic conditions adaptively. The network does not include at this stage a detailed model of switches and channels, therefore contention at buffer and port level is not simulated. Furthermore routing and forwarding delays introduced by switches are considered on an overall average basis and every single node is virtually equidistant from any other node, without simulating any particular physical topology. *Virtual channels*, in which different packets share a physical channel through time multiplexing, are not supported.
Since flits are allowed to hold *resources* while requesting others during the pipelined transmission, wormhole routing is susceptible to *deadlock*. Resources in this case are channels and buffers. The deadlock problem can be solved by allowing *packet preemption* or by the routing algorithm [35].

- In packet preemption, packets involved in a possible deadlock can be discarded and retransmitted or rerouted; in most architectures this solution is not used because of low latency and reliability constraints.
- The routing algorithm can impose a strictly monotonic order on network resources in order to avoid a necessary condition for deadlock, a *circular wait* scenario.

In our simplified model, network resources are not simulated individually and network behavior is emulated with aggregated parameters.

Each transmission has to reserve one aggregate channel resource and deadlock cannot occur.
Note the difference between deadlock at the network layer – with flits competing for hardware buffers and channels – and deadlock at system software layer – with messages competing for software resources such as device driver access and circular queues – as described in section B.9.

A.4 Network Interface

Each node features a network interface that links the node to the on-chip interconnection network. The network interface is attached to the node PCI bus and mapped into the processor I/O space. There are no shared memory locations between any two nodes, therefore if a node requires a remote object, a data transfer on the network is necessary. Network transactions can be initiated only by the network interface. The purpose of the network interface is basically to send messages from the node bus to the interconnection network and viceversa; this implies formatting and packetizing messages, buffering incoming and outgoing messages, interfacing with the bus and providing consistent state information through I/O registers. The node's network interface is made of two functional blocks, a *Controller* and a *Communicator*.

A.4.1 Controller

The first block, the *Controller*, simulates a hardware layer providing buffer management and interface mechanisms between network and datalink layers. This module is connected to the intra-node bus and maps registers to the non-cacheable I/O address space, namely read and write registers, status (e.g. transmit, receive ready), communication node identifier and message size registers. The size of read and write registers is a parameter of this hardware module. The processor architecture modeled by ML-RSIM requires memory-mapped I/O locations. Memory pages can be associated to different I/O devices and tagged as non-cachable using simu-

lator directives and operating system calls during the kernel initialization. The Controller splits outbound messages into packets and flits, formats a proper header containing routing information (e.g. source and destination identifiers, number of flits in a packet, path directions in the case of source routing) and finally enqueues flits in a circular output buffer. Incoming flits stored in a input buffer are merged and delivered to the processor through a read I/O register. Input and output buffer size is another parameter of the model.
A I/O status register is consistently updated to ensure a correct behavior of the system communication suite (OS device driver and middleware). The network interface does not raise any hardware interrupt when it receives data from the micronetwork, but it stores incoming packets in internal hardware buffers. The device driver *polls* the state register to find out if the buffers of the network interface contain some new data and to ensure that they are ready to receive to receive new data from the processor in the node. If input buffers get full and no other incoming packet can be stored, a flow control signal stops the transmission from the interconnect network. Similarly if output buffers are full because the network is congested, a busy bit is set in the state register to inform the device driver that it can not initiate bus transactions.

A.4.2 Communicator

The second block, the *Communicator* performs the actual data transfers through the network. Communicator and Controller interact in a consumer-producer scenario through an input buffer and an output buffer. The two circular buffers operate as mailboxes of flits.
The Communicator *consumes* flits in the output buffer and delivers them to the network. The current model defines as a parameter the latency to deliver the first flit of a packet to the destination (in clock cycles); similarly the throughput can be defined in cycles per flit.
Dually, the Communicator *produces* flits in the input buffer by storing incoming data from the interconnection network.

A.4.3 DMA mode of operation

The network interface module features also a *direct memory access* (DMA) mode of operation. Whereas in *standard mode* the system software feeds the hardware buffers of the network interface with I/O transfers to its read and write registers, in *DMA mode* the bus transactions take place without software and CPU intervention.
The system software programs the message transfers providing the network interface simply with a physical address and a message length. The network interface handles transfers directly and updates its status register when the transfer is complete.
Actual bus transactions and cache snooping are not currently simulated; within the simulator, the device can change user memory correctly without interfering with the cache coherence protocol. However, since statistics collected running benchmark on the platform showed that data cache hit rate is beyond 95% (section 6.8), the overall performance and behavior of the system is not heavily affected by neglecting this source of bus contention.
A parameter defines how many cycles it takes to copy a flit from memory to network interface's buffers and viceversa.

A.4.4 Interaction with software layers

The middleware layer can communicate to the Controller module of the network interface through the device driver embedded in the operating system kernel. To request a network transfer, the device driver checks the availability of the network interface module and, if ready, locks it for exclusive use while sending a message. A header containing the length of the message and source and destination identifiers are provided to the network interface. The data to be sent across the network is transfered to the output buffer through I/O accesses to the Communicator module's registers by the device driver.
In the software architecture we devised, the network interface can be accessed only by the middleware though system calls to the device driver

and not directly by user applications. This maximizes portability and reusability of the application software through a layered design approach and ensures protection of different processes' messages in a multiprogrammed environment, although it incurs overhead caused by switching to privileged mode and transfering control to the kernel with software traps.

A.5 Operating System Enhancements

The operating system shipped with ML-RSIM is based primarily on NetBSD, a project originated as an effort to port Unix BSD to non-Intel platforms [41, 26] such as SPARC. This OS is not aware of other nodes running in parallel within the simulator engine and it has been modified to support our interconnection model.
The "gate" to the internode interconnection network for a CPU is represented by the network interface. The network interface is a hardware module connected to the node's bus and is considered by the operating system as an I/O device. Access to I/O devices in most computer architectures is granted only to software routines running in privileged mode; these routines are kernel modules known as device drivers. In section A.5.1 we present the device driver we developed to communicate with the network interface.
When executing a parallel application distributed over several processor, each node must be assigned a unique identifier. The operating system is to provide services to find out what node and what processor within a node a process is running on. In section A.5.2 we describe the system calls we added to the OS kernel to meet these requirements.

