POLITECNICO DI MILANO

Facoltà di Ingegneria dell'Informazione

Corso di Laurea in Ingegneria Informatica

UNA STRATEGIA DI MAPPING DINAMICO PER APPLICAZIONI SU PIATTAFORME MANY CORE

Relatore: Prof.ssa Donatella SCIUTO

Correlatore: Ing. Antonio MIELE

Correlatore: Ing. Vincenzo RANA

Tesi di Laurea di:

Luca CERRI

Matricola n. 739369

Anno Accademico 2010–2011

a Papà, Mamma, Sara e Nonna Ada

Preghiera dell'Alpino

Sulle nude rocce, sui perenni ghiacciai, su ogni balza delle Alpi ove la Provvidenza ci ha posto a baluardo fedele delle nostre contrade, noi, purificati dal dovere pericolosamente compiuto, eleviamo l'animo a Te, o Signore, che proteggi le nostre mamme, le nostre spose, i nostri figli e fratelli lontani e ci aiuti a essere degni della gloria dei nostri avi.

Dio onnipotente, che governi tutti gli elementi, salva noi, armati come siamo di fede e di amore. Salvaci dal gelo implacabile, dai vortici della tormenta, dall'impeto della valanga. Fa' che il nostro piede posi sicuro sulle creste vertiginose, sulle diritte pareti, oltre i crepacci insidiosi: rendici forti a difesa della nostria Patria, della nostra Bandiera.

E tu, Madre di Dio, candida più della neve, Tu che hai conosciuto e raccolto ogni sofferenza ed ogni sacrificio di tutti gli Alpini caduti, Tu che conosci e raccogli ogni anelito ed ogni speranza di tutti gli Alpini vivi ed in armi, benedici e proteggi i nostri Reggimenti e Battaglioni.
Amen.

Ringraziamenti

Ed eccomi alla parte in assoluto più difficile da scrivere, i ringraziamenti. È difficile ricordare tutte le persone che mi sono state vicine in questi sei anni di "avventura politecnica"quindi, inevitabilmente, mi dimenticheroò qualcuno Spero non se la prenda troppo! Inizio con un piccolo *disclaimer*: l'ordine dei ringraziamenti è puramente casuale!

Prima di tutti però, mi sembra doveroso ringraziare *Mamma, Papà, Sara* e *Nonna*, che mi sono stati sempre vicini durante questi anni, soprattutto nei momenti di sconforto e disperazione post-esami andati male.

Un ringraziamento particolare va alle Professoresse *Sciuto* e *Bolchini*, ad *Antonio* e *Vincenzo* che mi hanno assistito durante lo svolgimento di tutto il lavoro fatto per realizzare questa tesi.

A *Arturo* e *Guido*, inseparabili compagni di avventure praticamente da sempre.

A *Giovanni* (ma quando mai ti ho chiamato così?), alle sue uscite inopportune e al suo sogno che non si realizzerà mai!

A *Daniele*, grande inventore di suffissi al mio nome durante le partite di pallavolo.

A *Federica, Marzia, Miriam, Michel, Jack, Marco (Cardo),Mario, Francesca, Giulia, Roberta, Rosi, Marco (Capo), Andrea (Maggio), Gabriele (Giussa), Daniela, Sofia e Marta* per le serate insieme e perchè sopportano ancora le mie frasi "pittoresche ".

A *Alessadra, Alessandra, Dario, Giorgio, Donato e Andrea* grandi compagni di uscite durante i primi anni di università.

A *Carlotta, Selenia, Daria, Matteo, Federico, Alessandro e Dario* per le tre settimane stupende a La Thuile e per avermi fatto capire che era ora di cambiare un po' le cose.

Ai professori *Lomazzi, Macchi, Di Napoli, Pagani, Gasparri, Maccarrone, Peroni, Bressan, De Marco e Scala* che hanno saputo coltivare e sviluppare la mia passione e hanno reso i primi di anni università un po' meno difficili grazie ai loro ottimi insegnamenti.

A *Carlo, Fabio, Claudio, Alberto e Alessandro* per aver permesso di trasformare la mia passione in lavoro.

Al *Gruppo Alpini di Legnano* e alle loro feste estive fatte immancabilmente il giorno prima di un esame. Ancora non mi rendo conto di come abbia fatto a passare quegli esami.

A tutti coloro che hanno creduto in me e mi hanno accompagnato in questi sei anni (e anche a chi ha fatto finta).

Infine una maledizione a LaTeX, per gli innumerevoli pomeriggi persi nel-

l'impaginazione della tesi. Penso che il terrore di riuscire a far stare una tabella in una pagina mi accompagnerà per molto tempo! :)

Luca

Indice

Elenco delle figure

Elenco delle tabelle

Elenco dei listati

Elenco delle abbreviazioni

BN Best Neighbor

DMA Direct Memory Access

EM ElectroMigration

FF First Free

FIFO First In First Out

FPU Floating Point Unit

GPU Graphics Processing Unit

LEC-DN Low Energy Consumption - Dependences Neighborhood

MAC Minimum Average Channel load

MIMD Multiple Instruction Multiple Data

MMC Minimum Maximum Channel load

MPSoC Multi-Processor SoC

MPU Memory Protection Unit

MTTF Mean Time To Failure

NBTI Negative Bias Temperature Instability

NN Nearest Neighbor

NoC Network-on-Chip

PE Processing Element

PL Path Load

QoS Quality of Service

RW Random Walk

SIMD Single Instruction Multiple Data

SM Stress Migration

SoC System-on-Chip

SPMD Single Program Multiple Data

TDDB Time-Dependent Dielectric Breakdown

TLM Transaction Level Modeling

VLIW Very Long Instruction Word

XML eXtensible Markup Language

Sommario

Negli ultimi anni, l'evoluzione tecnologica ha permesso la creazione di architetture fortemente parallele e, di conseguenza, lo sviluppo di software che possa sfruttare questa caratteristica. Tali architetture, denominate Multi-Processor SoC (MPSoC), hanno reso possibile l'esecuzione di applicazioni secondo il modello Multiple Instruction Multiple Data (MIMD) permettendo l'esecuzione di più applicazioni parallele contemporaneamente. Appartiene a questa categoria la piattaforma p2012 di S.T. Microelectronics, utilizzata come piattaforma di riferimento per questo lavoro di tesi.

L'introduzione di tali architetture ha reso necessario lo sviluppo di adeguate strategie di *mapping* delle applicazioni sulle risorse messe a disposizione dai MPSoC in quanto, le soluzioni esistenti, risultano inadeguate a causa del limitato modello di applicazione adottato.

Lo scopo del lavoro presentato in questa tesi è stato quello di definire un modello di applicazione ed architettura, sulla base del quale elaborare una strategia di *mapping* delle applicazioni adatta ad un ambito di calcolo intensivo. L'elaborazione di tale strategia ha considerato due obiettivi principali: la minimizzazione dei tempi di esecuzione delle applicazioni e l'utilizzo uniforme delle risorse, al fine di prevenire guasti dovuti al fenomeno dell'invecchiamento precoce dei componenti causato da un utilizzo sbilanciato degli stessi. Da un'analisi della letteratura infatti è emerso che nessuna strategia di mapping analizzata prende in considerazioni l'importante tema dell'affidabilità.

Al fine di verificare il comportamento della strategia di *mapping* elaborata, è stato implementato un simulatore dell'architettura p2012, che ha inoltre permesso il confronto con altre strategie di *mapping* presenti in letteratura e opportunamente adeguate al modello di applicazione ed architettura considerato. I risultati emersi dai confronti mostrano come la strategia di *mapping* presentata in questo lavoro di tesi raggiunga gli obiettivi preposti riuscendo a minimizzare il tempo di esecuzione delle applicazioni e a rendere uniforme l'utilizzo delle risorse presenti sulla piattaforma.

Capitolo 1

Introduzione

Negli ultimi anni, l'industria dei semiconduttori si è notevolmente evoluta. La crescente complessità delle architetture di calcolo ha infatti frenato, sino ad arrestare, la "corsa al MegaHerts"spingendo i grandi produttori di semiconduttori a sviluppare soluzioni *multicore* per continuare ad incrementare le prestazioni delle soluzioni basate sulle loro architetture.

Contemporaneamente allo sviluppo delle CPU *multicore* si è assistito alla esplosione delle Graphics Processing Unit (GPU), che da semplici acceleratori per grafica tridimensionale si sono evoluti in vere e proprie unità programmabili utilizzabili allo scopo di effettuare grandi quantità di calcoli in virgola mobile. Al giorno d'oggi infatti, risultano composte da centinaia di processori vettoriali operanti in parallelo e capaci di eseguire contemporaneamente lo stesso programma su grandi quantità di dati (modello *Single Instruction Multiple Data*, SIMD).

Il contemporaneo affinamento dei processi produttivi ha inoltre reso possibile l'integrazione di diversi tipi di unità di calcolo su un unico *die* dando vita ai System-on-Chip (SoC), architetture eterogenee contenenti diversi tipi di circuiti che spaziano dai processori *general purpose* agli acceleratori hardware dedicati. L'ulteriore affinamento delle tecniche di produzione (Intel sta attualmente utilizzando una tecnologia produttiva a 32nm [5]

ed è prossima a far debuttare processori prodotti con tecnologia a 22nm), l'esplosione del settore mobile (*smartphone*, *tablets* e sistemi di gioco portatili) e la crescente domanda di capacità di calcolo, hanno dato impulso inoltre allo sviluppo delle architetture di tipo Multi-Processor SoC (MPSoC), architetture fortemente parallele capaci di eseguire contemporaneamente, a differenza delle architettura GPU, programmi diversi lavorando su dati diversi (modello *Multiple Instruction Multiple Data*, MIMD). L'introduzione di siffatte architetture parallele (GPU, SoC, MPSoC, CPU *multicore*) ha creato la necessità di sviluppare nuovi algoritmi di *scheduling* e *mapping* delle applicazioni sulle numerose unità di calcolo che compongono questo tipo di architetture. Gli algoritmi di *schduling* e *mapping* classici, infatti, non sono in grado di gestire architetture fortemente parallele in quanto non pensati per gestire in modo efficiente la presenza di più unità di calcolo ottimizzando l'utilizzo delle risorse a disposizione.

L'esigenza di implementare tali algoritmi in modo efficiente si è inoltre manifestata nei SoC e nei MPSoC che tipicamente sono gestiti da versioni ridotte ed ottimizzate di sistemi operativi e che, dovendo solitamente operare in ambienti *real-time* o ad alta intensità di calcolo, richiedono che l'*overhead* dovuto alle operazioni di *mapping delle applicazioni sia minimo*.

In tali architetture, anche la comunicazione tra i vari elementi risulta cruciale e, per ottimizzare tale aspetto, sono sempre più comuni dispositivi MPSoC basati su Network-on-Chip (NoC), ovvero circuiti nei quali i componenti sono connessi e comunicano tra loro utilizzando un'infrastruttura analoga alle reti di calcolatori. L'elevata densità dei componenti presenti in queste architetture pone inoltre in primo piano il problema della prevenzione e della gestione dei guasti, sia transitori che permanenti, dovuti ai numerosi effetti sia elettromagnetici che termici, che possono accorciare notevolmente il tempo di vita degli elementi componenti il MPSoC qualora essi non vengano tenuti in considerazione utilizzando, in modo sbilancia-

to, le risorse ivi presenti. Risulta indispensabile quindi tenere in considerazione anche il fenomeno dell'invecchiamento dei componenti dovuto allo stress a cui tali sistemi vengono sottoposti.

1.1 Ambito di lavoro e obiettivi

Nonostante i sistemi di tipo MPSoC siano nati con lo scopo di riunire in un unica architettura diversi tipi di Processing Element (PE) per ottimizzare sia consumi che costi dei dispositivi mobili, sono sempre più frequenti le architetture di questo tipo sviluppate per eseguire contemporaneamente più applicazioni con un modello di esecuzione fortemente parallelo, come ad esempio applicazioni scientifiche o di elaborazione intensiva dei dati. Il MPSoC p2012 di S.T. Microelectronics, utilizzato come base in questo lavoro di tesi, appartiene proprio a questa categoria. La particolarità di questo MPSoC risiede nella sua struttura su due livelli: al posto di normali PE esso infatti contiene dei *cluster*, a loro volta contenenti diverse PE [6]. Un tipo di applicazione ed un'architettura così complesse richiedono quindi lo studio di adeguati algoritmi di assegnamento (*mapping*) che tengano in considerazione numerosi aspetti sia dal punto di vista delle prestazioni che dal punto di vista della tolleranza ai guasti. Essendo dunque un'architettura sviluppata recentemente e molto complessa non si rende necessaria la sola elaborazione di adeguati algoritmi di assegnamento, ma anche la creazione di un modello base sia dell'architettura che delle applicazioni che su di essa possono essere eseguite. Deve essere inoltre tenuta in considerazione l'importante tematica della tolleranza ai guasti prevedendo meccanismi (sia *software* che *hardware*) che permettano all'architettura di continuare a funzionare anche in presenza di guasti sia transitori che permanenti dovuti, in particolare, al fenomeno dell'invecchiamento precoce dei componenti. La definizione di adeguati algoritmi di assegnamento può dunque essere

effettuata solo dopo aver affrontato e risolto le problematiche esposte e rappresenta solo una parte del vasto ambito di ricerca in cui questo lavoro di tesi si pone. Il numero di problematiche da affrontare risulta quindi troppo elevato affinchè esse possano essere risolte con un unico lavoro; per questo motivo il lavoro svolto in questa tesi si è focalizzato sui seguenti obiettivi:

- Analisi dell'architettura p2012 ed elaborazione del relativo modello di esecuzione;
- Elaborazione di un modello di applicazione e del relativo modello di esecuzione basati sul modello dell'architettura;
- Elaborazione di una strategia base di assegnamento delle applicazioni che permetta di minimizzare i tempi di esecuzione delle applicazioni evitando l'invecchiamento precoce dei componenti presenti sull'MPSoC che possa così premettere l'esecuzione affidabile delle stesse.

Rispetto alle problematiche evidenziate, è stato tralasciato il trattamento del rilevamento e della gestione dei guasti assumendo i componenti presenti all'interno del MPSoC p2012 dotati di tali meccanismi.
L'elaborazione della strategia di assegnamento si è quindi concentrata sulla creazione di un algoritmo dinamico, basato sui modelli elaborati, che abbia la capacità di eseguire le applicazioni in modo adeguato sulla base delle risorse disponibili all'interno del MPSoC considerando eventuali *cluster* inutilizzabili e limitando il più possibile il fenomeno dell'invecchiamento precoce dei componenti dovuto a un utilizzo sbilanciato degli stessi.

1.2 Struttura del lavoro

L'esposizione del lavoro effettuato è strutturata nel seguente modo: Nel **Capitolo 2** viene descritto lo stato dell'arte per quanto riguarda gli algo-

ritmi di *mapping* dinamico su MPSoC basati su NoC, introducendo gli algoritmi con cui l'euristica sviluppata in questo lavoro di tesi è stata confrontata. Viene inoltre introdotto quanto presente in letteratura per quanto riguarda la modellazione dei guasti dovuti a Negative Bias Temperature Instability (NBTI) che portano al fenomeno dell'invecchiamento precoce dei componenti. Il **Capitolo 3** contiene invece l'analisi, la descrizione e la modellazione dell'architettura p2012 di ST e delle applicazioni considerate. Per quanto riguarda l'architettura p2012 viene posta particolare enfasi sulla sua composizione e sul funzionamento della stessa andando ad analizzare in particolare come le applicazioni possono essere eseguite su questa piattaforma. Il modello di applicazione utilizzato discende direttamente da questa analisi ed è stato elaborato in modo tale da essere coerente col modello di architettura elaborato. Nel **Capitolo 4** viene invece introdotto l'algoritmo di *mapping* dinamico implementato e la strategia su cui esso si basa, descrivendo nei dettagli le scelte fatte sia dal punto di vista dell'assegnazione delle applicazioni sulle unità processanti, sia dal punto di vista dell'affidabilità. Viene quindi introdotta la metrica utilizzata evidenziando le differenze che la distinguono da quanto presente in letteratura. Il **Capitolo 5** introduce tutto il lavoro svolto al fine di validare i modelli e la strategia di *mapping* elaborata. Esso pertanto introduce il simulatore sviluppato per poter verificare il funzionamento della srategia stessa ed i test che tramite esso sono stati effettuati al fine di verificare il comportamento della strategia stessa e di confrontarne le prestazioni con alcune tecniche presenti in letteratura. Chiude il capitolo l'analisi dei risultati ottenuti nei test. Nel **Capitolo 6** vengono quindi elaborate le conclusioni e introdotti i possibili sviluppi futuri al lavoro effettuato.

Chiude la trattazione del lavoro l'**Appendice A**, all'interno della quale vengono illustrati i dettagli di implementazione del simulatore dell'architettura utilizzato per effettuare i test sulla strategia di *mapping*.

Capitolo 2

Stato dell'arte

Questo capitolo introduce le tecniche di assegnazione (*mapping*) dinamica presenti in letteratura prese come riferimento durante lo sviluppo del lavoro. Non vengono prese in considerazione solo le tecniche con cui il lavoro è stato direttamente confrontato, ma viene fornita una descrizione più ampia di quanto presente in letteratura. Ulteriore scopo del capitolo è l'introduzione dei modelli di invecchiamento causati dai fenomeni di Negative Bias Temperature Instability (NBTI) fondamentali per l'introduzione della relativa metrica all'interno della soluzione sviluppata. Il capitolo è dunque articolato su due sezioni:

- Nella Sezione 2.1 vengono introdotti i lavori concernenti gli algoritmi di *mapping* dinamico Path Load, Lec-DN e Random Walk, utilizzati nel confronto con la soluzione proposta unitamente ad altri algoritmi di assegnazione dinamica presenti in letteratura, per fornire un quadro completo di quanto sviluppato dalla ricerca in questo campo negli ultimi anni;
- La Sezione 2.2 introduce invece i diversi modelli del fenomeno di invecchiamento dovuto a NBTI utilizzati in letteratura. Pur essendo modelli elaborati a livello elettronico, essi sono risultati adattabili al-

l'euristica sviluppata rendendo possibile l'introduzione, al suo interno, di metriche inerenti alla prevenzione di guasti sia permanenti che transitori, causati dal fenomeno dell'invecchiamento.