A.5.1 Network Interface Device Driver

The network interface device driver is a software component running within the operating system kernel. Its function is to communicate with the network interface hardware module through I/O transactions on the bus

and make its services available to upper software layers, namely the middleware.
In our SPARC-based architecture I/O transactions translate into reads and writes to non-cacheable memory locations mapped to network interface registers while initializing the kernel at boot-strap time.
The device driver is accessible by the middleware through system calls; a software trap transfers the control to the kernel, which in turn invokes the driver's routines in privileged mode. The device drivers implements two set of system calls, in order ot utilize the network interface in *standard mode* and in *DMA mode*.

- In *standard mode* operations, the actual contents of the message are transferred by software-controlled bus transactions. The CPU is busy issuing I/O read and write instructions to memory-mapped I/O locations. In this case it is the device driver to split messages into packets, whose size corresponds to the size of I/O read and write registers. `NI_Send()` and `NI_Receive()` system calls are provided; their arguments are a pointer to a buffer in user memory, the length of the message in bytes and the destination or source node. Standard mode driver routines operate as follows:

  1. update and check network interface's state register, in order to synchronize read and write operations with other processes and the network interface buffer manager.
  2. program the destination or source node identifier and message length
  3. split the message in several packets, whose length is equal to I/O write and read registers' size. Transfer the contents of the message in burst transfers to I/O registers.
  4. return a status code and the number of transferred bytes to the caller

  The semantics of standard mode system calls is *blocking*: if the network interface is busy or empty, they keep polling on status register

until the message can be sent or received completely. Applications can safely change the contents of the user buffers after system calls return. Statistics obtained running benchmarks on the architecture (section 6.9) made clear that packetization and bus transfers have to be carried on by the network interface in hardware.

- In *DMA mode*, the network interface can fetch and deliver the message contents between its hardware buffers and user memory without CPU's intervention. The network interface also splits messages into packets and flits for flow control and error detection purposes. `NI_DmaSend()` and `NI_DmaReceive()` system calls are provided; similarly to standard mode operations, their arguments are a pointer to a buffer in user memory, the length of the message in bytes and the destination or source node. DMA mode driver routines operate as follows:

  1. update and check network interface's state register, in order to synchronize read and write operations with other processes and the network interface buffer manager.
  2. program the destination or source node identifier and message length
  3. convert the virtual address in user space into a physical address
  4. send the physical address to the network interface and trigger a DMA transfer
  5. return a status code

  The semantics of DMA mode system calls is *non-blocking*; they return as soon as the network interface has been programmed properly to execute the bus transaction. While the network interface carries on I/O transfers between memory and its buffers, the CPU can procede with its execution flow concurrently. In order for the caller application to know whether the buffer has been transfered to network interface input buffers and it can be read or modified, the device driver introduces `DmaProbe` system calls as synchronization

points. `NI_DmaProbeSend()` and `NI_DmaProbeReceive()` simply access network interface state register to check `SEND_DONE` and `RECV_DONE` flags.

The middleware invokes `NI_Send()` and `NI_Receive()` to transfer data from the network interface to a software buffer pool and viceversa, whereas the device driver does not include any system buffer. The middleware maximizes parallism of computation and communication, in cooperation with the OS scheduler; it avoids when possible busy waiting loops by switching to other user processes when the network interface is busy or empty. User applications make use of more abstract `send` and `receive` routines as defined by *MPI* standard and do not invoke directly device driver's system calls; such high-level routines are implemented by the middleware (see chapter B).

A.5.2 Multiprocessor support

When implementing communication libraries for parallel architecture as a middleware layer, it is key to identify what node and what processor within a node a given process is running on.
On one hand there is the need to define a proper and unique user-level identifier for each communication end-point across all the processors in the system.
On the other hand the communication library needs to distinguish between processes running on the same processor, or on different processors of the same node, or in different nodes in order to decide how to transfer the data. In the former case, interprocess communication mechanisms such as pipes, channels, signals or shared variables in memory shall be used. In the second case the two end-points share the same physical address space and a shared-memory approach can be pursued. In the latter scenario, the communication layer shall start a I/O transaction with the network interface which in turn will send packets over the interconnection network.

The developer of a parallel application can privilege portability over

performance using standard libraries such as MPI (see section B.2.1) and letting the system allocate the processes across available computing resources at run-time. However, transparency of local and remote data communication introduced by the middleware layer implementing the communication library reduces the predictability of the system; in fact the same user-level operation involving data transfers between processes will have very different outcomes in terms of latency and thoughput depending upon their actual mapping to processors. A reduced predictability causes several issues in the design of an embedded real-time application which has to meet well-defined time constraints and deadlines.
When fine-tuning the application for a particular configuration, the software designer can choose to invoke directly operating system primitives by-passing the library, in order to gather detailed information about the physical topology of the parallel application over the processing architecture and control more precisely and effectively the behaviour of the system.

We added two system calls to the Lamix kernel: `getnode()` and `getprocessor()`. A standard UNIX convention has been sought, implementing the system calls as procedures invoked by the software trap manager in privileged mode. The kernel uses the underlying ML-RSIM interface to discover current node and CPU.
Note that in our configuration, where each node contains only one CPU, getprocessor() always returns 0.

Appendice B

Middleware for Networks on Chips

B.1 Introduction

The communication layers in a network-on-chip architecture follow the structure proposed by the OSI Reference Model for wide area computer networks. A layered methodology, described in section A.2, allows a modular hardware and software design and maximizes reusability of intellectual property and program code while facing reliability, quality-of-service and power-consuption issues of modern SoC design [2].
The middleware is a layer of system software running on each node processor whose main objective is to manage effectively interaction and communication among processes. Processes can be local to a node - in which case interaction is configured as a typical interprocess communication - or remote objects distributed across several nodes, in a way not dissimilar to distributed systems in geographical computer networks.
The programming paradigm we chose for this architecture is *message passing*, as described later in section B.2. The middleware layer exports to the higher levels of abstraction an *application programming interface* (API) based on the industry standard *Message-Passing Interface* (MPI) (section B.2.1).

B.2 Message Passing Architecture

Message passing is a programming model in which information is conveyed from a specific sender process to a specific receiver process. There is no shared location visible to all processes and communication is performed at the *application level* (or *user level*) through explicit *send* and *receive* calls.
Whereas in *non-uniform memory access* (NUMA) architectures, distributed memory locations are accessed transparently through the memory subsytem, in a pure distributed-memory message-passing architecture, communication is integrated at the I/O level. At a lower abstraction level, the *middleware* that implements a message-passing programming interface will perform many hardware and system level actions, such as buffering, managing bus transactions with the network interface, dealing with remote synchronization and handshaking protocols, besides performing the actual communication [10].

B.2.1 Message Passing Interface

MPI is a Message Passing Interface Standard for parallel machines, in particular those with distributed memory architectures, supported by major players in the industry, as well as research centers and universities. The first draft, known as MPI-1, was issued by the MPI Forum in 1992, followed by MPI-1.1 in 1995 and MPI-2.0 in 1997 featuring extensions such as C++ and Fortran-90 bindings, parallel I/O and dynamic processes.
The main design goal behind MPI is performance enhancement through overlapping of computation and communication. MPI defines a set of high-level routines and abstractions or API (Application Programming Interface), aimed to maximize portability and ease-of-use of parallel programs across several hardware platforms and programming languages. The interface is then implemented using lower level machine-dependent message passing algorithms over different parallel architectures such as

nCUBE Vertex, p4 and PARMACS, Intel NX/2, Cray T3D and T3E, IBM SP2, Thinking Machines CM-5, NEC Cenju-3 [20, 19].
In our case, the underlying hardware platform is a *distributed-memory multiprocessor*, as modeled by ML-RSIM's simulation engine (see Chapter A).

B.2.2 MPI Implementations

There exist several implementations of Message Passing Interface, both commercial and non. Cray, IBM, HP, Hitachi, SGI, NEC, Sun provide commercial MPI libraries for their parallel architectures.

Tabella B.1: Commercial MPI implementations

Vendor	Platforms
Cray	T3D/T3E
Hitachi	SR8000
HP	s700 and s800 HP-UX
IBM	RS/6000 and OS/390
SGI	IRIX 6.5 and UNICOS 9.3
NEC	SX Series
Sun	Solaris/UltraSPARC

Non-commercial implementations focus on clusters of POSIX-compliant systems (LAM/MPI by Indiana University), *Network of Workstations* (*NOW*) such as TCP-connected networks of Linux machines (DISI by University of Genova, Italy) or Macintosh (Appleseed by UCLA) and Myrinet clusters (MPI-FM by University of Illinois) [46]. Finally MPICH, an open-source portable MPI implementation by Argonne National Laboratory, supports most parallel architectures and TCP-connected networks of workstations [21, 23].

B.3 Implementing MPI as a middleware layer

The middleware consists of a user-level process or *daemon* running on the CPU of each node and a set of library functions accessible to users. The middleware daemon operates data structures and communicates with the

node network interface through system calls to a device driver; it provides interprocess communication and remote interaction mechanisms leveraging the capabilities of the underlying operating system and hardware model (Section B.5). User applications can utilize these communication services by linking to a library which implements a subset of the MPI industry standard (Section B.6). This architecture is shown in figure B.1.
Each node may have several concurrent processes participating in local and remote communication transactions, thanks to a Unix-like *multitasking* support provided by the kernel. However, light-weight processes or *threads* are not supported as concurrent execution units within the same process, i.e. the software environment is not *multithreaded*.
Implementing the middleware as a user process heavily reduces overhead due to context switching between user and privileged (or system) mode. At the same time, protection and correct behavior of user processes is guaranteed by granting access to communication operations through a well-defined application interface. As in most computer architectures, access to I/O devices is reserved to system code through *device drivers*.

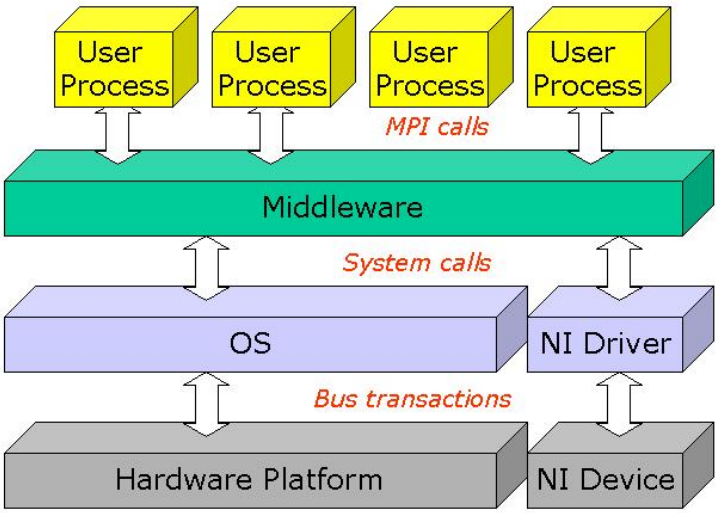


Figura B.1: Middleware implementation

MPICH and LAM-MPI provide a stable and full implementation of the MPI standard (Section B.2.2 for details), but they require full POSIX or

Solaris synchronization and shared memory features, not available in our simulation environment. Other MPI implementations support ethernet-connected multicomputers or proprietary multiprocessor configurations, but our simulated architecture presented some unique features that prevented us from utilizing exisiting message passing libraries.
The middleware implements the core of the MPI specifications, represented by functions `MPI_Init()`,`MPI_Close()`,`MPI_Send()`,`MPI_Recv()`, `MPI_Count()`,`MPI_Comm_rank()`,`MPI_Finalize()`.

B.4 Middleware Functional Blocks

The middleware daemon on each node allocates and manages the following data structures: an *MPI manager*, a *mailbox* and a *queue of pointers* per each endpoint and an *input buffer* to store incoming messages from the network interface.