2.1 Algoritmi di Mapping

Negli ultimi anni, numerosi sono stati gli approcci al *mapping* dinamico di applicazioni su Multi-Processor SoC, (MPSoC) basati su Network-on-Chip (NoC). In questa sezione vengono analizzati alcuni di questi algoritmi con lo scopo di confrontarli rispetto agli obiettivi che il lavoro decritto in questa tesi si è posto. Tutti gli algoritmi analizzati utilizzano un modello di applicazione che utilizza un grafo aciclico orientato all'interno del quale i nodi rappresentano i processi di cui un'applicazione è composta e gli archi rappresentano le comunicazioni tra essi. La maggior parte di questi algoritmi inoltre utilizza un modello di esecuzione dell'applicazione di tipo master/slave, dove il processo iniziale rappresenta il processo master, contenente il codice dell'applicazione mentre i successivi processi vendono definiti come processi slave e rappresentano eventuali servizi che possono essere richiesti dal processo master. Al fine di costruire una classificazione degli algoritmi analizzati,viene introdotta la tassonomia utilizzata in [7] per classificare gli algoritmi considerati. Tale tassonomia è basata su quattro criteri:

- Momento di assegnazione dell'applicazione;
- Numero di processi assegnati per ciascun Processing Element (PE);
- Gestione del *dispatching*;
- Architettura del Multi-Processor SoC (MPSoC) considerato.

Per quanto riguarda il **Momento di assegnazione dell'applicazione**, sono possibili due classificazioni:

- *statico* o *offline*, utilizzato solitamente su sistemi che devono eseguire applicazioni predefinite in modo periodico, che permette l'applicazione di tecniche di assegnazione che conducono a soluzioni ottime;
- *dinamico* o *online*, utilizzato quando non è possibile prevedere l'arrivo delle applicazioni, in questo caso le euristiche adottate risultano più semplici in quanto l'*overhead* dovuto all'assegnazione da adottare deve essere minimo. L'assegnazione dinamica di un'applicazione può essere a sua volta effettuato in due diversi modi:
  - *riservando le risorse*, ovvero verificando che vi siano abbastanza risorse disponibili per poter eseguire l'applicazione;
  - *non-riservando le risorse*, ovvero assegnando solo il primo processo dell'applicazione e assegnare gli altri man mano che vengono richiesti.

Anche per quanto riguarda il **Numero di processi assegnati per ciascun PE** sono possibili due valori:

- *processo-singolo*, ovvero a un PE viene assegnato un unico processo;
- *processo-multiplo*, ovvero a un PE possono essere assegnate più processi.

La classificazione della **Gestione del *mapping*** distingue tra

- *approccio centralizzato*, quando vi è una specifica PE dedicata alla gestione del *mapping* delle applicazioni;
- *approccio distribuito*, quando il MPSoC viene diviso in regioni contenenti ciascuna un PE deputato alla gestione delle applicazioni (*dispatcher*).

Infine, i valori di classificazione possibile per l'**architettura del MPSoC considerato** sono i seguenti:

Tabella 2.1: Elenco dei lavori analizzati

Autore	**Mapping**	**Risorse**	**Processo**	**Architettura**	**Controllo**	**Metriche**
Smit [1]	Dinamico	Riservate	Processo singolo	Eterogenea	Centralizzato	Consumo energia, QoS richiesto dalle applicazioni
Wildermann [2]	Dinamico	Non riservate	Processo singolo	Omogenea	Centralizzato	*Deadline* delle Applicazioni, consumo di energia, latenza di comunicazione
Schranzhofer [8]	Dinamico	Riservate	Processo singolo	Omogenea	Centralizzato	Consumo di energia
Singh [9][10]	Dinamico	Non riservate	Processo multiplo	Eterogenea	Centralizzato	Volume di comunicazione, consumo di energia
Carvalho [11][12][13]	Dinamico	Non riservate	Processo singolo	Eterogenea	Centralizzato	Volume di comunicazione
Mandelli [14][7]	Dinamico	Non riservate	Processo singolo	Omogenea	Centralizzato	Consumo di energia, volume di comunicazione
Weichslgartner [3]	Dinamico	Non riservate	Processo singolo	Omogenea	Distribuito	Volume di comunicazione, carico sulla NoC
Lavoro proposto	**Dinamico**	**Riservate**	**Processo multiplo**	**Omogenea**	**Centralizzato**	**Uniformità d' uso dei PE**

- *omogenea*, quando tutti i PE sono identici;
- *eterogenea*, quanto nel MPSoC sono presenti diversi tipi di PE.

L'algoritmo di assegnazione sviluppato in questo lavoro di tesi, secondo la tassonomia appena introdotta, lavora su un MPSoC **omogeneo**, in quanto tutti i *cluster* dell'architettura "p2012 "sono identici (vedi Sezione 3.2) e in cui il *dispatching* viene gestito con **approccio centralizzato**. L'algoritmo inoltre può essere classificato come **dinamico con preservazione delle risorse** e quindi necessariamente con assegnamento a **processo multiplo**.

In Tabella 2.1, viene riportata la tassonomia di tutti gli algoritmi di dispatching analizzati in questa sezione.

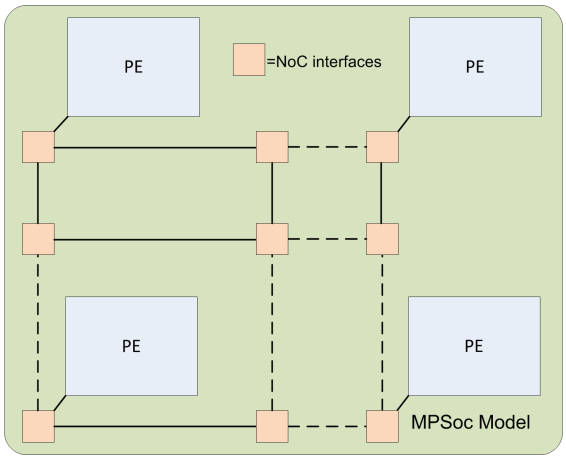


Figura 2.1: Modello di MPSoC utilizzato in letteratura

Tutti gli algoritmi analizzati in questo capitolo condividono il medesimo modello di MPSoC, che è riportato in Figura 2.1. Un MPSoC viene visto come un insieme di PE collegati tra di loro attraverso una NoC. I PE possono a loro volta essere identici tra loro oppure diversi.

2.1.1 Algoritmi basilari

Carvalho, in [12] e [13], introduce, oltre a Path Load, quattro strategie di assegnazione dinamica che fungono da base per algoritmi più evoluti come Path Load stesso o Random Walk. Le strategie introdotte in [12] e [13] sono le seguenti:

- First Free (FF);
- Nearest Neighbor (NN);
- Minimum Maximum Channel load (MMC);
- Minimum Average Channel load (MAC).

Tutte queste strategie utilizzano un modello di applicazione di tipo *master/slave* a controllo centralizzato.

First Free è la più semplice strategia di **mapping** adottabile. Essa consiste nell'assegnare l'applicazione da eseguire al primo PE libero trovato durante la scansione del dispositivo.

Nearest Neighbor è una semplice variazione di FF, che assegna l'applicazione da eseguire al PE libero più vicino, in termini di *hops*.

Minimum Maximum Channel load è un tipo di euristica più evoluta rispetto alle precedenti e ha lo scopo di minimizzare l'occupazione dei nodi della NoC. Per ottenere questo obiettivo essa lavora valutando il costo di ogni possibile assegnamento dell'applicazione da eseguire secondo la formula riportata nell'Equazione 2.1.

$$cost_k = \max_{l(i,j)}(rate_{l(i,j)}) \tag{2.1}$$

con:

i e j coordinate del nodo considerato;

$l(i,j)$ link considerato;

$rate_{l(i,j)}$ occupazione del link considerato.

L'euristica **Minimum Average Channel load** invece, è stata progettata per ridurre l'occupazione media dei nodi della NoC. Essa è stata derivata direttamente dall'euristica MMC andandone a modificare la funzione di costo come riportato nell'Equazione 2.2.

$$cost_k = \operatorname{avg}_{l(i,j)}(rate_{l(i,j)}) \tag{2.2}$$

I risultati riportati in [13] mostrano come, tra le quattro euristiche riportate, NN risulti essere la migliore. I risultati ottenuti con la simulazione di 99 applicazioni utilizzanti come base di confronto i tempi ottenuti con la strategia FF mostrano infatti che NN esegue la simulazione impiegando l'83% del tempo impiegato da FF, mentre MMC completa la simulazione impiegando il 98% del tempo e MAC invece porta a compimento la simulazione impiegando il 90% del tempo.

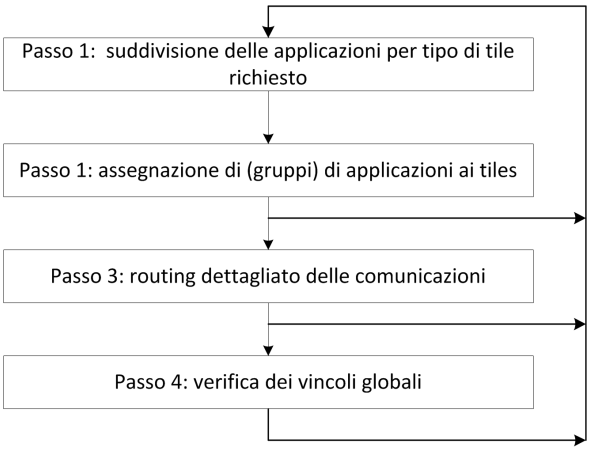


Figura 2.2: Passi dell'algoritmo di mapping sviluppato in [1]

2.1.2 Approccio Gerarchico

Smit, in [1], presenta un algoritmo di *mapping* dinamico di un singolo processo volto a minimizzare il consumo energetico del MPSoC. Il problema del mapping delle applicazioni sui PE viene formulato sotto forma di problema GAP (General Assignment Problem) che essendo NP-Completo [15] viene affrontato utilizzando un approccio iterativo basato su più livelli di elaborazione. Tali livelli sono illustrati in Figura 2.2. L'architettura utilizzata dall'algoritmo è analoga a quella illustata in Figura 2.1 con la differenza che i PE prendono il nome di *Tile*. Il funzionamento dell'algoritmo è dunque molto semplice:

- nel primo passo le applicazioni vengono divise a seconda del tipo di *tile* richiesto per l'esecuzione;
- nel secondo passo, invece, si assegnano i *tile* fisici alle applicazioni. Se l'assegnamento non va a buon fine, si ritorna al passo precedente modificando la divisione delle applicazioni;
- nel terzo passo viene applicato il *routing* delle comunicazioni in base all'assegnamento effettuato al passo precedente. Nel caso vengano

rilevate situazioni di eccessivo carico sui link coinvolti, l'algoritmo torna ai passi precedenti modificando l'assegnamento o la divisione delle applicazioni al fine di elaborare una nuova soluzione che permetta di ottenere un effettivo risparmio energetico riducendo il volume delle comunicazioni;

- infine, nel quarto passo, vengono verificati tutti i vincoli globali (ad esempio i vincoli temporali). Anche in questo caso, se qualche vincolo non viene rispettato, è possibile ripercorrere i passi precedenti alla ricerca di una soluzione migliore.

I risultati illustrati dall'articolo mostrano come l'approccio utilizzato porti alla soluzione ottima con solo tre iterazioni dell'algoritmo, pari a circa 2-3 ms di esecuzione su un processore ARM operante alla frequenza di 100Mhz, rispetto alle 10 ore di simulazione richieste da una ricerca esaustiva effettuata su un PC dotato di Pentium 4 a 2.4 GHz. Il lavoro svolto da Smit quindi, utilizza metriche e modello di applicazione sensibilmente diversi rispetto a quanto adottato in questo lavoro di tesi. Dal punto di vista delle applicazioni infatti, il modello adottato in [1] è sensibilmente diverso e più semplice rispetto a quanto utilizzato dal presente lavoro di tesi, andando quindi ad individuare un utilizzo dell'algoritmo in ambiti diversi da quelli ipotizzati per l'euristica sviluppata. Dal punto di vista delle metriche, invece, l'algoritmo presentato in [1] si occupa di ottimizzare il consumo energetico e di soddisfare eventuali requisiti di QoS, metriche che nel lavoro presentato in questa tesi non sono state prese in considerazione.

2.1.3 Approccio di Wildermann

In [2], **Wildermann** propone un algoritmo di assegnazione adattivo che opera su applicazioni modellate secondo il modello *master/slave*, che prevede un'applicazione composta da un processo *master* che gestisca l'ese-

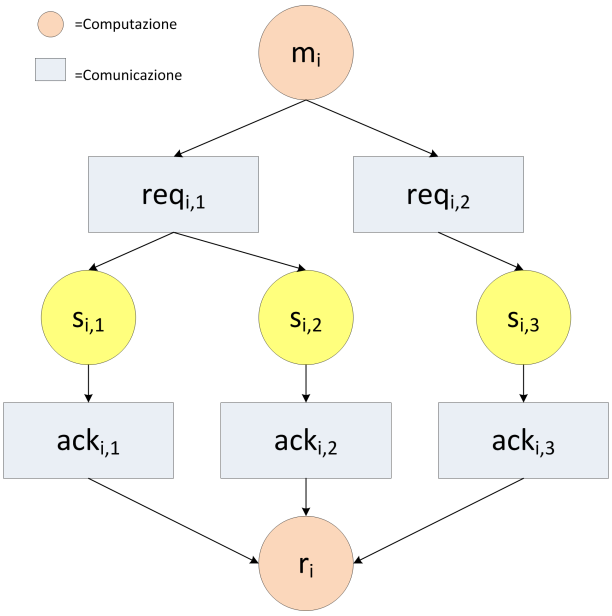


Figura 2.3: Task-graph utilizzato in [2]

cuzione dell'applicazione richiedendo l'allocazione dei processi slave al momento opportuno. L'idea di fondo di quanto presentato in [2] è quella di sviluppare un algoritmo che sia in grado di modificare il task-graph di un'applicazione qualora vi siano cambiamenti nei requisiti dell'applicazione non conosciuti a priori minimizzando volume di comunicazione e consumo energetico. Gli autori in [2] descrivono le applicazioni con un grafo diretto bipartito composto da due tipi di vertici: i vertici di computazione e i vertici di comunicazione, come mostrato in Figura 2.3.

I processi sono assegnati alle PE connesse alla NoC minimizzando il valore della funzione di costo riportata nell'Equazione 2.3.

$$\begin{aligned} cost_t(s_{i,j}) = &\alpha_1 \cdot W_t \cdot (s_{i,j}, PE_{x,y}) + \\ &\alpha_2 \cdot D_t \cdot (s_{i,j}, PE_{x,y}) + \\ &\alpha_3 \cdot N_t \cdot (s_{i,j}, PE_{x,y}) \end{aligned} \tag{2.3}$$

La funzione di costo utilizzata è una combinazione lineare di tre metri-

che:

- W_t rappresenta l'ultimo istante temporale in cui l'applicazione può essere eseguita;
- D_t rappresenta il costo della comunicazione tra i vari processi di cui è composta un'applicazione. Questa metrica viene utilizzata per valutare quindi il consumo energetico del MPSoC;
- N_t rappresenta la valutazione dei vicini in termini di risorse occupate da altri processi eventualmente in esecuzione su di essi.

α_1, α_2 eα_3 rappresentano i pesi dati alle tre metriche. L'algoritmo in [2] è stato verificato utilizzando i valori $\alpha_1 = 0.45$, $\alpha_2 = 0.45$ e$\alpha_3 = 0.1$ e andando ad eseguire applicazioni generate in modo casuale con comportamento anch'esso casuale. I risultati mostrano come l'approccio utilizzato risulti efficace, in termini di energia consumata e di latenza di comunicazione, sopratutto nel caso di applicazioni dal comportamento molto dinamico. Il punto debole dell'euristica però emerge nel caso in cui si abbia come requisito principale la minimizzazione della comunicazione tra due PE. In questo caso una politica di tipo NN risulta più efficace in quanto assegna il processo richiesto al PE libero più vicino all'unità sulla quale è in esecuzione il processo richiedente andando, di conseguenza, a limitare il percorso seguito dai dati scambiati dai due processi. Anche questa euristica si dimostra però inadatta agli scopi che questo lavoro di tesi si è prefisso. Questo a causa del modello di applicazione adottato che è di tipo master/slave. Dal punto di vista della metrica invece possono essere tratti spunti interessanti. Infatti, nonostante l'euristica tenda a minimizzare il consumo energetico, essa tiene in considerazione anche la latenza di comunicazione tra il processo master e i processi slave, metrica presente anche nel lavoro sviluppato in questa tesi.

2.1.4 Approccio di Schranzhofer

L'algoritmo proposto da **Schranzhofer** in [8], invece, ha come obiettivo la minimizzazione dell'energia consumata durante l'esecuzione di applicazioni su sistemi MPSoC. Al fine di raggiungere l'obiettivo posto, in [8] il modello di MPSoC utilizzato (vedi Figura 2.1) viene arricchito con informazioni riguardanti il consumo di energia statico e dinamico di ciascun PE. In particolare il consumo **statico** viene definito come il consumo di un PE quando esso è acceso mentre il consumo **dinamico** viene definito come il consumo energetico che si osserva durante la computazione e quindi l'utilizzo del PE al massimo delle potenzialità. Anche il modello di applicazione adottato in questo lavoro è quindi costruito per contenere informazioni sul consumo di energia dei processi contenuti in esse. Per ogni applicazione inoltre vengono elaborati possibili modi di esecuzione, dei quali uno soltanto alla volta potrà risultare attivo. Le applicazioni inoltre possono avere processi in comune. Vengono quindi elaborati degli **scenari** che rappresentano combinazioni di modi di esecuzione delle varie applicazioni. Questi scenari vengono rappresentati come prodotto cartesiano dei grafi dei modi di esecuzione delle applicazioni e rappresentano le entità sulle quali andrà eseguito l'algoritmo di *mapping*. Questo algoritmo di *mapping* pertanto richiede una fase di pre-calcolo nella quale tutti i possibili modi di elaborazione e tutti i possibili scenari vanno elaborati richiedendo quindi la conoscenza delle applicazioni che saranno eseguite dal MPSoC. Un esempio di scenario è illustrato in figura 2.4

Una volta calcolati i possibili scenari, vengono anche pre-calcolati dei possibili assegnamenti delle applicazioni sul MPSoC andando a utilizzare le informazioni sul consumo energetico contenute sia nel modello del MPSoC sia nel *task-graph* delle applicazioni. I *template* ottenuti saranno quindi disponibili all'algoritmo di assegnazione che, osservando il comportamento del sistema, andrà di volta in volta a scegliere il *template* adatto alla

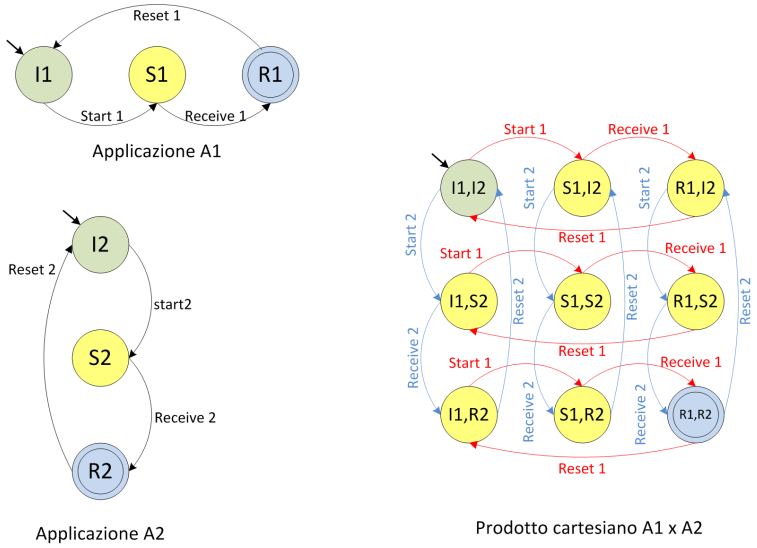


Figura 2.4: Task-graph di due applicazioni e relativo task-graph intersezione

situazione osservata. I risultati pubblicati mostrano come effettivamente il pre-calcolo dei template porti ad un risparmio di energia variabile tra il 32% e il 45% a seconda dell'ambito di applicazione in cui l'algoritmo viene applicato. Questo risparmio però è ottenuto grazie a una pesante fase di pre-calcolo che, oltre a richiedere la conoscenza del MPSoC, richiede anche la conoscenza di tutte le applicazioni che da esso verranno eseguite, che rappresenta un forte limite di questo approccio.

L'adozione del medesimo modello di applicazione delle soluzioni precedenti e il ricorso a una pesante fase di pre-calcolo rendono inadatta questa euristica al confronto con quanto sviluppato in questo lavoro di tesi a causa della richiesta della conoscenza di tutte le applicazioni che dovranno essere eseguite, informazione che non è disponibile nel lavoro oggetto di questa tesi in quanto l'euristica sviluppata è stata creata per gestire applicazioni non conosciute a priori inibendo così la possibilità di effettuare qualsiasi tipo di pre-calcolo.

2.1.5 Approccio Path Load (PL)

L'algoritmo Path Load viene presentato da **Carvalho et Al.** in [11], [12] e [13], insieme ad altre quattro tecniche, ovvero NN, Best Neighbor (BN), MMC e MAC. Il modello di MPSoC utilizzato è quello base, illustrato in Figura 2.1. L'algoritmo lavora su MPSoC eterogenei, è dinamico e lavora senza riservare le risorse allocando un processo alla volta per ogni PE. Lo scopo dell'euristica è quello di minimizzare il volume di traffico sulla NoC. Affinchè l'euristica possa essere implementata correttamente, il modello di applicazione deve contenere informazioni sul traffico scambiato tramite NoC da ogni singolo processo di cui essa si compone. Per questo motivo ogni applicazione viene modellata, come negli altri approcci, come un grafo diretto aciclico composto da processi. Ogni processo contiene informazioni su tempo di esecuzione, su tipo di esecuzione (software o hardware) e su volume di dati scambiati tramite NoC. Il modello di applicazione inoltre implementa il paradigma master/slave definendo, per ogni processo, una lista di altri processi che da esso possono essere richiamati. Il processo chiamante prende il nome di processo master mentre i processi richiamati vengono denominati processi slave.

Per funzionare correttamente, l'algoritmo necessita di una fase di preassegnamento dei processi iniziali che, come evidenziato in [13], influenza significativamente le performance della soluzione e deve essere affrontato con cautela. Il pre-assegnamento del processo iniziale può essere effettuato a run-time sulla base dello stato del MPSoC. Una volta effettuato il preassegnamento, Path Load viene eseguito per allocare i processi richiesti di volta in volta dal processo iniziale o da uno dei suoi processi slave. Grazie alle informazioni sul volume delle comunicazioni contenute in ciascun processo, l'euristica Path Load (PL) può essere definita come la minimizzazione del costo della comunicazione tra il processo master, allocato sul PE_i, e ogni possibile allocazione su un qualunque PE_j del processo slave, dove

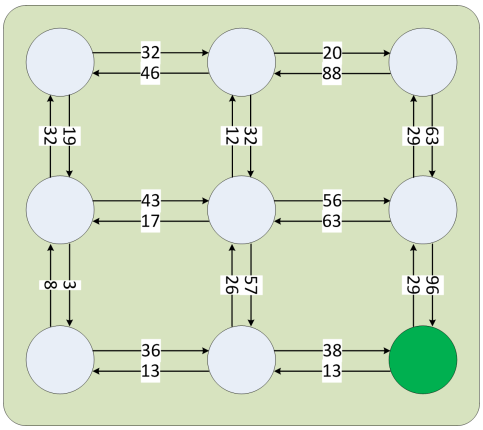


Figura 2.5: Modello di MPSoC utilizzato durante il calcolo di PL

i e j rappresentano l'identificativo del PE all'interno dell'architettura.

$$cost_k = \sum rate_{c(i,j)} + \sum rate_{c(j,i)} \tag{2.4}$$

Il costo della comunicazione è definito con l'Equazione 2.4 ed è composto da due parti:

- il costo della comunicazione tra master e slave (prima parte dell' equazione);
- il costo della comunicazione tra slave e master (seconda parte dell' Equazione);

dove la funzione $rate_{c(i,j)}$ rappresenta il volume di occupazione del link della NoC dal PE_i al PE_j. Durante l'esecuzione dell'algoritmo quindi, il MPSoC viene modellato essenzialmente come illustrato in Figura 2.5 dove ogni PE coincide con un nodo e per ogni link vengono indicati i volumi di occupazione in entrambe le direzioni di comunicazione. Per validare la bontà dell'algoritmo, in [13] e [12] sono state eseguite applicazioni composte da 10 processi e l'euristica PL è stata confrontata con le euristiche NN, BN, MMC e MAC. Il risultato mostrato in [13] evidenzia l'efficacia della tecnica PL, che effettivamente riduce il volume di traffico sulla NoC, anche se

la tecnica NN rimane una buona soluzione solo di poco inferiore a PL. Anche in questo caso il limite principale che rende inadatta la tecnica di Path Load all'utilizzo sulla piattaforma considerata dal lavoro di tesi risiede nel modello di applicazione considerato. Path Load utilizza però una funzione di costo che considera come metrica il solo volume di informazioni scambiate, senza combinarla con altre metriche quali consumo di energia o QoS. Per questo motivo, l'algoritmo di Path Load è stato scelto per il confronto diretto con la soluzione sviluppata in questo lavoro di tesi.

2.1.6 Approccio di Singh

Singh, in [9] e [10], utilizza il modello base di MPSoC illustrato in figura 2.1 e un modello di applicazione basato sul paradigma master/slave e il cui task-graph contiene informazioni sul volume di comunicazione tra i vari processi appartenenti a un'applicazione. La strategia implementata, compie come primo passo l'allocazione dei processi iniziali (processo master) che, al momento della richiesta di processi aggiuntivi, attivano l'algoritmo di *mapping* dinamico. Una volta allocati i processi iniziali, il MPSoC viene suddiviso in ***cluster* virtuali**, composti da tutti i PE adiacenti al PE sul quale sono allocati i processi iniziali, come mostrato in Figura 2.6.

L'euristica, quindi, tenta di assegnare i processi richiesti da un processo master nel corrispondente cluster virtuale, adottando una precisa strategia. Nel lavoro analizzato, sono state implementate due strategie: **Nearest Neighbor (NN)** adeguatamente modificato per funzionare con i *cluster* virtuali e **PL** (vedi 2.1.5), anch'esso con le opportune modifiche. L'algoritmo è stato verificato effettuando test su un simulatore di MPSoC basato su una NoC di grandezza 8x8. I risultati mostrano come le versioni adattate delle due euristiche risultino migliori in termini di occupazione della NoC rispetto alle loro versioni "pure", questo grazie all'introduzione dei *cluster* virtuali. Dal punto di vista dei tempi di esecuzione, invece, essi risultano

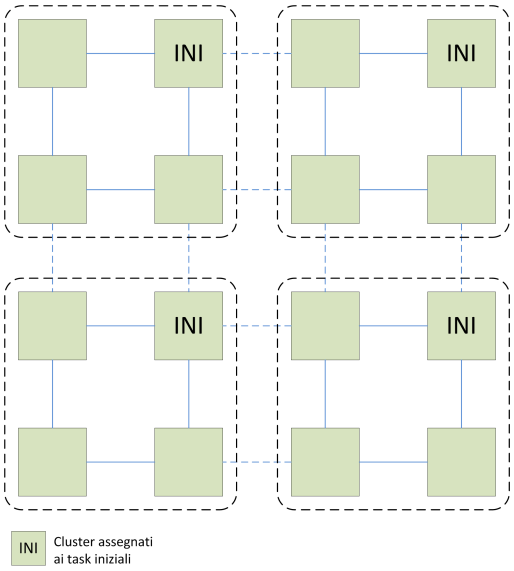


Figura 2.6: Suddivisione del MPSoC in *cluster* virtuali

sostanzialmente invariati. Nonostante il modello di applicazione utilizzato sia sempre quello master/slave, la tecnica utilizzata da questo algoritmo per minimizzare il volume dei dati scambiati tra i processi risulta interessante, sopratutto per quanto riguarda la definizione dei *cluster* virtuali a cui si è ispirata la scelta dell'identificazione delle aree implementata nell'euristica sviluppata in questo lavoro di tesi. Tuttavia la limitazione al solo modello di applicazione master/slave e l'inserimento della metrica di risparmio energetico rendono questa euristica non adatta agli scopi del lavoro e quindi ad un confronto diretto con l'euristica sviluppata in questo lavoro di tesi.

2.1.7 Approccio Low Energy Consumption - Dependences Neighborhood (LEC-DN)

L'algoritmo di *mapping* basato su euristica LEC-DN viene introdotto in [14] nella sua versione base e in [7] nella sua versione estesa con una fase di pre-assegnamento. Entrambe le versioni dell'algoritmo lavorano sul mo-

dello di MPSoC illustrato in Figura 2.1 e usano un modello di applicazione basato su quello utilizzato dall'algoritmo PL (vedi Sezione 2.1.5) ma che permette a ogni processo di comunicare con più processi diversi, andando così ad estendere il modello master/slave superando quindi questo tipo di organizzazione. L'euristica LEC-DN ha lo scopo di minimizzare il consumo di energia utilizzata e per ottenere lo scopo prefissato utilizza due funzioni di costo per selezionare il PE a cui assegnare un processo:

- la prossimitá del PE candidato, in numero di *hops*, rispetto agli altri PE assegnati all'applicazione;
- il volume di comunicazione tra processi.

La prima funzione di costo viene utilizzata come metrica principale, mentre la seconda funzione di costo viene utilizzata in presenza di un processo che necessita di comunicare con almeno altri due processi. In questo caso la seconda funzione di costo permette di assegnare il processo al PE più vicino a quello assegnato al processo con cui verrà scambiato un maggior volume di informazioni. Il funzionamento dell'euristica è semplice e può essere riassunto nel modo seguente:

- se il processo da allocare comunica con un solo altro processo, esso viene allocato utilizzando la tecnica NN;
- se invece il processo necessita di comunicare con almeno due altri processi, viene cercato un PE disponibile all'interno del rettangolo definito dalla posizione di questi processi, come illustrato in Figura 2.7. Per selezionare il PE a cui assegnare il processo, viene utilizzato il modello basato su consumo di energia e volume di dati scambiati proposto da **Hu et al.** in [16].

I risultati ottenuti in [14] mostrano come la versione base di LEC-DN migliori, seppur di poco, quanto ottenibile con le euristiche NN e BN, uti-

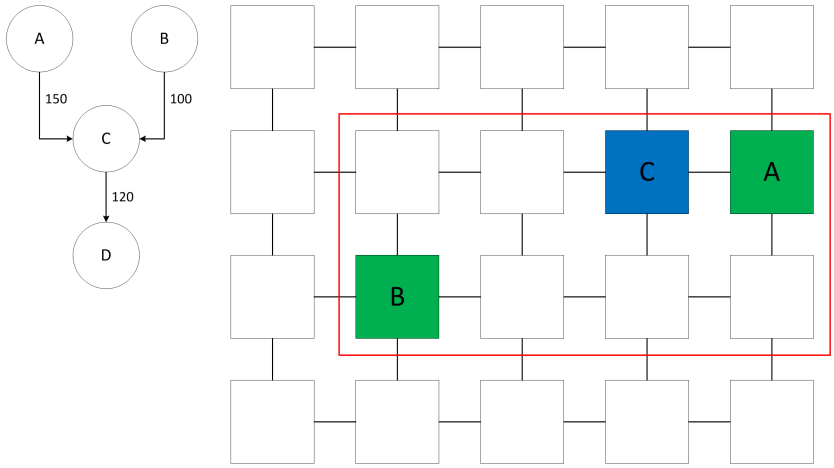


Figura 2.7: Rettangolo dei PE utilizzabili per l'assegnazione di un processo che comunica con almeno due processi

lizzando però un modello di applicazione che prevede che ogni processo possa comunicare con più processi.

La versione estesa di LEC-DN, descritta in [7], si differenzia dalla versione base per la presenza di una fase di pre-assegnazione implementata per poter utilizzare l'euristica nel caso di assegnazione multi-processo. La fase di pre-assegnazione viene introdotta con lo scopo di raggruppare set di processi sugli stessi PE. Per effettuare questa pre-assegnazione, viene valutato il volume di comunicazione che un processo sorgente ha con tutti i suoi processi destinazione. I processi destinazione della comunicazione vengono quindi pre-assegnati sullo stesso PE del processo sorgente se e solo se il processo destinazione non possiede a sua volta delle comunicazioni con volume di scambio maggiore rispetto a quello che ha con il processo sorgente. Una volta eseguita la pre-assegnazione, potrà essere eseguita la versione base di LEC-DN che andrà ad assegnare i processi non assegnati durante la fase precedente. I risultati esposti in [7] mostrano che la versione estesa di LEC-DN è sensibilmente migliore, in termini di consumo energetico, rispetto alla versione base della stessa euristica; nel peggiore dei casi infatti, il consumo ci LEC-DN esteso è del 14.3% inferiore rispet-

to a quello di LEC-DN base. Dal punto di vista dei tempi di esecuzione invece, LEC-DN esteso risulta in linea con le euristiche NN e BN. Grazie al modello di applicazione che supera la limitazione master/slave imposta negli altre soluzioni esaminate, l'algoritmo LEC-DN è stato scelto per il confronto diretto con la soluzione elaborata nel lavoro oggetto di questa tesi. Nonostante questo, l'algoritmo non può essere adottato all'interno dell'euristica sviluppata a causa della presenza delle metriche sul consumo di energia non considerate nello sviluppo dell'euristica.

2.1.8 Approccio Random Walk

Un algoritmo di *mapping* basato sull'euristica Random Walk è stato proposto da **Wichslgartner et al.** in [3]. Questa euristica è appositamente studiata per applicazioni strutturate ad albero (vedi Figura 2.8) in cui sono presenti sia processi di elaborazione che processi di comunicazione, ed utilizza un modello di architettura composta da PE omogenee ed analogo a quanto illustrato in Figura 2.1.

Il problema dell'assegnazione dei processi a un PE viene scomposto in due parti:

- una parte di assegnazione dei processi alle PE;
- una parte di routing delle comunicazioni tra processi una volta che essi siano tutti assegnati a un PE.

L'algoritmo che implementa assegnamento e *routing* è distribuito e implementa l'euristica Random Walk. Esso si svolge su due fasi:

- nella prima fase il processo iniziale viene assegnato dal PE master del MPSoC a un generico PE;
- nella seconda fase, il PE a cui è stato assegnato il processo iniziale si prende carico di effettuare l'assegnazione degli altri processi di cui è

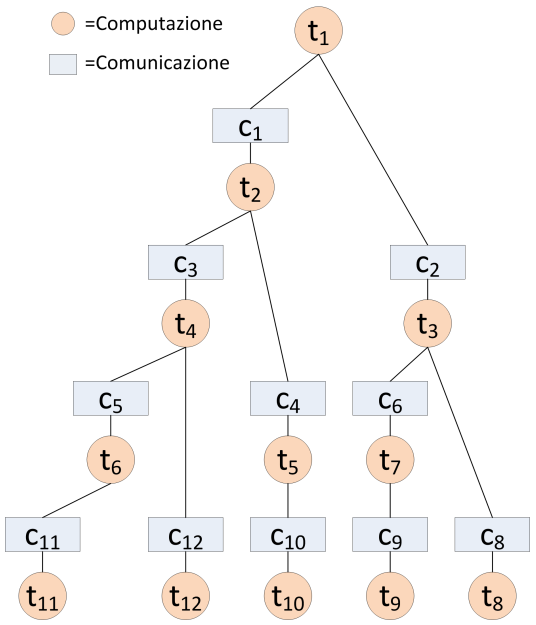


Figura 2.8: Task-graph considerato in [3]

composta l'applicazione nel momento in cui essi verranno richiesti. L'assegnamento è effettuato utilizzando l'euristica Random Walk.