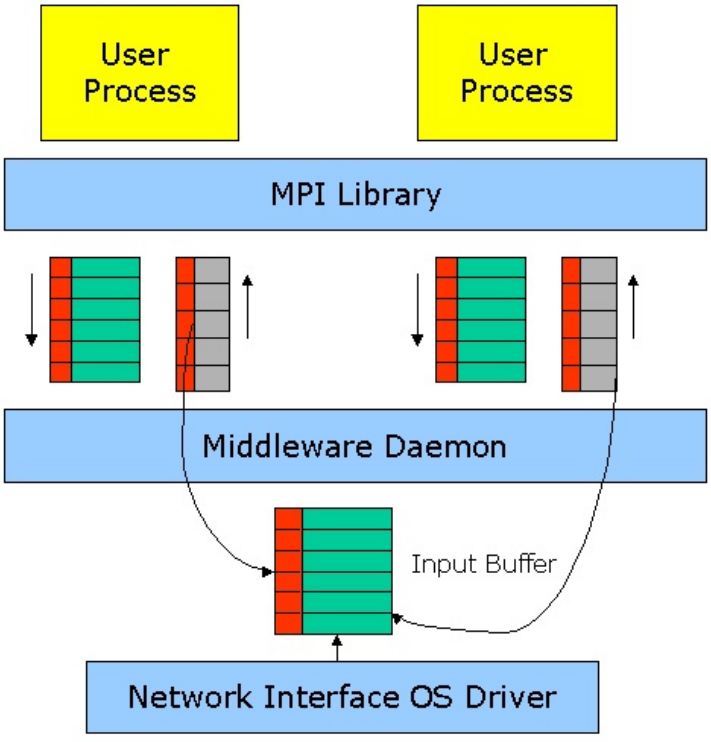


Figura B.2: Middleware Functional Blocks

B.4.1 MPI Manager

In order to keep track of node-wide (local) information, upon invocation of `MPI_Init()` by a process in the node, an *MPI* manager structure is created. It contains information such as the number of local active endpoints and of outstanding send requests waiting for `ready-to-receive` messages from remote processes. When both numbers reach zero, the middleware process exits, since every process involved in MPI communication has invoked `MPI_Finalize()` and terminated.

B.4.2 Process Mailbox

One mailbox per endpoint process is allocated to store outgoing messages. Local and internode messages are stored by the middleware while waiting for a matching ready-to-receive object (Section B.11.2). A `ready-to-receive` object contains an *MPI message header*, i.e. source and destination MPI end-point identifiers, message tag and datatype; all these fields have to match with the header of a message in the mailbox before any actual data transfer can start [1].
The data transfer can be memory-to-memory for local message-passing or memory-to-network interface for internode communication. In the latter case, the message can be transfered from the mailbox either by the CPU via software or by DMA (Sections A.4 and A.5.1).
If only *synchronous* `MPI_Send()` is used, each process has a one-message mailbox, since the flow of execution is blocked until the message has been received and acknowledged by the recipient. In order to use *buffered* `MPI_Send()`, this mailbox can contain many outstanding messages waiting to be sent to the network, while sender processes continue their tasks (Section B.11.1 for details about semantics of `send` and `receive` operations).

[1] The MPI standard defines data type conversions and *wildcards* for source and destination fields as a more flexible matching rule

B.4.3 Process Pointers Queue

Each communication endpoint has its own *inbox*. In order to minimize memory-to-memory transfers of the same message across different software layers (application, middleware, operating system), inboxes are not implemented as queues of messages. Pointers to other locations within middleware data structures are used instead. If the message has been delivered from a remote process, the pointer links to a slot in the input buffer (Section B.4.4 below). If the message was sent by process P_1 running on the same node, the pointer links to a slot in P_1's mailbox (Section B.4.2 above).

B.4.4 Input Buffer

The middleware keeps a queue for incoming `ready-to-receive`, `transfer` and `receive-acknowledge` objects from the network interface. The contents of the input buffer are updated by the *middleware daemon* polling the network interface for new data. If the network interface is empty, the OS scheduler is invoked to switch to another process before entering any busy waiting loop.

Each slot in mailboxes, pointers queues and inputbuffers has a synchronization flag which describes its status in the communication procedure.

`FREE` the slot is empty and ready to be modified

`CLAIMED` the slot has been reserved and a memory (local) transfer is in progress

`SENT` the slot contains the message to be sent

`RECEIVING` a remote transaction is in progress, the message cannot be freed yet

`DELIVERED` an acknowledge message has been received from the message recipient. The sender can now mark the slot as free.

The flag is used as a synchronization point between the *middleware daemon* and the MPI processes.

These data objects are allocated in shared memory in order to allow intercommunication between the middleware and each MPI process. A semaphore is associated with each object to grant mutual exclusive access and consistent operations. Access to shared structures is performed by the middleware process and a user library to grant protection of different user spaces.
The kernel shipped with ML-RSIM, *Lamix*, reserves the segment between 0x6000000 and 0x8000000 for memory pages shared among the processes on the same CPU; this segment is then mapped to the virtual address space of each process at address 0x60000000. Lamix implements the system calls `shmget()`, `shmat()`, `shmctl()`, compliant with System V shared memory support, in order to allocate, map and control shared pages. [41]
Processes which register as communication endpoints and the middleware layer interact through buffers and mailboxes allocated in shared memory. With this approach, most MPI communication operations can be executed at the user-level without requiring performance inefficient system calls. Providing an effective mechanism of synchronization among processes is key for a functional and efficient implementation of a message passing library in a multitasking node (Section B.8).

B.5 Middleware Daemon

One instance of the middleware runs on each node as a user daemon process. It initializes the data structures described above and manages transactions from the network interface to the *input buffer* and from *mailboxes* to network interface. Transfers of messages between network interface and middleware are handled through system calls to the operating system, namely to the network interface device driver.
When using *DMA mode* system calls, the middleware can procede in paral-

lel until a synchronization point is needed because of data dependencies; when message contents need to be modified or read, a `probe` operation ensures the DMA transfer has executed to completion. *Standard mode* system calls have a blocking semantics and the middleware has to wait for the whole message to be copied into memory by the device driver, causing heavy performance penalties, possibly buffer congestion in the network interface and even network congestion in communication-intensive workloads (Section 6).

The middleware daemon runs a cycle until no more active MPI endpoints are left and no more buffered messages need to be sent. The cycle performs two tasks:

1. transfers available messages (if any) from the network interface to the middleware input buffer

2. goes through the input buffer analyzing stored messages to conform to application-level protocols (Section B.11.2) and to initiate new network transactions.

Application-level messages in the input buffer can be of three types: `ready-to-receive`, `transfers` and `receive-acknowledge`. There are no `request-to-send` messages because the application-level protocol is *receiver-initiated* (Section B.11.2).

- every `ready-to-receive` object is matched against the headers of messages stored in process mailboxes. If a matching message is found, the middleware initiates a transaction with the operating system device driver to transfer the contents of the message to the network interface.

- every `transfer` object is linked to a slot in the pointers queue of the recipient process and the synchronization flag changed to `SENT`. The user library can now transfer the data directly from the middleware input buffer to the user space.

- every `receive-acknowledge` object is matched against the headers of messages stored in process mailboxes. If a matching message is found, the synchronization flag is set as `DELIVERED`.

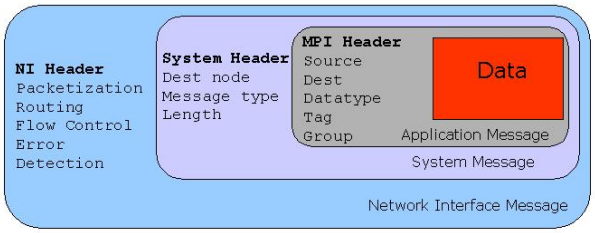


Figura B.3: Message encapsulation

B.6 Middleware User Library

The middleware user library implements MPI operations and is linked at compile time to user processes. It handles the transfer of data across application and middleware layers with one memory-to-memory copy:

- outgoing messages are copied from the user buffers to the middleware mailbox.
- incoming messages are copied, going through the pointers queue, from the input buffer (remote messages) or a process mailbox (local messages) to the user buffers.

The MPI library also realizes different semantics for its services by synchronizing on middleware flags. For example a synchronous `send` is realized waiting on the corresponding flag in the mailbox to become `DELIVERED` before returning, whereas a buffered `send` returns as soon as the flag gets the value `SENT`, without waiting for an acknoledge.

B.7 Unique identifiers

"Source" and "destination" of communication operations hold a different meaning for each abstraction layer. At physical layer, each point-to-point

link connects unambiguously two components of the network, such as switches and network interfaces. At datalink layer, switches need to distinguish different input and output ports for each packet depending on resource availability and routing information from the upper layer. At network and transport layers, source and destination are nodes (processors attached to the switched network).
The system software extends the notion of communication endpoints from hardware node identifiers to more abstract process identifiers.

Node identifiers The OS device driver programs the network interface with a destination node identifier when sending a message. The message handed to the device driver by the middleware contains an MPI header with source and destination endpoints, of which neither operating system nor network interface and lower layers are aware of.

Communication endpoint identifiers The middleware exports to the application layer an abstract interface compliant with the MPI specifications (Section B.2.1). Each MPI message has an header with source and destination identifiers, data type, length and a tag field. In this context source and destination are user processes that registered themselves as *MPI communication endpoints* calling `MPI_Init()`.

To create network-wide unique identifiers, we used 4-byte fields. The 2 most significant bytes store the node identifier, while the 2 least significant bytes store the endpoint identifier. The middleware keeps track of local and remote processes, executing either interprocess communication through UNIX shared memory for local interaction or invoking operations of the network interface device driver for inter-node message transfers.

B.8 Concurrent access and synchronization

Concurrent access algorithms fall into three categories.

- Traditional algorithms are *locking*, in the sense that processes enter critical sections in a mutual exclusive fashion; when a process holds an exclusive lock, all other processes are delayed until the lock is released and no concurrent access to a critical section is allowed.

- *Non-blocking* algorithms do not enforce mutual exclusion and hence avoid that processes wait to be granted access to a critical section for arbitrary amounts of time. A process reads a value from a data structure, performs all the computation based upon this value and atomically writes back the results to the data structure. If another process changed the value during a computation phase, the results are discarded and the computation restarted. Major drawbacks of these algorithms are the overhead necessary to guarantee an atomic transfer of the results and the cost of restarting and reexecuting computation tasks.

- Applying problem-specific information to non-blocking algorithms to tackle these performance issues and optimizing for the most common cases lead to *lock-free* algorithms.

Lamix does not provide any kernel support for POSIX or System V semaphores, locks and message queues; furthermore kernel-based synchronization is known to heavily reduce performance because of context switches and system calls overhead. We implemented our own synchronization mechanism for the message passing library.
Several concurrent access algorithms can be drawn from parallel architectures literature; algorithms suitable for multiprocessor environments rely on instructions which execute a load and a store atomically with respect to all other memory accesses. Examples of these instructions, provided natively by most processors commercially available, are *Test & Set*, *Swap* and *Compare & Swap* [33].

The processor modeled by ML-RSIM features both *Test & Set* (`LDSTUB` in SPARC instruction set) and *Swap* instructions; it lacks *Compare & Swap*, introduced with SPARCv9, which prevents us from adopting any lock-free

algorithm. Our locking mechanism is now based on a spin lock with exponential backoff. A spin lock is a lock for which the "lock held" condition is handled by busy waiting [44]. Exponential backoff consists of geometrically increasing waiting times in order to reduce CPU contention.
Another approach is based on *ticket locks*: to obtain a lock, processes acquire a ticket number and wait for a service counter to show the same number. The order on processes waiting on a lock imposed by ticket lock algorithms reduces cache-coherence traffic but hardware support for an atomic *Fetch & Add* operator is needed in order to prevent starvation [33]. This operator was not featured by the instruction set simulated in our processor model so we didn't implement ticket locks.

B.9 Deadlock prevention

When using software locks in order to guarantee mutual exclusive access to resources (buffers, queues, shared variables for instance), a *deadlock* may occur. A deadlock is a situation in which different processes are waiting on locks that will never be released. For example, suppose two processes P1 and P2 running on a single processor need to be granted mutual exclusive access to resources A and B.