Per quanto riguarda la prima fase, il processo iniziale può essere assegnato utilizzando varie tecniche, [3] in particolare indica le tecniche k-means [17], Farthest-Away di **Hochbaum et al.** [18] e Near Convex Region di **Chou et al.** [19].

Una volta assegnato il processo iniziale è possibile quindi assegnare gli altri processi componenti l'applicazione attraverso la tecnica Random Walk. L'euristica funziona nel seguente modo:

- quando viene richiesta l'esecuzione di un nuovo processo, il PE che sta eseguendo il processo deve scegliere su quale PE assegnare il processo;
- il PE seleziona casualmente un PE vicino che sia dotato di sufficienti risorse per eseguire il processo. Il PE selezionato esegue la stessa pro-

cedure escludendo dalla scelta il PE da cui è stato scelto. La procedura viene ripetuta un arbitrario numero di volte;

- una volta terminato il passo precedente, il PE contenente il processo che ha richiesto l'esecuzione, valuta il migliore dei PE selezionati secondo la metrica utilizzata e assegna il processo a quel PE;

In [3], la metrica utilizzata è quella del carico medio della NoC. I risultati riportati mostrano che il lavoro svolto in [3] permette di ottenere risultati paragonabili a quelli ottenibili con PL e NN sia in termini di tempi di esecuzione che in termini di volume di traffico scambiato sulla NoC.

Grazie alla metrica utilizzata, l'algoritmo Random Walk è stato anch'esso confrontato con la soluzione sviluppata in questo lavoro di tesi. Tuttavia, la struttura dell'applicazione ad albero risulta troppo vincolante in quanto non adatta alla soluzione sviluppata in questo lavoro di tesi che prevede una diversa struttura dell'applicazione (vedi Capitolo 3).

2.2 Modelli di invecchiamento su sistemi MPSoC

Questa sezione contiene l'analisi dei lavori presenti in letteratura riguardanti la modellazione del fenomeno dell'invecchiamento su sistemi MPSoC. Essendo tali architetture recenti, la letteratura è povera di lavori che affrontino l'argomento e solo due fonti sono risultate adatte allo scopo che questa tesi si prefigge. Queste due fonti permettono comunque di dare una solida base teorica alle scelte effettuate all'interno dell'euristica in materia di prevenzione del fenomeno dell'invecchiamento. A dispetto dei numerosi algoritmi di *mapping* dinamico su sistemi MPSoC presenti in letteratura, infatti, pochi di essi tengono in considerazione gli effetti che uno sbilanciato utilizzo dei PE può apportare al sistema, in termini di guasti sia transitori che permanenti. L'elevato uso dei PE può portare infatti al surriscaldamento dell'area del chip su cui esso è fisicamente presente,

andando ad introdurre possibili effetti di NBTI che possono sommarsi ad altri effetti quali ElectroMigration (EM), Time-Dependent Dielectric Breakdown (TDDB) e Stress Migration (SM). L'unione di questi fenomeni può portare i componenti più utilizzati di un MPSoC ad invecchiare più velocemente rispetto ai componenti meno utilizzati. Questo fenomeno, detto **invecchiamento precoce**, deve essere minimizzato il più possibile a fine di massimizzare la vita media dei componenti dell'intero MPSoC. Per questo motivo, al fine di sviluppare una soluzione che prenda in considerazione questo fattore, si sono analizzati alcuni lavori di modellazione del fenomeno presenti in letteratura.

In [20], **Huang** elabora un algoritmo di allocazione dei processo su MPSoC che permetta di allungare la vita media del sistema andando a prevenire l'insorgere di fenomeni di invecchiamento. Il lavoro, effettua il mapping dei processo utilizzando la tecnica di *Simulated Annealing* guidata dalla funzione di costo riportata nell'Equazione 2.5.

$$\mathrm{Cost} = \mu \cdot 1_{\exists i : e_i > d_i} - \mathrm{MTTF}^{sys} \tag{2.5}$$

L'Equazione 2.5 è composta da due parti essenziali: Una prima parte che verifica il soddisfacimento dei requisiti di deadline di un processo, una seconda parte che considera il fenomeno dell'invecchiamento basandosi sul Mean Time To Failure (MTTF) del sistema. Il calcolo del MTTF del sistema è molto oneroso, se affrontato utilizzando un modello accurato, e non rende possibile l'implementazione dell'algoritmo presentato in questa soluzione, pertanto il lavoro propone una soluzione basata su alcune ottimizzazioni che permettono di ottenere un calcolo approssimato, ma comunque utilizzabile, del valore del MTTF. Il lavoro presentato in [20] suggerisce quindi come un uso omogeneo, dal punto di vista del tempo di attività di ciascun componente del MPSoC, possa portare a un significativo incremento della vita media dei componenti stessi, andando a limitare il fenomeno di invecchiamento.

I risultati mostrati in [20] mostrano come l'effettiva vita media di un MPSoC risulti effettivamente allungata fino al 60% qualora la soluzione venga adottata, in particolare, in sistemi real-time.

Il secondo lavoro analizzato, è riportato in [4]. **Paterna et al.** propongono un metodo che permetta di utilizzare i PE con meno probabilità di guasti, mantenendo invece in *idle* i PE con più probabilità di guasto. Questo lavoro non propone un modello di invecchiamento, ma un'euristica basata sulle informazioni derivate da esso. Il modello di architettura utilizzato in questo lavoro è leggermente diverso da quanto illustrato in Figura 2.1. Pur essendo infatti basato su NoC, il modello di MPSoC utilizzato è composto da un processore master della famiglia st231 e da un numero configurabile di unità Very Long Instruction Word (VLIW) che fungono da acceleratori. Ogni acceleratore inoltre possiede una *cache* istruzioni ed è anche presente una memoria condivisa tra processore master e acceleratori. La differenza sostanziale dall'architettura di Figura 2.1 risiede nel modo di eseguire le applicazioni. Essendo i PE acceleratori hardware il modello di esecuzione adottato dall'architettura è del tipo Single Program Multiple Data (SPMD) e pertanto può essere eseguita una sola applicazione alla volta, nonostante gli acceleratori possano eseguire porzioni diverse del codice dell'applicazione. Il modello di applicazione invece è il classico modello che vede la stessa come un grafo diretto aciclico i cui nodi sono i processi e i cui lati rappresentano i legami di precedenza tra gli stessi. La soluzione proposta in [4], quindi, viene eseguita dal processore master e si occupa di calcolare la quantità di tempo che ciascun acceleratore del MPSoC deve passare in *idle*. Per calcolare questo tempo, vengono utilizzate le informazioni restituite dal modello di invecchiamento implementato nella soluzione, che nella fattispecie non viene specificato, lasciando intendere come un qualunque modello che restituisca i dati richiesti dall'euristica possa essere utilizzato. I tempi di *idle* così calcolati vengono quindi inseriti in una tabella, che viene

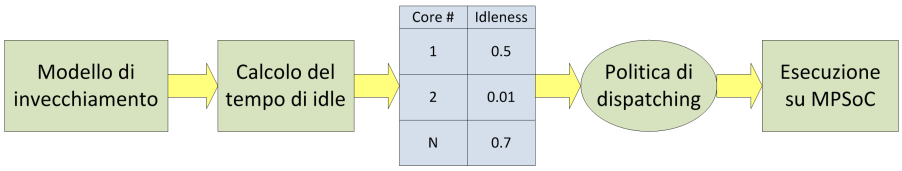


Figura 2.9: Funzionamento dell'euristica presentata in [4]

consultata dal *mapper* per scegliere quali acceleratori utilizzare per eseguire l'applicazione. Il funzionamento dell'euristica è illustrato in Figura 2.9. I risultati ottenuti dalla simulazione dell'euristica mostrati in [4] mostrano che essa riesce ad allungare la vita media dei componenti, andando però a rallentare l'esecuzione delle applicazioni in modo proporzionale al numero di acceleratori che vengono forzati in *idle* a causa dell'intervento dell'euristica. Nel peggiore dei casi comunque, il calo di prestazioni rimane nell'ordine del 20%.

In questo capitolo sono state introdotte le euristiche presenti nello stato dell'arte riguardanti le tematiche del *mapping* dinamico di applicazioni su architetture di tipo MPSoC e delle strategie implementabili al fine di ridurre il fenomeno dell'invecchiamento precoce dovuto a un uso sbilanciato delle risorse presenti all'interno dei dispositivi. Sono state inoltre scelte tre delle euristiche presentate come soluzioni con cui confrontare il lavoro sviluppato in questa tesi. Tali euristiche sono Path Load, Low Energy Consumption - Dependences Neighborhood e Random Walk.

Nel capitolo successivo verrà introdotto il modello utilizzato per lo sviluppo dell'euristica oggetto del lavoro presentato in questa tesi e basato sull'architettura p2012 di S.T. Microelectronics, in grado di eseguire applicazioni scomponibili in sub-applicazioni a loro volta di tipo *multithreaded*.

Capitolo 3

Modello Sviluppato

Il primo obiettivo che il lavoro descritto in questa tesi si è posto, è quello di analizzare ed elaborare un modello di esecuzione adeguato della piattaforma p2012 di S.T. Microelectronics. In questo capitolo viene affrontata sia l'analisi dell'architettura che l'introduzione di un modello di applicazione con il relativo modello di esecuzione. Tale modello è stato poi utilizzato per validare l'algoritmo di *mapping* sviluppato in questo lavoro di tesi. Il modello è composto da due parti:

- il modello dell'architettura utilizzato;
- il modello di applicazione utilizzato.

Il modello dell'architettura viene introdotto illustrando la piattaforma utilizzata come base per lo sviluppo del lavoro, ovvero il Multi-Processor SoC (MPSoC) "p2012 "sviluppato da ST Microelectronics. Il modello di applicazione utilizzato viene invece introdotto specificando le esigenze che hanno portato allo sviluppo di tale modello, diverso da quanto utilizzato in letteratura, e analizzando il funzionamento delle applicazioni sulla piattaforma p2012. Il capitolo si articola su cinque sezioni: nella **Sezione 3.1** viene introdotto il tipo di applicazioni utilizzate come base per il modello elaborato, nella **Sezione 3.2** viene analizzata la piattaforma "p2012 ", nella

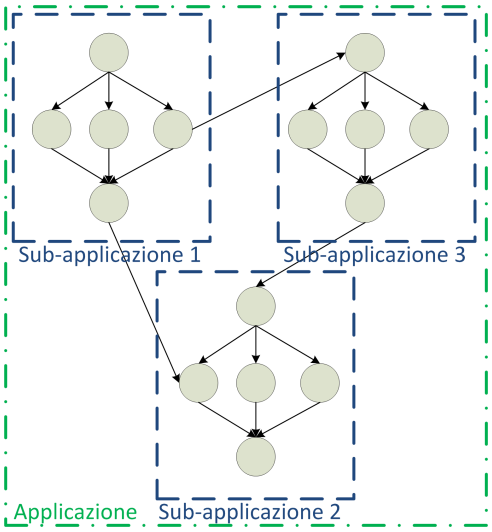


Figura 3.1: Tipo di applicazione utilizzato come riferimento per lo sviluppo del modello

Sezione 3.3 viene analizzato il comportamento del *mapping* e dello *scheduling*, mentre la modellizzazione dell'architettura è introdotta nella **Sezione 3.4**. La **Sezione 3.5** contiene invece l'analisi del metodo di esecuzione delle applicazioni sulla piattaforma, insieme all'introduzione del rispettivo modello.

3.1 Applicazioni considerate

Nel Capitolo 2 sono stati introdotti diversi algoritmi presenti in letteratura che implementanti tecniche di *mapping* dinamico. La maggior parte di questi algoritmi utilizza un modello di applicazione basato sul paradigma *master/slave*. Al fine di un corretto sviluppo dell'euristica oggetto di questa tesi, il paradigma *master/slave* risulta limitante, in quanto si vuole utilizzare un modello di applicazione che sia facilmente eseguibile dall'architettura p2012 sfruttano la possibilità di eseguire un elevato numero di *thread* contemporaneamente. Il modello di applicazione sviluppato quindi, utilizza come unità minima i *thread* che a loro volta formano il *task-graph* dell'applicazione. Data la natura della piattaforma p2012 pe-

rò si è deciso di strutturare il modello dell'applicazione su due livelli, per meglio adattare l'esecuzione della stessa all'architettura sviluppata da S.T. Microelectronics.

L'applicazione viene quindi vista come composta da una o più sub-applicazioni, a loro volta composte da parti del *task-graph* ed eseguibili in parallelo. In questo modo si ottiene un modello di applicazione di riferimento che permette di eseguire contemporaneamente parti della stessa, che a loro volta poi possono sfruttare i numerosi Processing Element (PE) di cui ciascun *cluster* dell'architettura p2012 è composto (vedi Paragrafo 3.2).

Tale modello deve inoltre considerare i vincoli di precedenza presenti tra i *thread* che compongono l'applicazione.

Tali vincoli infatti possono portare a vincoli di precedenza tra sub-applicazioni che devono essere correttamente gestiti in fase di esecuzione dell'applicazione. Il modello deve quindi necessariamente permettere di rappresentare correttamente un'applicazione che possieda le seguenti caratteristiche:

- l'applicazione può essere composta da una o più sub-applicazioni
- ciascuna sub-applicazione è composta da più *thread*, alcuni dei quali eseguibili parallelamente, che oltre a formare il *task-graph* dell'applicazione possono avere vincoli di precedenza con *thread* appartenenti a sub-applicazioni diverse da quella di appartenenza.

In Figura 3.1 è illustrato un esempio di struttura di un'applicazione con le caratteristiche individuate.

3.2 Piattaforma p2012

L'architettura p2012, descritta dettagliatamente in [6], è un'architettura fortemente parallela su System-on-Chip (SoC) organizzata su due livelli, denominati livello "Network-on-Chip (NoC) e livello Cluster".

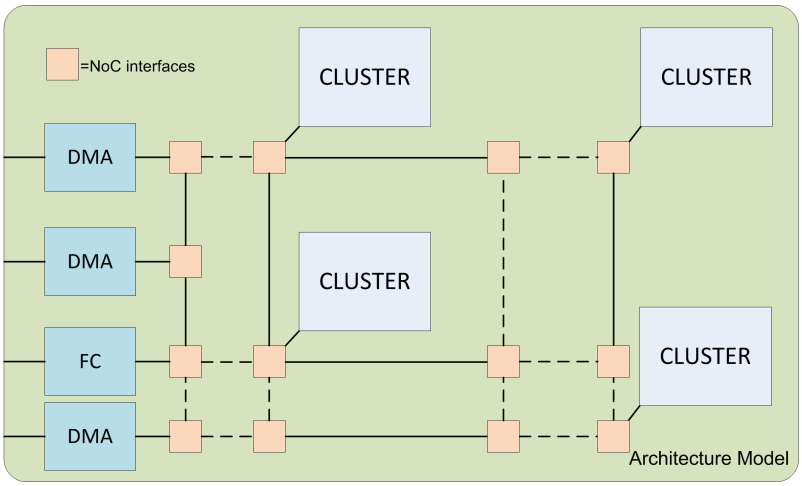


Figura 3.2: Livello NoC dell'architettura

- nel livello "NoC", l'architettura viene vista come una NoC bidimensionale reticolare nella quale ciascun router è collegato ad un Cluster. Oltre ai Cluster, tutti identici, sono collegati alla NoC anche alcuni componenti di controllo della stessa e il Fabric Controller. Questo livello coincide con l'intero SoC.

- nel livello "Cluster"invece, ciascuno dei cluster identici, modellizzato come sistema *multicore*, viene caratterizzato sia architetturalmente che funzionalmente.

È dunque evidente come l'organizzazione dei componenti all'interno dell'architettura sia rivolta all'esecuzione fortemente parallela di processi. Per questo motivo è quindi necessaria un'attenta caratterizzazione della stessa per permettere l'elaborazione di un adeguato modello di funzionamento. Tale modello rappresenta la base su cui sviluppare un efficiente algoritmo di *mapping* che possa.

3.2.1 Livello NoC

Il livello principale dell'architettura è rappresentato in Figura 3.2. Oltre

alla presenza dei nodi identici, è possibile identificare alcuni componenti ausiliari collegati anch'essi direttamente alla NoC: il *Fabric Controller* e alcuni controllori Direct Memory Access (DMA). Il *Fabric Controller* è l'elemento incaricato alla gestione dell'assegnamento delle applicazioni sui *cluster* presenti nell'architettura mentre i controllori DMA si occupano di gestire il trasferimento dei dati e del codice dell'applicazione dalla memoria del *Fabric Controller* al nodo assegnato per l'esecuzione dell'applicazione o di parte di essa. Da quanto descritto in [6], il funzionamento dell'architettura può essere descritto nel modo seguente:

- su ogni *cluster* può essere eseguita un'unica applicazione (o parte di essa) alla volta. L'applicazione può essere multiprocesso;
- il codice di ogni applicazione è contenuto all'interno della memoria accessibile dal *Fabric Controller*. È però necessario che il codice di ciascuna applicazione, o di ciascuna parte di applicazione, abbia dimensioni contenute (massimo 128KB) a causa della limitata grandezza della cache istruzioni presente all'interno di ciascun *Cluster*;
- la NoC viene utilizzata sia per l'invio di pacchetti di comunicazione tra i vari componenti, sia per l'invio dei dati e del codice gestito dai controllori DMA;
- il codice ed i dati di inizializzazione delle applicazioni vengono trasferiti dal *Fabric Controller* ai *Clusters* tramite tre canali DMA: due canali (uno per il codice e uno per i dati iniziali) diretti dal *Fabric Controller* al *Cluster* destinazione e un canale diretto nel senso opposto per l'invio dei risultati di esecuzione. I tre canali DMA utilizzano la NoC come mezzo di trasferimento dei dati.