P1	P2
lock(A)	lock(B)
work on A	work on B
lock(B)	lock(A)
work on B	work on A
unlock(A)	unlock(B)
unlock(B)	unlock(B)

For certain instances of CPU scheduling, P1 and P2 will be undefinitely blocked on the locks, for example:

P1	P2
lock(A)	
work on A	
	lock(B)
	work on B
lock(B)	
	lock(A)

In this example, P1 locks the resource A and waits on the resource B, that will never be released by P2 which is in turn blocked waiting for A.
Directed graphs can provide a description model for deadlocks: we define a system *resource allocation graph* (V,E), with V partitioned into two types of vertices, the set of active processes P and the set of all resource types R. A directed edge $P_i \rightarrow R_j$ is called *request edge*. A directed edge $R_j \rightarrow P_i$ is called *assignment edge*.
In the hypothesis of single instance resources, a deadlock occurs if and only if there is a cycle in the resource allocation graph. In order to eliminate a circular wait condition and thus prevent deadlocks, one can define a global order on resources and ensure the same order for requests by processes. A stricter condition is the following: a process must release all resources before acquiring a new one. The system software we developed implements the latter approach.
At a different layer of the architecture – the network layer – we also mentioned a deadlock problem. Note that the context is very different; in that case, flits acquire resources such as physical channels and hardware buffers along their path across network switches (Section A.4).

B.10 Network interface management

Middleware access the network interface only through system calls provided by the device driver embedded in the operating system. In the modeled architecture, every I/O device is mapped into a protected address space to which only privileged code (i.e., the OS kernel) can be granted

access. Whereas this "filtered" access causes a heavy performance penalty, it guarantees protection and correctness of operation since only trusted software modules can operate with the network interface.
On the other hand, it would be possible to conceive an architecture where user-level code is allowed to access intercommunication hardware components directly, for performance reasons. Security and protection could be achieved by using a library (embedded in the middleware) to interact with the network infrastructure.

B.11 Send and Receive operations

In the message-passing programming paradigm the software designer utilizes explicit `send` and `receive` operations to express inter-process and inter-node communication. Several semantics can be specified for these functions at the application level, in relation with buffering policies and synchronization behaviors (Section B.11.1).
In order to properly implement these policies, the middleware layer provides mechanisms such as handshake protocols (Section B.11.2) and buffer management (Section B.11.3).

B.11.1 MPI Communication modes

The message-passing interface standard defines different *communication modes* or semantics for `send` and `receive` operations.

Blocking `send` and `receive` return only when the buffer containing the outgoing or incoming message can be safely accessed and modified by the caller.
On the contrary, *non-blocking* operations initiate communication but do not complete it before returning. The application developer shall then use a `probe` function to verify that operations have concluded with a `probe` operation. Non-blocking communication primitives allow a higher overlapping of computation and communication. Suitable hardware, such as a DMA controller, can carry on data transfers while the CPU continues to

execute the program until the *probe* synchronization point.

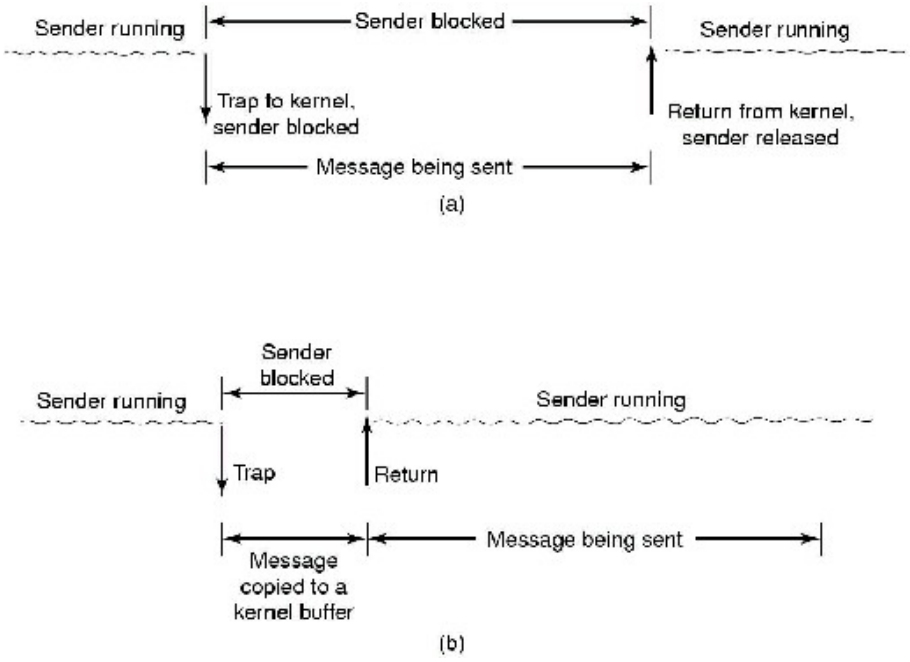


Figura B.4: Semantics of `MPI_Send()`

MPI defines several semantics for both blocking and non-blocking operations. A *synchronous* `send` returns control to the caller only after a matching `receive` has been posted, the data has been transfered succesfully and stored in the application space and an acknowledge has been returned to the sender. A synchronous send is *non-local*, as its completion depends on the occurrence of a matching receive, possibly on a remote node. Coupled with a blocking receive, it represents a synchronization point or *rendez-vous*.

A *buffered* mode `send` operation may complete before a matching receive is posted. The MPI implementation must buffer the outgoing message so that a matching `receive` is not required for its completion (*local* `send`). The amount of available buffer space is controlled by the user and insufficient space shall throw an exception. The MPI standard also allows the user to attach buffering resources to communication operations for a more precise and closer control.

The *ready* communication mode provides that a matching `receive` has already been posted. This saves handshake protocols at lower levels and may improve performance.
MPI *standard* or default mode does not mandate buffering of outgoing messages and the `send` operation is to be considered *non-local*, i.e its completion depends on the occurrence of a matching `receive` [20, 10].

Our MPI library implements blocking, synchronous and buffered `send` and blocking `receive`.

B.11.2 Application protocol

Application-level protocol is based on a simple *receiver-initiated* handshake mechanism. Traditionally, in a 3-way handshake protocol, the sender initiates the transaction with a `request-to-send` message. After a `ready-to-receive` is delivered from the recipient, the data can be transfered. At the end of a successful transmission, the destination node sends back a `receive-acknowledge`, as shown in figure B.5.