L'architettura così descritta ricorda molto da vicino l'organizzazione interna di una moderna Graphics Processing Unit (GPU), che rende possibile

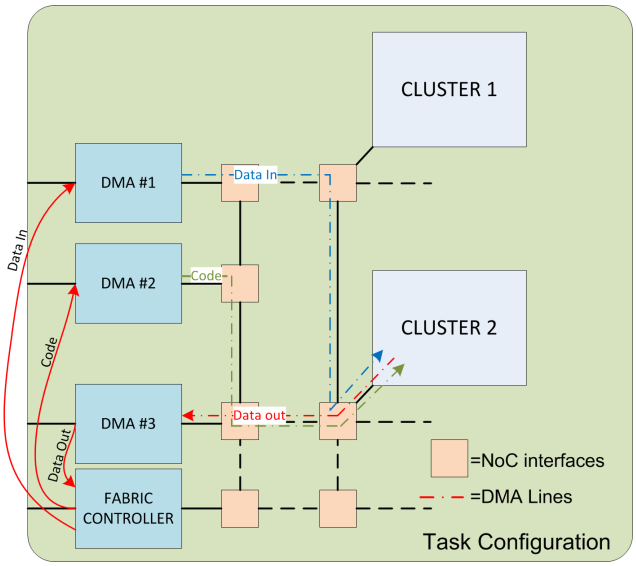


Figura 3.3: Assegnamento di un processo a un *cluster*: trasferimento dei dati

la parallelizzazione di un'applicazione mediante la contemporanea esecuzione della stessa su un elevato numero di unità processanti (modello Single Instruction Multiple Data, SIMD). A differenza di una GPU però, l'architettura considerata permette di assegnare a ciascun *Cluster* applicazioni diverse, andando di fatto a implementare un modello di esecuzione parallelo non a livello di dati, ma a livello di applicazioni. Benchè da tenere in considerazione, le soluzioni di *scheduling* e/o *mapping* applicate alle GPU non sono direttamente applicabili all'architettura descritta poichè considerano il parallelismo a livello dati e non a livello applicazione. Considerando quanto illustrato, diventa quindi critico il trasferimento dei dati dal *Fabric Controller* ai *Clusters* in quanto potrebbe rappresentare un importante collo di bottiglia. A verifica di quanto detto, l'effettivo comportamento dell'assegnazione di un'applicazione a un *Cluster*, illustrato in Figura 3.3 conferma come il *mapping* dell'applicazione sia un potenziale collo di bottiglia a questo livello dell'architettura. I soli tre canali DMA potrebbero quindi non garantire le prestazioni necessaria e la loro gestione risulta particolarmente critica ai fini dell'efficienza.

Dal punto di vista dell'esecuzione delle applicazioni, invece, può essere ipotizzato il seguente funzionamento (ricordando che il *mapping* viene effettuato dal *Fabric Controller*):

- l'applicazione da eseguire viene caricata all'interno della memoria del *Fabric Controller*;
- l'applicazione viene assegnata a uno o più cluster (a seconda della grandezza e dalle risorse richieste dalla stessa) in base alla politica di mapping implementata all'interno del Fabric Controller;
- Codice e dati vengono trasferiti dal *Fabric Controller* al *Cluster* designato secondo le modalità precedentemente descritte e rappresentate in Figura 3.3;
- l'applicazione viene eseguita dal *Cluster*;
- una volta completata l'esecuzione, i risultati vengono inviati al *Fabric Controller*.

La corretta modellizzazione del funzionamento descritto risulta di particolare importanza per la creazione di un simulatore dell'architettura, sopratutto per quanto riguarda i tempi di trasferimento del codice e dei dati dal *Fabric Controller* al *Cluster* deputato all'esecuzione dell'applicazione. Tali tempi dipendono essenzialmente dalla grandezza dei dati da trasferire, dalla velocità della NoC e dal numero di nodi da attraversare all'interno della stessa per raggiungere il *Cluster* destinazione. Il modello di esecuzione a livello "NoC "può essere quindi visto come un modello "a grana grossa", dove l'unità minima di computazione considerata è il singolo *Cluster*. Essendo il numero di *Clusters* e la velocità della NoC i parametri caratterizzanti del livello "NoC"essi risultano importanti per il modello creato e vengono formalmente definiti nel modo seguente, con i vincoli sul numero massimo di *Clusters* e sulla parità ripresi da [6]:

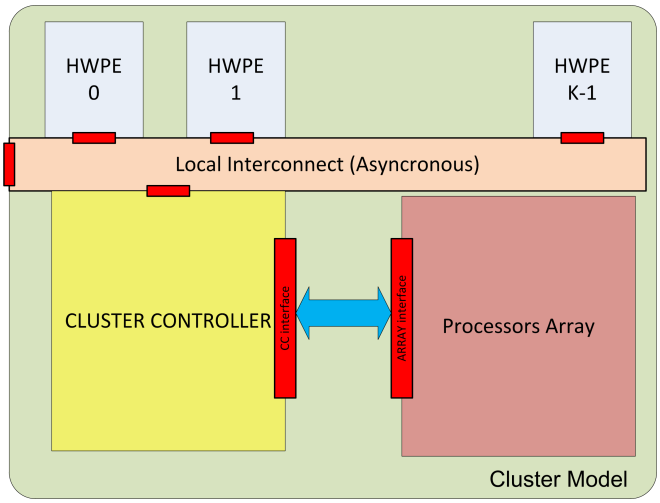


Figura 3.4: Livello Cluster dell'architettura

$$\begin{aligned} N_{cx} &= \text{Larghezza, in numero di clusters, della NoC} \\ N_{cy} &= \text{Altezza, in numero di clusters, della NoC} \\ 0 &< N_{cx} + N_{cy} \leqslant 16 \ \text{ e } \ (N_{cx} + N_{cy}) \bmod 2 = 0 \end{aligned} \tag{3.1}$$

$$V_{NoC} = \text{Velocità della NoC [KB/s]} \tag{3.2}$$

3.2.2 Livello cluster

Mentre a livello NoC il modello si occupa di descrivere il l'assegnamento delle applicazioni, a livello cluster vi è la necessità di effettuare lo scheduling dei *thread* che compongono l'applicazione assegnata ad un *Cluster*. Anche in questo caso le operazioni devono essere svolte in modo efficiente e si considerano già implementati aspetti di tolleranza ai guasti.

In questo paragrafo sono quindi analizzati la struttura ed il funzionamento di un singolo *Cluster* al fine di ricavarne i parametri descrittivi sui quali è stato poi costruito il modello. L'architettura interna di un singolo *Cluster*, raffigurata in Figura 3.4 è composta da tre elementi principali:

- il *cluster controller*;
- L'*array* di processori;
- le aree configurabili con acceleratori *hardware*.

Il *cluster controller* è l'analogo del *fabric controller* a livello NoC per un singolo *Cluster*, ovvero il componente deputato alla gestione dell'applicazione in esecuzione e delle risorse dello stesso. L'*array* di processori invece, contiene fino a un massimo di 16 elementi processanti (CPU STxP70-4 [6]) operanti in parallelo che condividono due aree di memoria:

- Un'area da 128KB suddivisa in 64 banchi e chiamata P$, utilizzata come cache condivisa per il codice;
- Un'area da 256KB, chiamata TCDM e utilizzata come cache condivisa per i dati;

Il terzo componente, l'*array* di aree configurabili con acceleratori *hardware*, rappresenta un'interessante soluzione per poter accelerare processi che richiedono particolari funzioni. Gli acceleratori non sono però ri-configurabili ma fissi e dipendenti dalla specifica implementazione dell'architettura. Gli acceleratori possono comunicare direttamente solo con il *Cluster Controller* quindi ogni comunicazione processori-acceleratori deve necessariamente passare da esso. Le Figure 3.5 e 3.6 completano la descrizione dell'architettura di un *Cluster* illustrando la struttura interna del *Cluster Controller* e dell'*array* di processori. Per quanto riguarda il *Cluster Controller*, l'elemento di maggior interesse è il *Cluster Processor*, identico ai processori presenti nell'*array* di elementi processanti, che ha il compito di coordinare e gestire i vari elementi presenti sul *Cluster*. Oltre ad esso sono anche presenti tre controllori DMA per la gestione delle comunicazioni con il *Fabric Controller* (secondo lo schema descritto in 3.2.1), l'interfaccia NoC per la comunicazione con gli altri clusters e l'interfaccia di comunicazione dedicata verso l'*array*

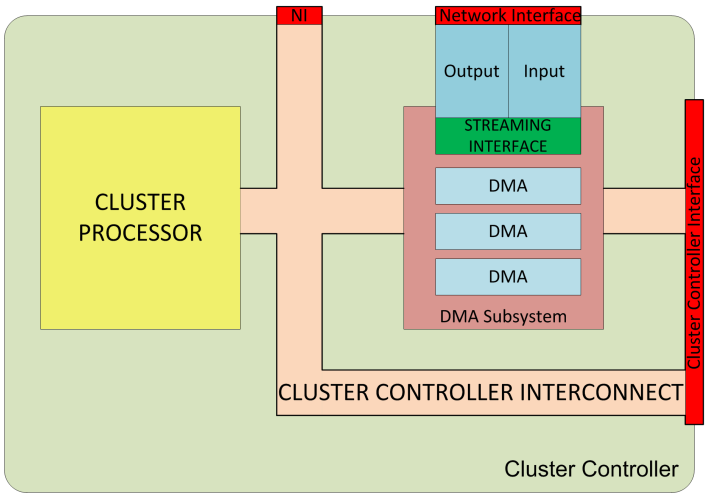


Figura 3.5: Architettura del Cluster Controller

di acceleratori *hardware*. Per quanto riguarda l'*array* di unità processanti, è possibile notare, come introdotto in precedenza, lo schema di memoria totalmente condivisa da tutti i processori (divisa in P$ e TCDM [6]) e una parte denominata "*Hardware Accelerator*"che fornisce primitive di sincronizzazione implementate in *hardware* e limitate funzionalità di tolleranza ai guasti (essenzialmente *watchdogs hardware*).

Un aspetto critico, quindi, risiede nelle funzionalità supportate dall'*array* di elementi processanti per quanto riguarda la gestione dei processi in esecuzione sulle CPU. La gestione dei *thread*, infatti, non avviene tramite un mini sistema operativo caricato nel cluster controller, ma attraverso eventi lanciati dalle CPU e intercettabili dalle stesse, gestiti tramite quanto offerto dall'*Hardware Accelerator* implementato nell'*array*. All'atto pratico, quindi, lo scheduling dei *thread* sulle varie CPU avverrà sulla base degli eventi di creazione, attesa, modifica e fine esecuzione messi a disposizione dall'architettura nell'*hardware accelerator* stesso. Eventuali strutture dati di supporto dovranno risiedere nella memoria condivisa TCDM. Questo tipo di funzionamento deve essere necessariamente preso in considerazione nell'elaborazione del modello di applicazione gestibile da un cluster e servi-

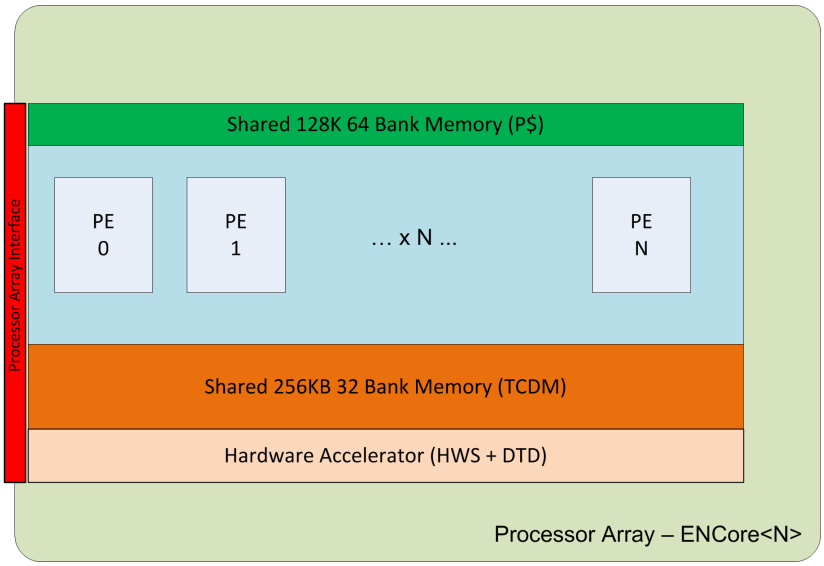


Figura 3.6: Architettura dell'array di elementi processanti

rà inoltre come base sia per lo scheduling dei *thread* facenti parte dell'applicazione da eseguire che per l'implementazione del meccanismo stesso all'interno del simulatore.

L'architettura di un cluster risulta quindi complessa. A fronte di questa complessità risultano utili, ai fini dell'elaborazione del modello architetturale, i seguenti due parametri:

$$N_{CPU} = \text{Numero di CPU nell'array}, 1 \leqslant N_{CPU} \leqslant 16 \tag{3.3}$$

per quanto riguarda il numero di elementi processanti presenti e

$$N_{AA} = \text{Numero di Aree per Acceleratori}, 1 \leqslant N_{CPU} \leqslant 8 \tag{3.4}$$

per quanto riguarda il numero di aree configurabili con acceleratori *hardware*. Tutti i vincoli imposti sono ricavati da [6]. È inoltre importante rimarcare che gli elementi processanti presenti nell'*array* debbano essere identici.

Un aspetto molto importante a questo livello riguarda inoltre la presenza di meccanismi di rilevamento e tolleranza ai guasti. Essi risultano fon-

damentali per la corretta esecuzione delle applicazioni e per l'implementazione di una strategia di *mapping* che permetta di bilanciare il carico di lavoro sui PE. Come introdotto nel Capitolo 1 l'ambito di ricerca all'interno del quale questo lavoro di tesi si pone è molto vasto e si è quindi scelto di non affrontare la tematica dell'implementazione di meccanismi di tolleranza ai guasti all'interno dei *cluster* di cui la piattaforma p2012 si compone. Per questo motivo, al fine di implementare correttamente una strategia di *mapping* efficace, si è scelto di considerare i *cluster* dotati di meccanismi di tolleranza ai guasti (rilevamento e diagnosi) perfettamente funzionanti che permettano di ottenere le informazioni necessarie ad una corretta gestione di eventuali unità totalmente o parzialmente guaste.

A chiusura del paragrafo, viene riportata la Tabella 3.1 nella quale vengono riassunti i parametri descrittivi dell'architettura sia a livello NoC (introdotti nel Paragrafo 3.2.1) sia i parametri a livello cluster.

Tabella 3.1: Elenco di tutti i parametri descrittivi dell'architettura

Riferimento	**Parametro**	**significato**
Livello NoC	N_{cx}	Larghezza, in numero di clusters, della NoC
	N_{cy}	Altezza, in numero di clusters della NoC
	V_{NoC}	Velocità della NoC [KB/s]
Livello Cluster	N_{CPU}	Numero di CPU presenti nel Cluster
	N_{AA}	Numero di aree per Acceleratori presenti nel Cluster

3.3 Mapping e Scheduling delle applicazioni

L'introduzione del modello di architettura (Paragrafo 3.2) e il tipo di applicazione che si vuole eseguire sull'architettura stessa (Paragragfo 3.1), hanno reso necessaria l'introduzione di un metodo base di *mapping* delle sub-applicazioni sui *Clusters* e di scheduling dei *thread* sui PE interni ai *Clusters* adattato ad essa. Tale metodo deve tener conto del modello totalmente dinamico di esecuzione, ovvero con tempo di arrivo e struttura delle applicazioni non noti a priori e deve supportare l'utilizzo dell'euristica elaborata in questo lavoro di tesi Come riportato nella Sezione 3.2, a livello NoC, il *Fabric Controller* è il componente incaricato per l'esecuzione del *mapping* delle applicazioni mentre, a livello cluster, le operazioni di *scheduling* sono gestite dal *Cluster Controller*. Questa sezione si occupa di introdurre delle politiche base di *mapping* delle applicazioni e *scheduling* dei *thread* andando inoltre a individuare eventuali strutture software di supporto necessarie ai componenti che gestiscono le due operazioni. La sezione si articola su due Paragrafi: nel **Paragrafo 3.3.1** viene trattato il livello NoC, mentre nel Paragrafo **3.3.2** viene trattato il livello cluster.