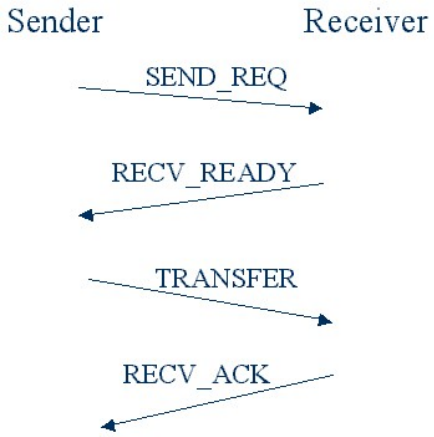


Figura B.5: Handshake Protocol

However, the MPI standard defines a message *header* which specifies source, destination, datatype and a tag field for each message. In this case, where every message can be identified easily by its header, a receiver-

initiated protocol can be devised [10]; by doing so the latency is reduced by a significant amount that would be associated to the transmission of the `request-to-send` as usual across all software and hardware layers (Figure B.6).
The receiver sends a `ready-to-receive` object to the source of the message with an MPI message header attached. At the sender, the middleware layer will match the header with the pending requests of (MPI_Send()) and start the proper data transfer.

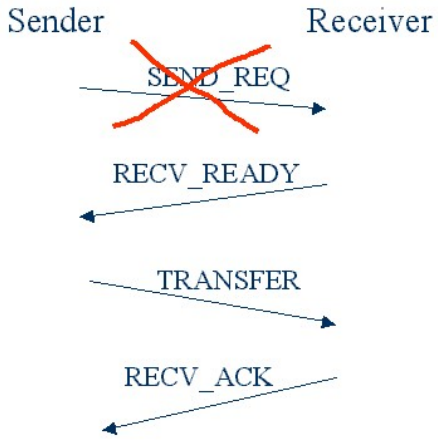


Figura B.6: Receiver-Initiated Handshake Protocol

B.11.3 Buffer Management

Buffer management is strictly related to synchronization semantics of communication functions. *Synchronized* `MPI_Send()` reduces the need of buffering resources because the execution proceeds only after the data transfer has completed and there are no pending requests. In this case just one-message buffer can be allocated by the middleware, at the expense of an execution flow which is blocked waiting for an acknowledge from the receiver. With *buffered* `MPI_Send()` the overall situation is very different since the middleware is supposed to guarantee enough buffer space for outstanding requests to send.
A precise profiling and tuning phase in the design flow is key to enhance the performance of an embedded application; the designer should be

able to reconfigure the middleware in order to find the optimal buffer size for *inbox* and *outbox* queues in what turns out to be a trade-off between memory resources (and power consumption) versus overlapping of communication and computation.
More importantly, the size of buffers may be related even to correctness of the system with respect to pending messages defined by the handshake protocol. For sake of simplicity, assume *blocking* send and receive and an input buffer of 2 messages. Consider the following pseudocode snippet:

```
Node 0                Node 1                Node 2                Node 3
send(msg1, node1);    recv(buff1, node0);   recv(buff1, node0);   recv(buff1, node0);
send(msg2, node1);    recv(buff2, node0);   recv(buff2, node0);   recv(buff2, node0);
send(msg3, node1);    recv(buff3, node0);   recv(buff3, node0);   recv(buff3, node0);
send(msg1, node2);
send(msg2, node2);
send(msg3, node2);
send(msg1, node3);
send(msg2, node3);
send(msg3, node3);
```

`recv` calls cause node 1, 2 and 3 to send `ready-to-receive` to node 0. Assume that the messages from node 2 and 3 arrive before the one from node 0 and fill up the 2-location input buffer at node 0. Node 0 will wait forever for a `ready-to-receive` to execute its first `send`.
Using *non-blocking* or *buffered* communication functions adds more complexity to this crucial problem, because multiple `ready-to-receive` messages could flood the input buffer at the sender.

Appendice C

Porting RTEMS

RTEMS is designed to be easily ported to different processors and boards, thanks to a modular design, CPU libraries and Board Support Packages. Automatic Makefile generation tools, linker scripts and compiler specification files are available in order to precisely control the build process and target the operating systems to a particular platform [6].
There already exist ports to several architectures, including Intel, PowerPC, ARM, MIPS and SPARC in the `cpukit` package. Since ML-RSIM models superscalar RISC processors which utilize SPARC instruction set, we cross-compiled RTEMS to a SPARC target taking advantage of the ERC32 port realized by the European Space Agency, included in the RTEMS distribution.

C.1 CPU Porting

The European Space Agency's ERC32 is a multiprocessor computing core designed for embedded space applications. ERC32 features three SPARC V7 processors based on Cypress 7C601 and 7C602 with additional error-detection and recovery functions. The memory controller (MEC) implements system support functions such as address decoding, memory interface, DMA interface, UARTs, timers, interrupt control, write-protection, memory reconfiguration and error-detection. The core is designed to work at 25MHz, but using space qualified memories limits the system frequency

to around 15 MHz, resulting in a performance of 10 MIPS and 2 MFLOPS. ESA also provides development tools, such as the SPARC Instruction Simulator (SIS), VHDL models of hardware components and documentation [48, 9].

On the other hand, RISC processors simulated by ML-RSIM (as well as RSIM) borrow architecture concepts from MIPS R10000 but provide the SPARC version 8 32-bit instruction set [27, 40]. At the same time, processor state registers, control registers and trap handling are based on the 64-bit platform SPARC version 9 [41]. It turns out that ML-RSIM does not model any particular existing processor and RTEMS kernel had to be tuned to its unique architecture.

Critical portions of RTEMS code have been rewritten, migrating from the SPARCv7-compliant ERC32 to the SPARCv9-flavored ML-RSIM processor, namely [43, 44, 41]:

- Interrupt management
- Register window management
- Processor state register and control registers
- Virtual memory and TLB hardware support

C.1.1 Interrupt management

SPARCv9 introduced a new trap table with 8-instruction or 32-byte handlers. RTEMS trap table has also been adapted to meet the conventions of the simulator framework, in particular the need to map ad-hoc simulator traps (for statistics, logs and exception handling) besides software and hardware traps. SPARCv9 also introduced four or more level of traps and automatic saving of the trap state in register stacks (Trap State, Trap Type, Trap Program Counter, Trap Next Program Counter).