3.3.1 Livello NoC: Mapping

A livello NoC, il *Fabric Controller* esegue il *mapping* delle applicazioni, ovvero si occupa di assegnare le applicazioni, o parti di esse, ai *Clusters*. In [6] non viene indicata una precisa politica di *mapping* seguita dal *Fabric Controller*, pertanto si ipotizza che esso utilizzi un sistema di tipo First In First Out (FIFO)(Figura 3.7). Il funzionamento è composto dai seguenti passi:

- ogni volta che un'applicazione richede di essere eseguita, essa viene inserita in una coda di applicazioni da eseguire;
- l'applicazione all'inizio della coda viene scelta dall'algoritmo di *mapping*;

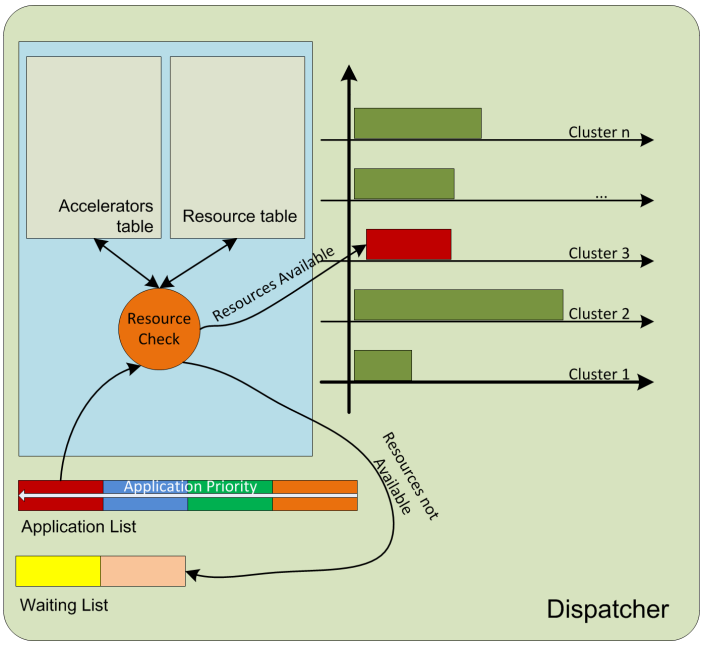


Figura 3.7: Schema del *mapper* base delle applicazioni

- viene effettuata una verifica della disponibilità delle risorse del sistema, assicurando che le richieste dell'applicazione da eseguire possano essere soddisfatte. Per effettuare questa operazione si ipotizza la presenza, nella memoria del *Fabric Controller*, di strutture di supporto contenenti informazioni sullo stato dei clusters e sugli acceleratori disponibili su di essi (implementate utilizzando delle tabelle). Se le risorse disponibili sono sufficienti, l'applicazione verrà assegnata al *Cluster* più adeguato all'esecuzione sulla base delle informazioni contenute nelle tabelle, altrimenti essa sarà inserita in una coda di applicazioni rigettate ed in attesa di essere eseguite;

- se l'applicazione può essere eseguita, codice e dati iniziali vengono trasferiti al *Cluster* responsabile dell'esecuzione rendendo possibile l'esecuzione della stessa;

- si passa all'applicazione successiva, dando priorità ad eventuali ap-

plicazioni presenti all'interno della coda di attesa delle applicazioni rigettate;

Si rende necessario quindi implementare le strutture di supporto tramite due tabelle al fine di rendere possibile la verifica dello stato delle risorse e la scelta del *Cluster* più adatto all'esecuzione di una determinata applicazione o parte di essa, nel caso sia composta da più sub-applicazioni. La prima tabella, denominata "Tabella degli acceleratori"(*Accelerators Table* in Figura 3.7) contiene la lista degli acceleratori *hardware* presenti nel MPSoC e la loro collocazione nei *Clusters*. In questo modo, se un'applicazione richiede che alcune sue parti vadano eseguite in *hardware*, è possibile assegnare l'applicazione ai *Clusters* adatti a soddisfare le richieste. Nel caso non sia possibile soddisfare qualche requisito di esecuzione *hardware* la parte dell'applicazione interessata può eseguita in software su una normale PE.

La seconda tabella invece, denominata "Tabella delle Risorse"(*Resource Table* in Figura 3.7), contiene una riga per ogni *Cluster* presente nel MPSoC contenente le informazioni sullo stato del *Cluster* stesso e delle risorse contenute in esso. Grazie all'uso combinato delle due tabelle, quindi, è possibile scegliere il *Cluster* che più si presta ad eseguire l'applicazione da assegnare.

3.3.2 Livello Cluster: Scheduling

Una volta assegnata ad un cluster, l'esecuzione dell'applicazione viene gestita dal *Cluster Controller*, che ha il compito di eseguire correttamente e sui PE selezionati, siano essi acceleratori *hardware* o CPU, i *thread* che compongono l'applicazione. Dato il modello di applicazione utilizzato (vedi Sezione 3.5) ed il rispettivo *task-graph*, riportato in Figura 3.10, lo scheduling dei *thread* deve essere eseguito correttamente e in modo tale che vengano rispettate le dipendenze tra le applicazioni.

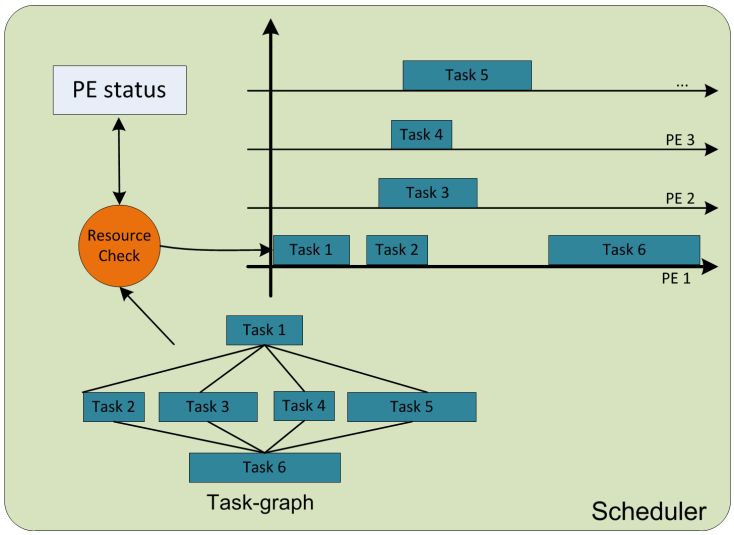


Figura 3.8: Schema dello *scheduler* dei *thread*

In Figura 3.8 viene illustrato, in modo analogo a quanto fatto per il *Fabric Controller*, il funzionamento della fase di *scheduling*. Anche all'interno del *Cluster Controller* è quindi necessario utilizzare delle strutture di supporto utili al fine della corretta allocazione dei *thread* sui PE presenti all'interno del *Cluster*. In particolare, viene previsto l'uso di una tabella contenente le informazioni sullo stato delle CPU e degli acceleratori *hardware*, in modo tale che lo *scheduler* possa allocare correttamente i vari *thread* escludendo unità guaste o in uso. Lo scheduling base delle applicazioni verrà quindi eseguito nel seguente modo:

- una volta ricevuto il codice e i dati dell'applicazione da eseguire, vengono lette le informazioni riguardanti il primo *thread* ed esso viene allocato sul primo PE libero, a seconda che si tratti di un *thread hardware* o di un *thread software*;
- completata l'esecuzione del primo *thread*, lo *scheduler* alloca, utilizzando una politica di tipo FIFO, i *thread* paralleli sui PE ed attende il completamento dell'ultimo di essi;

- una volta che anche l'esecuzione dell'ultimo *thread* parallelo risulta completa, viene schedulato il *thread* finale;
- completato il *thread* finale il *Cluster* invia i risultati di elaborazione al *Fabric Controller* e richiede una nuova applicazione da eseguire; qualora non risulti disponibile alcuna applicazione esso si mette in attesa.

3.4 Modello della piattaforma p2012

Il modello di architettura elaborato, sulla base di quanto esposto nel Paragrafo 3.2, è un modello di NoC arricchito con i parametri individuati e permette di implementare i modelli di *mapping* e *scheduling* presentati nella Sezione 3.3. Un *Cluster* pertanto viene definito come una quadrupla

$$C =< x, y, N_{CPU}, N_{AA} > \tag{3.5}$$

dove:
x, y sono le coordinate del cluster all'interno della NoC e
N_{CPU} e N_{AA} sono i parametri individuati nel Paragrafo 3.2.
Con questa definizione di *Cluster* è quindi possibile impostare la definizione dell'intero SoC come una quadrupla

$$SoC =< G_{NOC}, N_{cx}, N_{cy}, V_{NoC} > \tag{3.6}$$

con:
N_{cx}, N_{cy} e V_{NoC} parametri definiti nel Paragrafo 3.2 e G_{NOC} grafo diretto che descrive la geometria della NoC. Per completare la descrizione del modello dell'architettura è sufficiente definire G_{NOC} e specificare i vincoli di collegamento tra i *cluster* su di essa tenendo in considerazione la struttura a *mesh* della NoC, come specificato in [6].

$$
\begin{aligned}
&G_{NOC} =< C, L > \\
&\quad \text{con} \\
&C = \{(x, y, N_{CPU}, N_{AA}) \mid 0 \leqslant x \leqslant N_{cx} \wedge 0 \leqslant y \leqslant N_{cy}\} \\
&L = \{c_1, c_2 \mid c_1, c_2 \in C \wedge d_m(c_1, c_2 = 1)\}
\end{aligned}
\tag{3.7}
$$

Come introdotto, G_{NOC} viene descritto come un grafo diretto in cui i nodi rappresentano i *cluster* e gli archi i collegamenti tra di essi. L'Equazione 3.7 impone che ogni *cluster* abbia delle coordinate di posizione coerenti con la larghezza e l'altezza della NoC e descrive i collegamenti tra i *cluster* imponendo che due *cluster* siano collegati tra loro se e solo se la loro distanza di Manhattan è risulti uguale a 1. La distanza di Manhattan viene calcolata con la seguente formula:

$$d_m = (c_1, c_2) = |x_1 - x_2| + |y_1 - y_2| \tag{3.8}$$

Con la definizione della distanza di Manhattan, ogni parte del modello di architettura è stato specificato. Tutte le definizioni sono riassunte in tabella 3.2

Tabella 3.2: Elenco di tutti i parametri descrittivi della NoC

Oggetto	**Definizione**	**Parametri**	**Significato**
Cluster	$C =< x, y, N_{CPU}, N_{AA} >$	x, y	Coordinate del *Cluster* all'interno della NoC
		N_{CPU}, N_{AA}	Parametri definiti in 3.2
NoC	$G_{NOC} =< C, L >$	C	Set dei *Clusters*
		L	Set dei collegamenti tra *Clusters*
SoC	$SoC =< G_{NOC}, N_{cx}, N_{cy}, V_{NoC} >$	G_{NoC}	Descrizione della NoC
		N_{cx}, N_{cy}, V_{NoC}	Parametri definiti in 3.2

3.5 Modello di applicazione

L'obiettivo del lavoro di tesi è quello di elaborare un algoritmo efficiente di *mapping* per l'esecuzione di applicazioni fortemente parallele, pertanto il modello di applicazione è stato elaborato per rappresentare correttamente questo tipo di applicazioni. Il modello considerato prevede che ogni applicazione possa essere suddivisa in macro unità dette sub-applicazioni. Ciascuna sub-applicazione può essere eseguita su un singolo cluster ed è a sua volta composta da *thread* eseguibili dalle unità processanti o dagli acceleratori *hardware* presenti in ciascun *cluster*.

3.5.1 Esecuzione di una sub-applicazione

Una sub-applicazione può essere eseguita su un singolo cluster una volta trasferiti i dati ed il codice allo stesso. La gestione dell'esecuzione è affidata al *cluster controller* che dovrà coordinare l'esecuzione dei *thread* ed inviare i dati al *Fabric Controller* una volta completata la stessa. Un *thread*, essendo l'unità minima eseguibile del modello, può essere eseguito su una CPU presente nell'*array* di elementi processanti, oppure su un acceleratore *hardware* qualora la funzionalità richiesta fosse disponibile su uno degli acceleratori di cui ogni *cluster* è munito. Il *Cluster Controller*, però, come descritto nel Paragrafo 3.2, possiede solo limitate funzionalità di gestione dei *thread* (creazione, attesa e distruzione). In generale, quindi, il flusso di esecuzione della sub-applicazione assegnata al *Cluster* può essere rappresentato come in Figura 3.9 (vedi [21]). Tale esecuzione è composta da un blocco iniziale non parallelo, un blocco centrale in cui vengono eseguite più sezioni in parallelo e un blocco finale, eseguito dopo la corretta esecuzione di tutte le sezioni del blocco centrale. Il flusso di esecuzione di una sub-applicazione è stato quindi strutturato in modo analogo al flusso di esecuzione di un'applicazione *multithread* Al posto dei blocchi iniziali, finali e paralleli,

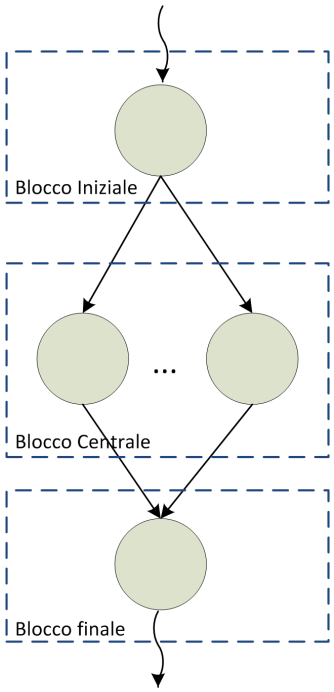


Figura 3.9: Flusso di esecuzione di una sub-applicazione

vengono eseguiti i *thread* che compongono la sub-applicazione, e il compito del *Cluster controller* sarà quindi quello di gestire correttamente gli eventi per assegnare correttamente i *thread* sulle risorse presenti all'interno del *cluster*. In questo modo viene sfruttato l'elevato numero di elementi processanti, in quanto ogni *thread* può essere associato ad uno di essi. Utilizzando un modello di esecuzione di questo tipo, la gestione degli eventi supportati dall'*Hardware Accelerator* presente all'interno dell'*array* di elementi processanti risulta di fondamentale importanza. Il modello di applicazione deve inoltre permettere di modellare dipendenze sia tra *thread* appartenenti alla medesima sub-applicazione, che tra *thread* appartenenti a sub-applicazioni diverse ma facenti parte della stessa applicazione. Pertanto il modello di esecuzione mostrato in Figura 3.9 è stato adattato al modello di applicazione adottato che viene illustrato in Figura 3.10. Sulla base di quanto introdotto, l'esecuzione di una generica sub-applicazione all'interno di un

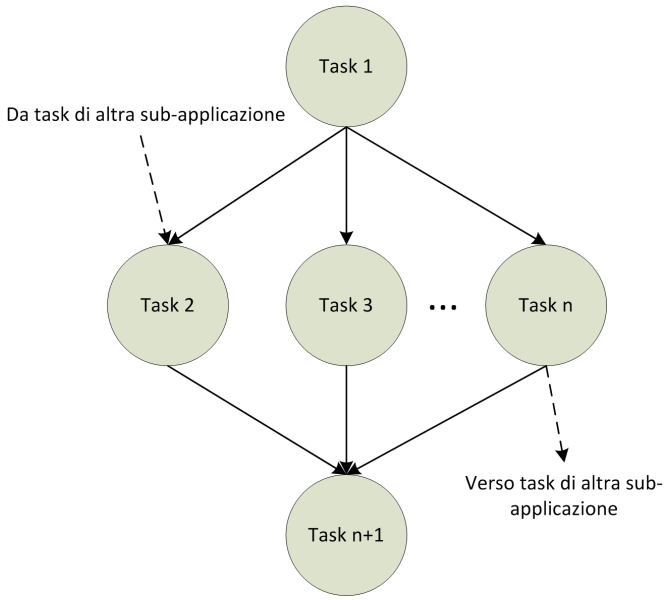


Figura 3.10: Modello di esecuzione di una sub-applicazione

cluster può essere quindi schematizzata nel modo seguente:

- il *Cluster Controller* riceve dal *Fabric Controller* codice e dati iniziali via NoC;
- il *Cluster Controller* assegna il primo *thread* e inizia l'esecuzione dell'applicazione;
- una volta eseguito il primo *thread* vengono assegnati ed eseguiti i *thread* del blocco centrale. Nel caso non vi siano sufficienti elementi processanti disponibili i *thread* verranno gestiti tramite una politica di tipo FIFO;
- nel caso non sia possibile eseguire un *thread* a causa di vincoli di precedenza con *thread* appartenenti ad altre sub-applicazioni, il *thread* verrà congelato in attesa che arrivi un messaggio di sincronizzazione da parte del *cluster* responsabile dell'esecuzione del *thread*;

- quando arriva un messaggio di sincronizzazione, i *thread* in attesa vengono eseguiti non appena vi siano sufficienti risorse disponibili;
- al termine dell'esecuzione di tutti i *thread* eseguibili in parallelo potrà essere eseguito il *thread* del blocco finale, andando a completare in questo modo l'esecuzione della sub-applicazione;
- una volta terminato il blocco finale, il *Cluster Controller* dovrà occuparsi di inviare i risultati dell'esecuzione al *Fabric Controller*.

Il modello della sub-applicazione deve considerare quindi il funzionamento esposto e possedere inoltre alcune informazioni aggiuntive, sia per quanto riguarda le sub-applicazioni in cui un'applicazione è divisa, sia per i singoli *thread* componenti ogni singola sub-applicazione. Le informazioni legate alle sub-applicazioni saranno dunque le seguenti:

- risorse necessarie alla corretta esecuzione della sub-applicazione, in termini di unità funzionali richieste che devono essere presenti all'interno delle unità processanti;
- elenco dei *thread* componenti la sub-applicazione;
- eventuale elenco degli acceleratori *hardware* richiesti per rendere possibile l'esecuzione dei *thread* componenti la sub-applicazione che potrebbero richiedere l'esecuzione su acceleratore *hardware*.

Le informazioni legate ai singoli *thread* che andranno considerate sono le seguenti:

- tempo di esecuzione, sia in *software* che in *hardware*;
- eventuale acceleratore *hardware* richiesto;
- eventuali *thread* da cui il processo dipende, anche se appartenenti ad altre sub-applicazioni;

- eventuali *thread* appartenenti ad altre sub-applicazioni che dipendono dal processo.

Dato il tipo di informazioni richieste per poter elaborare sul modello di applicazione un simulatore che permetta di validare l'euristica proposta in questo lavoro di tesi, diviene chiaro che sia necessario un lavoro di *profiling* delle applicazioni per poter ottenere la suddivisione delle applicazioni in sub-applicazioni che abbiano la struttura illustrata in Figura 3.10 e per poter ottenere le informazioni sui *thread* quali acceleratori *hardware* richiesti e stima dei tempi di esecuzione.

3.5.2 Modello

In questa sezione, viene introdotto il modello di applicazione utilizzato, descrivendo i parametri derivati dall'analisi del funzionamento e la struttura del modello stesso. Esso viene introdotto a partire dalla definizione formale dei *thread* per arrivare alla definizione formale di applicazione. Da quanto riportato nel Paragrafo 3.5.1, ad ogni *thread* devono essere associate le informazioni riguardanti i tempi di esecuzione e, in caso di possibile esecuzione in *hardware*, l'acceleratore richiesto. Un *thread* viene quindi definito come una tripla

$$T =< T_{sw}, A_{req}, T_{hw} > \tag{3.9}$$

dove:

$T_{sw}[s]$=tempo di esecuzione in software;

A_{req}=acceleratore *hardware* richiesto;

$T_{hw}[s]$=tempo di esecuzione in *hardware*.