C.1.2 Register window management

SPARC processors feature the *register window architecture*, devised at the University of California at Berkeley in the early eighties. At any given time, an instruction can access 8 global registers and a *window* of 8 input, 8 local and 8 output registers. Each window *shares* its input and outputs registers with the two adjacent windows, whereas local registers are unique to each window. When a SAVE instruction is executed (for example after a CALL instruction), the active window switches to the next one and shared registers can be used for parameter passing avoiding expensive accesses to the stack in most cases. In order to provide for arbitrarily nested function calls with a finite set of registers, a window register exception management is necessary to flush and retrieve registers to the stack when there are no more available windows.
SPARCv9 introduced deep changes as of dealing with register windows: a new Current Window Pointer register has been introduced (previously the CWP was a field in the Processor State Register); the CANSAVE, CANRESTORE, OTHERWIN, WSTATE registers ensure validity and coherency of registers as opposed to previous versions' Window Invalid Mask register. It's worth noting that these changes effect only few, although critical, portions of RTEMS's kernel, in particular window overflow/underflow (also known as spill/fill), context switching, system calls, trap handlers and interrupts' entry and exit code.

C.1.3 Processor state register and control registers

In SPARCv9 architecture Processor State Register, Window Invalid Mask and Trap Base Register have been deleted. The following SPARCv8's fields in Processor State Register have become separate registers: Processor Interrupt Level register, Current Window Pointer register, Trap Type register, Trap Base Address register, Version register, Condition Codes Register. Other changes concern floating-point unit, alternate space access and enhanced memory models.
See [44, Appendix K] for details. ML-RSIM defines its own control regi-

ster PSTATE to define privileged or user mode, enable and disable traps, manage address translation and Transaction Look-aside Buffers (TLB).

C.1.4 Virtual memory and Transaction Look-aside Buffer hardware support

RTEMS for SPARC does not support virtual memory and makes use of a flat physical memory address space. ML-RSIM's processor model features data and instruction TLB enable bits in the processor state register, which have to be reset. A glitch in MLRSIM implementation of TLBs and address translation causes the data TLB enable bit to be set after traps and interrupts, so appropriate privileged code has been added to interrupt and system calls handlers to make sure that virtual memory is always disabled.

C.2 Board Support Package Porting

RTEMS has been designed to be ported across different hardware configurations maximizing the amount of software that can be reused. A relatively small number of source files containing board and peripherals dependent code, such as device drivers, linker and compiler scripts, shared memory support driver, must be either developed from scratch or fine tuned to fit particular hardware specifications. Developers can take advantage of an ample library of drivers and board components, available respectively in `c/src/lib/libchip` and `c/src/lib/libbsp` packages.
In our case, the target hardware platform is the model simulated by ML-RSIM. We worked on ML-RSIM source code to provide RTEMS with the correct memory and I/O address spaces and suitable data structure within the simulator engine.

C.2.1 Device drivers

ML-RSIM provides a memory-mapped input-output model. We embedded a UART device in ML-RSIM in order to support communication bet-

ween the simulator engine and the operating system kernel avoiding proprietary simulation traps and function calls.
According to the specifications of our UART device model, we developed a device driver for RTEMS and included it in the board support package targeted to ESA's ERC32 project.

Ringraziamenti

Vorrei ringranziare il Prof. Luca Benini ed il Prof. Giovanni De Micheli per avermi offerto l'opportunità di svolgere la tesi presso il Computer Systems Lab a Stanford University; un sincero ringraziamento va al CAD Group ed in particolare a Terry Tao Ye, Takashi "Unbelievable" Yokoi, Armita Peymandoust ed Evelyn Ubhoff per gli stimoli e i consigli nell'ambito del progetto di ricerca e soprattutto per la loro amicizia e aiuto durante la mia permanenza. Grazie al Prof. Lambert Schaelicke di University of Notre Dame per il suo supporto tempestivo e preciso durante lo sviluppo del simulatore.

Ringrazio i miei colleghi presso L'Università di Bologna Gianluca Biccari, Matteo Dall'Osso, Luca Giovannini, Francesco Poletti per la loro disponibilità al confronto e alla collaborazione durante il lavoro di tesi. Un grazie sentito va inoltre a Marco Dozza, Carlo Chiesa, Davide Castaldini, Giampalo D'Ambra per l'aiuto e l'esempio nell'intero percorso dell'Università.

Grazie a Luca "Kappa" Capelletti, Gianluca Guasti, Elena Dall'Agata, Rita Grossi, Matteo & Dario Gasparri, Alessio Armaroli, Edoardo Caldesi, Matteo "Tim" Selleri per la loro amicizia e appoggio. Grazie a Matteo "Monty" Monticelli, Carlo Gubellini, Rita Cevenini, Silvia Simoni, Francesca Melega, Gaia Giovannini, Gabriele Gelati, Andrea Gasparini, Stefano Zacchiroli e al gruppo Villanova I.

Grazie ai *Fuori le mura* per il divertimento e le serate a suonare: Federi-

co Ecchia, Pierre, Matteo "Facco" Facchini, Viaggi. Grazie a Tony Montano, Philip Diderichsen e Bevis Metcalfe per la musica improvvisata a San Diego. Grazie al coro *S.Ambrose*.

Grazie ai roommates dell'apt. 521 all'international house presso University of California, San Diego: Matt David, Jesse Coward, Irvin Teh, Patrick Alexander. Grazie a Merete Jacobsen, Santi Furnari, Alessio Benaglio, Francesco Pepe, Gladys Evangelista, Ilka Beinhorn, Sarah Wurmnest, Carrie Cheng, Joana Pinto, Jean-Charles Couteau, Mathias Palmqvist, Stephan Duetzmann, Eugenie Schonek, Lorenzo Cappellari, Marianna Bellon, Loxa Tamayo Marquez, Matin Sarfaraz, Michael D'Ortenzio, Rachel Adams.

Un grazie a Lynette, JB & Andy Skelton per avermi ospitato a Palo Alto; a Elena Pacchierotti, Claudio Storelli, Valeria Bertacco, Fabrizio "Bizio" Agosta, Giovanni Segni, Mauro Sylos Labini, Roberto "Dubbo" Dugnani, per le spaghettate e le serate italiane; a Farrah e la sua famiglia per la cena di Natale a San José; a Robert "RoJo" Rodger, Beatrice "Bici" Barbareschi & *The TenRz* per i *rally* della Leland Stanford Junior University Marching Band. Grazie a Jasmine Rogers, Chris & Paul, Alex, Micole, Hiroko, Kana, `Shahanas79`, Liatte.