Nel caso non sia possibile l'esecuzione in *hardware* di un *thread*, A_{req} e T_{hw} assumeranno rispettivamente i valori -1 e 0.

Data la definizione di *thread*, è possibile definire il *task-graph* di una sub-applicazione come un grafo aciclico orientato:

$$G =< T_i, E_i > \tag{3.10}$$

con:

T_i=set dei *thread*;

E_i=set dei vincoli tra *thread*.

Detti $pred(x)$ e $succ(x)$ due funzioni che restituiscono rispettivamente il numero di predecessori e il numero di successori di un nodo e detto i il numero di *thread* presenti all'interno del *task-graph*, è possibile esprimere i vincoli sui *thread* come:

$succ(T_x) = i - 2$ se $x = 0$

$prec(T_x) = i - 2$ se $x = i$

$prec(T_x) = 1$ e $succ(T_x) = 1$ se $0 < x < i$

È inoltre possibile definire le funzioni $req(x)$ e $unl(x)$ che restituiscono, rispettivamente, i *thread* richiesti dal *thread* corrente per una corretta esecuzione e i *thread* sbloccati dal *thread* corrente. Con la definizione del *task-graph* è possibile definire una sub-applicazione come una tripla

$$S =< G, FB_{req}, A_i > \tag{3.11}$$

dove:

G=*task-graph* della sub-applicazione;

FB_{req}=byte di requisiti sulle unità funzionali degli elementi processanti;

A_i=lista di acceleratori richiesti dalla sub-applicazione.

In Figura 3.11 viene illustrata la struttura del parametro FB_{req}, che viene visto come una parola composta da K bit all'interno della quale ciascun bit rappresenta la richiesta di una specifica un'unità funzionale. Dato che all'interno dei cluster dell'architettura p2012 vengono utilizzati dei processori STxP70-4, in accordo con quanto descritto all'interno del *datasheet* del-

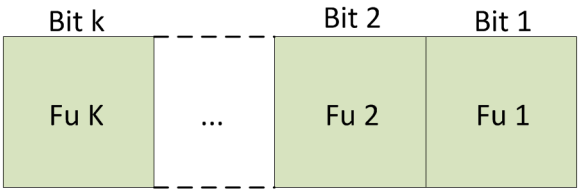


Figura 3.11: Parola di rappresentazione delle unità funzionali delle unità processanti

l'architettura. le unità funzionali di cui si dovrà tener conto per ogni cluster (i processori inseriti in un cluster sono infatti identici) sono :([22])unità Floating Point Unit (FPU), unità Memory Protection Unit (MPU), estensione Vec-x Single Instruction Multiple Data (SIMD), estensione DSP-x, controller DMA interno, controller Eventi, controllore di memoria, controllore delle interruzioni.

Con la definizione del modello di sub-applicazione, il modello di applicazione può essere definito come un grafo orientato aciclico

$$R =< S_i, D_i > \tag{3.12}$$

dove:

S_i contiene l'elenco delle sub-applicazioni che compongono l'applicazione;

D_i contiene i vincoli presenti tra le sub-applicazioni.

Con l'Equazione 3.12 il modello di applicazione è completo. In Tabella 3.3 sono riassunte tutte le varie componenti del modello di applicazione.

In questo capitolo sono stati introdotti i modelli di applicazione e di architettura utilizzati durante lo svolgimento del lavoro presentato in questa tesi andando così a completare il primo obiettivo posto. Il modello di applicazione di basa sull'esigenza di poter modellare l'esecuzione di applicazioni scomponibili in sub-applicazioni eseguibili in parallelo e a loro volta composte da diversi *thread*. Il modello è stato quindi elaborato sulla base di queste esigenze ma anche sulla base del modello dell'architettura p2012 al fine di ottenere un modello effettivamente utilizzabile sulla stes-

Tabella 3.3: Elenco di tutti i parametri descrittivi di un'applicazione

Oggetto	Definizione	Parametri	Significato
Processi	$T =< T_{sw}, A_{req}, T_{hw} >$	$T_{sw}[s]$	Tempo di esecuzione del processo eseguito in hardware
		A_{req}	Acceleratori hardware richiesti dal processo
		$T_{hw}[s]$	Tempo di esecuzione del processo eseguito in hardware
Task-graph	$G =< T_i, E_i >$	T_i	Set dei processi
		E_i	Set dei vincoli tra processi
Sub-applicazione	$S =< G, FB_{req}, A_i >$	G	Task-graph dei processi componenti la sub-applicazione
		FB_{req}	Unità funzionali richieste dalla sub-applicazione
		A_i	Acceleratori hardware richiesti dalla sub-applicazione
Applicazione	$R =< S_i, D_i >$	S_i	Set delle sub-applicazioni
		D_i	Set dei vincoli tra sub-applicazioni

sa. Il modello di architettura è stato quindi elaborato rispecchiando il più fedelmente possibile struttura e funzionamento della piattaforma p2012 di S.T. Microelectronics.

Nel capitolo successivo viene quindi introdotta la strategia di *mapping* che è stata elaborata utilizzando i modelli introdotti in questo capitolo al fine di raggiungere l'obiettivo primario dell'ottimizzazione dell'utilizzo delle risorse presenti sulla piattaforma MPSoC p2012.

Capitolo 4

Strategia di Mapping

Nel Capitolo 3 è stata affrontata la parte di analisi e modellazione dell'architettura p2012 e delle applicazioni che verranno utilizzate su di essa, andando di fatto ad esaurire il lavoro riguardante il primo obiettivo che questo lavoro di tesi si è posto. In questo capitolo, viene introdotto quanto svolto al fine di elaborare una strategia di *mapping* adatta al modello elaborato, che rappresenta il secondo e principale obiettivo del lavoro illustrato in questa tesi.

Data la struttura a due livelli dell'architettura p2012 (vedi Capitolo 2) l'euristica è stata implementata a livello Network-on-Chip (NoC).

Dal punto di vista dell'affidabilità il maggior requisito è rappresentato dal dover bilanciare l'utilizzo delle risorse presenti sulla piattaforma p2012 al fine di evitare l'insorgere del fenomeno dell'invecchiamento precoce dei componenti dovuto ad un loro utilizzo continuo ed intensivo. Al fine di raggiungere questo obiettivo si sono assunti i *cluster* dotati di meccanismi di tolleranza ai guasti correttamente funzionanti, come esplicitato nella Sezione 3.2. Dal punto di vista delle prestazioni invece, il requisito principale richiesto è la minimizzazione dei tempi di esecuzione delle applicazioni che devono essere mandate in esecuzione non appena vi siano sufficienti risorse disponibili e le cui sub-applicazioni devono essere assegnate a cluster

adiacenti. La strategia elaborata deve quindi essere utilizzabile all'interno di un algoritmo di *mapping* e deve soddisfare sia i requisiti di affidabilità che quelli sulle prestazioni.

Questo capitolo articola la descrizione dell'elaborazione di tale strategia in cinque sezioni. Nella **Sezione 4.1** vengono descritti nel dettaglio gli obiettivi che la strategia deve soddisfare, con particolare riferimento all'analisi degli algoritmi svolta nel Capitolo 2. Nelle **Sezioni 4.2 e 4.3** vengono introdotte rispettivamente le metriche adottate al fine di minimizzare i tempi di esecuzione delle applicazioni e di prevenire il fenomeno dell'invecchiamento precoce dei componenti.

Nella **Sezione 4.4** viene quindi introdotta la strategia di *mapping* sviluppata sulla base delle metriche individuate. Chiude il capitolo la **Sezione 4.5** che introduce le modifiche effettuate alle strategie Path Load (PL), Low Energy Consumption - Dependences Neighborhood (LEC-DN) e Random Walk al fine di renderle utilizzabili all'interno del lavoro svolto per poter effettuare i confronti con la strategia sviluppata

4.1 Obiettivi

L'obiettivo primario della strategia sviluppata è quello di riuscire a considerare, oltre ai soliti parametri ottimizzati in letteratura, anche il fenomeno dell'invecchiamento dovuto a un eventuale utilizzo non uniforme delle risorse presenti sul Multi-Processor SoC (MPSoC). In particolare, la gran parte degli algoritmi descritti in letteratura si focalizzano sulla minimizzazione del volume di comunicazione e del consumo energetico (Vedi Tabella 2.1) non considerando l'importante fenomeno dell'invecchiamento e delle conseguenze che esso apporta al MPSoC. Dato il focus del modello di applicazione descritto nella Sezione 3.4, ovvero il calcolo parallelo, in questo lavoro è stata rivolta particolare attenzione alla minimizzazione dei tempi

di esecuzione delle applicazioni e all'introduzione di un metrica che permetta di limitare il fenomeno dell'invecchiamento precoce dei componenti andando ad abbandonare i vincoli riguardanti il consumo di energia. L'obiettivo posto al lavoro è stato quindi quello di ottenere un metodo euristico che possa essere confrontato con quanto presente in letteratura, tenendo in considerazione che il modello di applicazione utilizzato non è di tipo *master/slave*, ma permette alle varie sub-applicazioni di comunicare tra loro con l'unico vincolo che esista un'unica sub-applicazione iniziale. A livello di utilizzo delle risorse, invece, si è posto l'obiettivo di ottenere un utilizzo uniforme delle stesse rispettando così il relativo requisito.

Per questi motivi si è scelto di implementare un algoritmo di *mapping* dinamico e che riservi le risorse necessarie all'intera esecuzione di un'applicazione. L'architettura p2012 impone inoltre un algoritmo di tipo centralizzato ([6]) che sarà eseguito dal *Fabric Controller*. Nel lavoro svolto si è considerata una versione del MPSoC omogenea, ma l'euristica è stata sviluppata tenendo in considerazione la possibile esigenza futura di adattare l'algoritmo ad architetture eterogenee.

4.2 Metriche adottate per minimizzare il tempo di esecuzione di un'applicazione

Uno dei due obiettivi principali della strategia elaborata è la minimizzazione del tempo di esecuzione delle applicazioni. Il modello esecuzione adottato rappresenta un'applicazione come composta da più sub-applicazioni che possono essere eseguite in parallelo sui *cluster* della piattaforma p2012 (vedi Capitolo 3). Questo comporta che due sub-applicazioni che devono comunicare tra loro debbano utilizzare l'infrastruttura di comunicazione del MPSoC. Se le sub-applicazioni sono allocate su due cluster distanti, il tempo necessario alla comunicazione sarà necessariamente più

alto rispetto a quello necessario nel caso un cui le sub-applicazioni siano allocate su *cluster* contigui. Oltre a portare a tempi di comunicazione maggiori, l'allocazione delle sub-applicazioni su *cluster* non contigui potrebbe portare anche al fenomeno della frammentazione delle applicazioni. Tale fenomeno è illustrato in Figura 4.1 e consiste nell'avere le sub-applicazioni appartenenti ad un'applicazione in esecuzione su *cluster* non contigui e sparsi in modo disordinato all'interno del MPSoC (in Figura 4.1 i quadrati con riempimento a righe oblique rappresentano le sub-applicazioni di un'applicazione in esecuzione, quelli con riempimento a quadrati rappresentano i *cluster* liberi mentre quelli con riempimento a rombi rappresentano i *cluster* occupati). La strategia di *mapping* dovrà quindi ovviare a questo inconveniente andando ad allocare le sub-applicazioni componenti un'applicazione su *cluster* contigui, andando così a evitare che i tempi di comunicazione divengano eccessivi e ad evitare il problema della frammentazione. Il problema di minimizzare il tempo di esecuzione può essere quindi ridefinito come l'allocazione delle sub-applicazioni su *cluster* contigui mantenendo una forma dei *clusters* scelti che sia il più regolare possibile, possibilmente rettangolare, al fine di limitare il più possibile il fenomeno della frammentazione.

Si è quindi deciso di effettuare il *mapping* di tutte le sub-applicazioni nel momento in cui un'applicazione chiede di essere eseguita sul MPSoC. Per poter effettuare l'assegnamento correttamente, rispettando il vincolo di allocare le sub-applicazioni su *clusters* contigui, viene introdotto il concetto di "aree di clusters liberi", illustrato in Figura 4.2. Un'area di *clusters* liberi è definita come un insieme di *clusters* contigui non guasti e non occupati da sub-applicazioni. In Figura 4.2 sono rappresentate tre aree di *clusters* liberi che possono essere sfruttate per l'allocazione di sub-applicazioni. Un approccio di questo tipo permette di mantenere ordinata l'allocazione delle sub-applicazioni sui *clusters* evitando un importante effetto collatera-

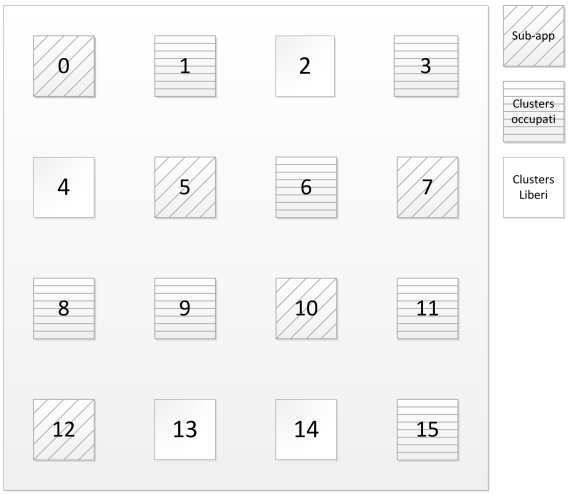


Figura 4.1: Frammentazione di un'applicazione in esecuzione

le della frammentazione che consiste nell'avere i *clusters* liberi sparsi nel MPSoC, fenomeno che obbligherebbe l'applicazione ad attendere un numero di *cluster* liberi contigui sufficiente a contenere tutte le sub-applicazioni ritardandone così l'esecuzione.

Un'applicazione viene quindi assegnata a un'area di *clusters* liberi, precisamente alla più piccola area di *clusters* liberi che possa contenerla. In questo modo il problema del *mapping* delle sub-applicazioni può essere affrontato solo sull'area assegnata all'applicazione. La strategia elaborata affronta quindi l'assegnamento di un'applicazione e delle sub-applicazioni che la compongono attraverso i seguenti passi:

- nel primo passo viene analizzata la situazione del MPSoC e vengono rilevate le aree di *clusters* liberi;
- nel secondo passo viene individuata la più piccola area che possa contenere l'applicazione da eseguire. Nel caso una tale area non esista, l'applicazione verrà messa in attesa in un'apposita coda;
- nel terzo passo viene effettuato il *mapping* delle sub-applicazioni sui *clusters* componenti l'area a cui l'applicazione è stata assegnata.

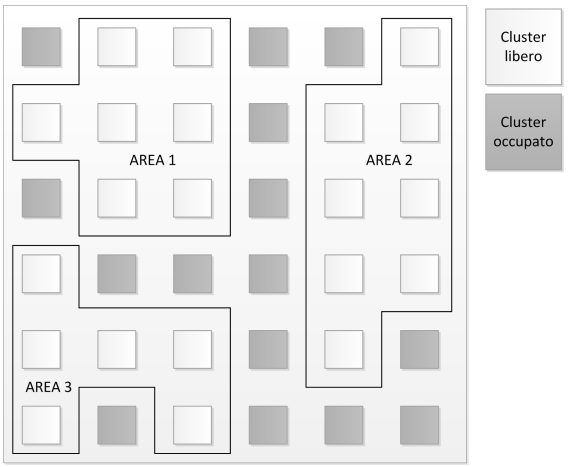


Figura 4.2: Aree di clusters liberi

Nei prossimi paragrafi verranno approfondite le problematiche del rilevamento delle aree di clusters liberi e dell'assegnazione delle sub-applicazioni in quanto essa è critiche per quanto riguarda il buon funzionamento della strategia sia quando l'area assegnata è composta da più *clusters* rispetto al numero delle sub-applicazioni in cui essa è divisa, sia quando si è in una situazione in cui tutti i *clusters* della piattaforma sono liberi.

4.2.1 Rilevamento delle aree di clusters liberi

Al fine di rilevare correttamente le aree di *cluster* liberi è stato elaborato un algoritmo ispirato agli algoritmi di *blob detection* applicati alle immagini. Tale algoritmo, è riportato nel listato 4.1.

```
1  Input: Stato del MPSoC
2  Output: Lista delle aree di cluster liberi
3  begin
4          numRighe=MPSoC.width;
5          numColonne=MPSoC.height;
6          listaCluster=nuovaLista();
7
8          for(i=0; i<numRighe; i++)
9                  for(i=0; i<numColonne; i++)
10                 {
```

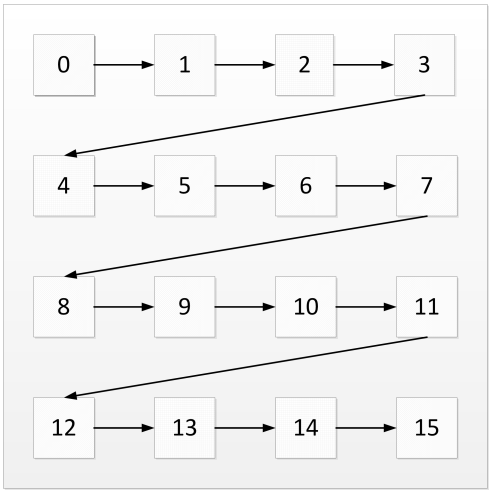


Figura 4.3: Ordine della scansione del MPSoC effettuata dall'algoritmo di rilevazione delle aree di *cluster* liberi

```
11                          if(MPSoC[i][j].occupied)
12                          {
13                                  continue;
14                          }
15                          else
16                          {
17                                  clusterLiberi=verificaAdiacenze();
18                                  if(clusterLiberi)
19                                  {
20                                     listeCluster=
21                                        getClusterList(MPSoC[i][j]);
22                                     lista=listeCluster.merge();
23                                  }
24                                  else
25                                  {
26                                     lista=creaNuovaLista();
27                                  }
28                                  lista.insert(MPSoC[i][j]);
29                                  aggiornaListeCluster(lista);
30                          }
31                  }
32  end
```

Listato 4.1: Pseudo codice per la rilevazione delle aree di *cluster* liberi

L'algoritmo elaborato effettua la rilevazione delle aree libere effettuando un'unica scansione dello stato del MPSoC. Esso lavora calcolando le liste di *cluster* liberi in modo incrementale. L'algoritmo riceve in ingresso i dati sullo stato del MPSoC e restituisce le liste di cluster liberi calcolate. Le prime operazioni effettuate (linee 4 e 5) sono volte a acquisire le dimensioni del MPSoC al fine di effettuare correttamente la scansione dello stesso. Viene quindi inizializzata la lista che conterrà le liste di *cluster* liberi. La scansione del MPSoC viene effettuata dalla linea 8 alla linea 30 del Listato 4.1. La prima operazione effettuata nel ciclo (linea 11) è la verifica dell'utilizzabilità del cluster. Nel caso in cui il cluster non sia utilizzabile si passa alla scansione di quello successivo (linea 13). In caso contrario l'algoritmo procede all'aggiornamento delle liste di *cluster* liberi. Tale aggiornamento viene effettuato andando a vedere quali *cluster* adiacenti a quello selezionato sono utilizzabili (linea 17). Qualora essi siano presenti, il *cluster* corrente permette di unire le aree contenenti i *cluster* liberi adiacenti. Per questo motivo vengono lette le aree contenenti tali *cluster* ed unite (linee 20 e 21) creando così una nuova lista. Nel caso in cui invece nessuno dei *cluster* adiacenti sia disponibile, viene creata una lista vuota (linea 24). Le ultime operazioni effettuate consistono nell'inserimento del *cluster* corrente all'interno della lista individuata e viene quindi aggiornata la lista di aree individuate (linee 27 e 28). A fine esecuzione le liste di cluster liberi saranno contenute all'interno della variabile listeCluster. In Figura 4.3 è rappresentato l'ordine di scansione dell'algoritmo mentre in figura 4.4 viene presentato un esempio di funzionamento dello stesso.

4.2.2 Mapping delle sub-applicazioni su MPSoC scarico

Il primo caso affrontato nello sviluppo dell'assegnamento delle sub-applicazioni ai *cluster* è stato il caso rappresentato dal problema di effettuare il *mapping* a MPSoC scarico, ovvero con tutti i *cluster* disponibili. In

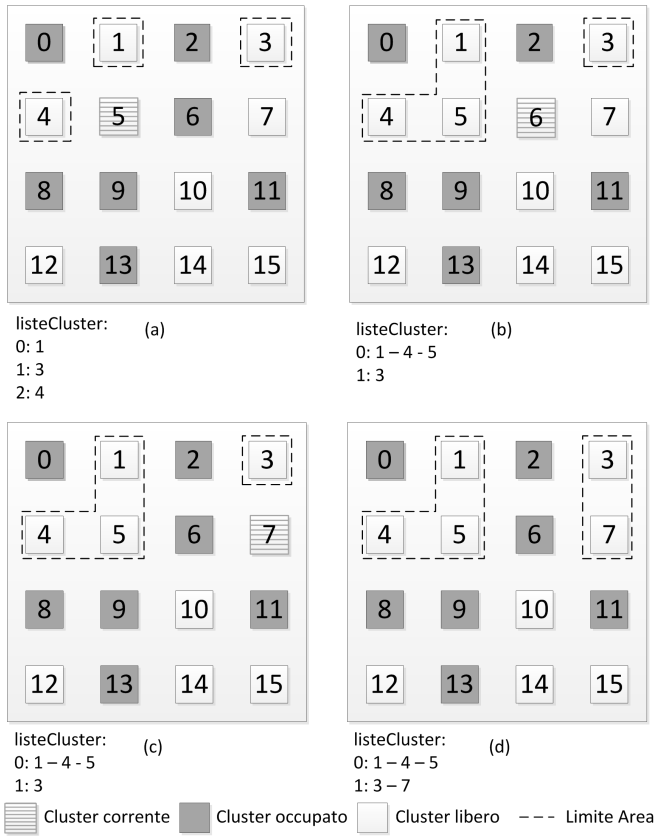


Figura 4.4: Esempio di funzionamento dell'algoritmo di rilevamento delle aree

questo tipo di situazione il rilevamento delle aree non deve essere effettuato e l'assegnamento da effettuare risulta critico al fine di mantenere le forme delle aree il più regolari possibili anche nelle successive situazioni di MPSoC a regime. L'algoritmo di assegnamento sviluppato lavora sul numero di *hops* che separano il *Fabric Controller* dai *clusters* sulla NoC e sul numero di vicini che un *cluster* possiede. In particolare viene scelto come *cluster* iniziale il *cluster* **più vicino**, in numero di *hops*, al *Fabric Controller* e poi vengono annessi ai *clusters* scelti, in alternanza, il *cluster* con il numero minore di vicini e il *cluster* col numero maggiore di vicini. In questo modo si riesce ad ottenere una buona regolarità dell'area selezionata, come mostrato dalle forme ottenute per applicazioni composte da 4 e 8 sub-applicazioni sono mostrate in Figura 4.5.

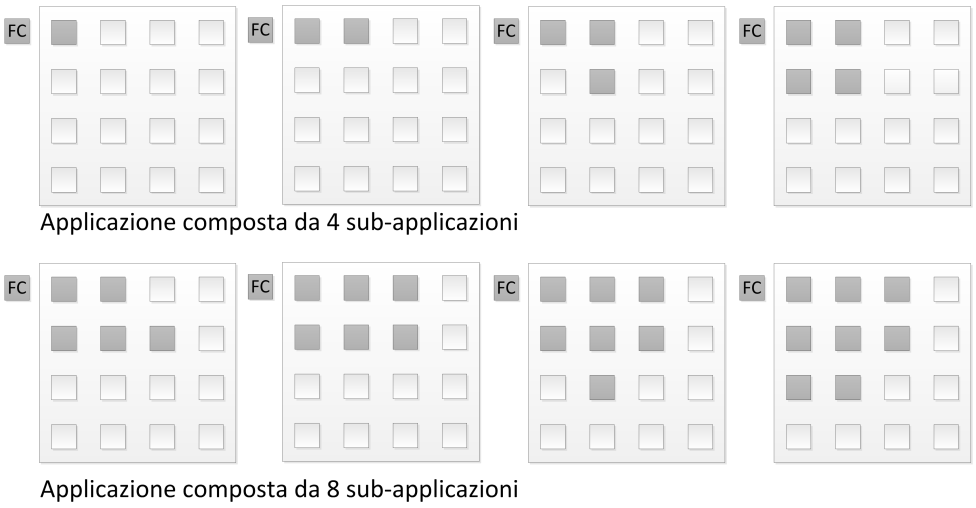


Figura 4.5: Forme ottenute per le applicazioni composte da 4 e 8 sub-applicazioni

In Figura 4.6, invece, sono rappresentati i risultati di alcune prove condotte al fine di determinare la miglior strategia di scelta dei *cluster* a cui assegnare le sub-applicazioni da eseguire. La scelta di selezionare in alternanza il *cluster* con il minor numero di vicini liberi e quello con il maggior numero di vicini liberi deriva proprio da questi test, che hanno mostrato come strategie di scelta più semplici portano all'ottenimento di forme che non permettono il raggiungimento degli obiettivi che il lavoro si è posto.

Scegliendo sempre il *cluster* col minor numero di vicini liberi (Figura 4.6 a), l'assegnamento delle sub-applicazioni tende a seguire il perimetro dell'area individuata, portando le applicazioni composte da un elevato numero di sub-applicazioni a vedersi assegnate le stesse su *cluster* lontani tra loro. Questo comportamento non è accettabile in quanto contrasta con l'obiettivo di minimizzare i tempi di esecuzione delle applicazioni. Dall'altro lato però (Figura 4.6 b), anche il comportamento ottenuto scegliendo sempre il *cluster* col maggior numero di vicini liberi è inaccettabile perchè porta a forme troppo irregolari.

Per questo motivo, dopo aver provato diverse soluzioni, si è giunti alla conclusione di scegliere, in alternanza, il *cluster* con il minor numero di

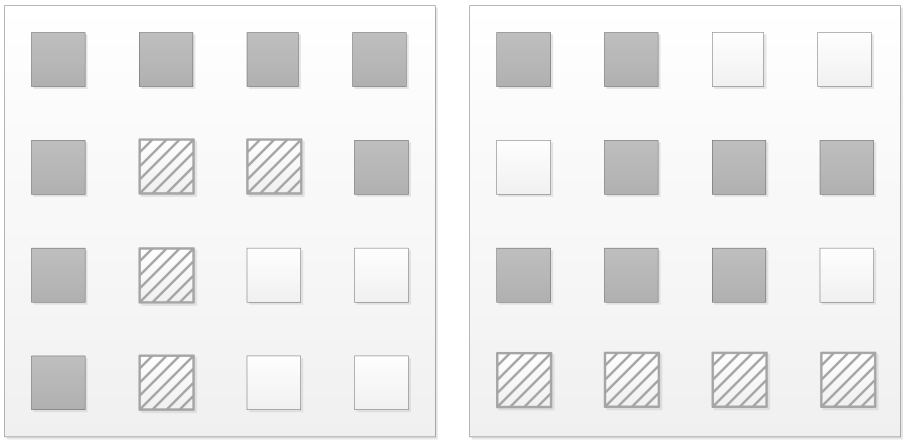


Figura 4.6: Comportamenti di alcuni test effettuati durante lo sviluppo dell'euristica

vicini e quello col maggior numero di vicini, ottenendo i risultati illustrati in Figura 4.5 Il lato negativo di questo approccio risiede nella scelta del *cluster* iniziale.

La scelta di utilizzare il *cluster* più vicino, in termini di *hops*, al Fabric Controller porta infatti a uno squilibrio nell'utilizzo dei *clusters*. Basando la scelta sul numero di *hops* di distanza dal Fabric Controller infatti, il metodo elaborato tende a utilizzare di più i *cluster* ad esso più vicini, portando degli squilibri nell'utilizzo dei *cluster*. Tale comportamento contraddice quindi l'obiettivo della strategia rispetto alla metrica dell'uniformità di utilizzo. Il problema è stato risolto con l'introduzione delle tecniche utilizzate per prevenire il fenomeno dell'invecchiamento come mostrato nella Sezione 4.4.

4.2.3 Mapping delle sub-applicazioni su un'area con System-on-Chip (SoC) parzialmente occupato

Il *mapping* delle sub-applicazioni su un'area rilevata con MPSoC a regime mantiene le stesse regole di quanto visto nella Sezione 4.2.2. La diffe-

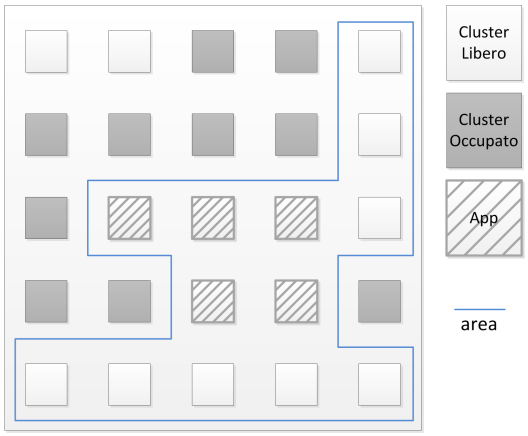


Figura 4.7: Esempio di *mapping* di un'applicazione composta da 5 sub-applicazioni su un'area generica

renza risiede nelle forme ottenute che, benchè rispettino il vincolo riguardo all'adiacenza dei *clusters*, risultano fortemente influenzate dalla forma dell'area individuata che a sua volta dipende dall'evoluzione dell'esecuzione delle applicazioni. Nel caso in cui si abbia un'area di *clusters* liberi sufficientemente grande, in termini di larghezza e altezza (in numero di *clusters*), tale da permettere lo sviluppo di forme regolari, la strategia si comporta esattamente come descritto nella Sezione 4.2.2 confermando l'effettività della soluzione elaborata. Altrimenti, la strategia si adatta all'area rilevata proponendo un assegnamento di forma irregolare. Un esempio di *mapping* di un'applicazione è riportato in Figura 4.7. L'esempio conferma come, laddove vi sia spazio sufficiente, la strategia tenda a generare assegnamenti compatti nonostante la grandezza dell'area di *cluster* liberi e la sua irregolarità.

Nel Listato 4.2 è invece riportato lo pseudo codice di questa prima versione di strategia elaborata.

```
1 Input: Applicazione da allocare, stato del MPSoC
2 Output: Lista dei cluster a cui sono assegnate le sub-appli\-cazioni
3 begin
4         aree=rilevaAree(statoMPSoC);
```

```
5         areaScelta=minArea(applicazione.size, aree);
6         for(i=0; i<numClusters; i++)
7                 if(min(areaScelta[i].distanceFromFC))
8                         primoCluster=areaScelta[i];
9         applicazione.subApplicazioni[0].clusterAssegnato=primoCluster
10
11        i=0
12        while(numeroApplicazioniAssegnate<applicazione.size)
13        begin
14                if(sceltaMin)
15                        clusterScelto=minNumeroVicini(areaScelta);
16                        sceltaMin=false;
17                        sceltaMax=true;
18                if(sceltaMax)
19                        clusterScelto=maxNumeroVicini(areaScelta);
20                        sceltaMax=false;
21                        sceltaMin=true;
22                applicazione.subApplicazioni[i].clusterAssegnato=clusterScelto
23                numeroApplicazioniAssegnate+1;
24        end
25 end
```

Listato 4.2: Pseudo codice per il mapping delle applicazioni

L'algoritmo è molto semplice e fa uso dell'algorimo di rilevamento delle aree libere introdotto nel Paragrafo 4.2.1. Esso riceve in ingresso l'applicazione da assegnare ai *cluster* e lo stato del sistema e restituisce la lista dei *cluster* a cui le sub-applicazioni sono assegnate. Come primo passo (linea 4), viene eseguita la ricerca delle aree di *cluster* liberi e viene scelta l'area rilevata più piccola che può contenere l'applicazione da eseguire (linea 5). Viene quindi scelto come *cluster* iniziale quello più vicino al *Fabric Controller* (linee 6-8) e ad esso viene assegnata la prima sub-applicazione da eseguire (linea 9).

Si entra quindi nel ciclo di assegnamento delle rimanenti sub-applicazioni (linee 12-20), all'interno del quale vengono scelti, in accordo con la strategia elaborata, i *cluster* col minor numero di vicini liberi(linee 15-17) in alternanza a quelli col maggior numero di vicini liberi (linee 19-21);

una volta scelto il *Cluster*, ad esso viene assegnata la sub-applicazione da eseguire e si passa alla sub-applicazione successiva (linee 22 e 23). Una volta assegnate tutte le sub-applicazioni l'algoritmo termina e l'applicazione sarà stata assegnata correttamente.

4.3 Metriche per prevenire il fenomeno dell'invecchiamento

L'euristica derivante da quanto riportato nella Sezione 4.2 soffre del problema dovuto alla scelta del *cluster* iniziale basandosi sulla vicinanaza al *Fabric Controller*. Questa scelta risulta contrastante con l'obiettivo di rendere uniforme l'utilizzo delle risorse presenti sul MPSoC e pertanto deve essere modificata dalle metriche utilizzate per prevenire il fenomeno dell'invecchiamento. Da quanto si evince dall'analisi effettuata nel Capitolo 2, l'invecchiamento di un singolo componente dipende da molti fattori, tra i quali un ruolo di primaria importanza è rappresentato dal tempo di utilizzo di un componente. Pertanto, è lecito supporre che più un componente viene utilizzato e più la sua probabilità di guastarsi aumenta. Per allungare la vita media del componente quindi è necessario che tutti i componenti del SoC vengano utilizzati mediamente per lo stesso tempo in modo tale da evitare che anche uno solo di essi possa essere utilizzato in modo troppo intensivo. È lecito inoltre supporre, non prendendo in considerazione fattori come la *variability* di produzione , che la probabilità di guasto dei componenti presenti nella piattaforma utilizzata sia identica per tutti i *cluster*, in quanto la versione dell'architettura "p2012"presa come riferimento è dotata di *cluster* identici ([6]). Per rendere quindi la strategia in grado di prevenire il fenomeno dell'invecchiamento precoce si è reso necessario modificare la strategia elaborata nella Sezione 4.2. Le modifiche introdotte riguardano principalmente la scelta del *cluster* iniziale e l'introduzione di

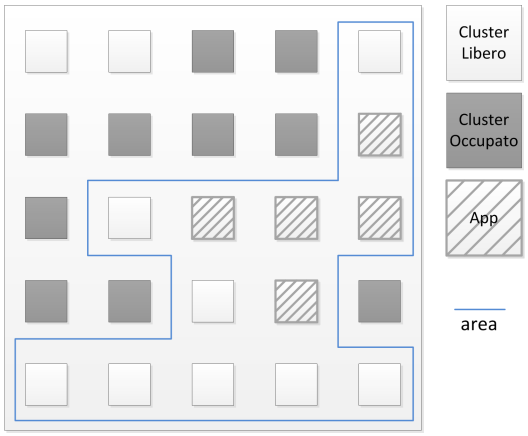


Figura 4.8: Esempio di *mapping* di un'applicazione composta da 5 sub-applicazioni utilizzando anche la metrica di prevenzione dell'invecchiamento precoce

alcune condizioni durante la scelta alternata dei *cluster* con minimo/massimo numero di vicini. Il criterio di scelta dei *cluster* utilizzato per prevenire il fenomeno dell'invecchiamento è il tempo di utilizzo di un singolo *cluster*, inteso come tempo effettivamente impiegato nell'esecuzione delle sub-applicazioni.

Utilizzando questa strategia di scelta il *cluster* iniziale viene quindi selezionato non più utilizzando come metrica il numero di *hops* ma utilizzando il nuovo criterio di scelta. Le condizioni aggiunte invece, vengono verificate allorquando accade che vi siano più *clusters* con lo stesso numero di vicini e si debba scegliere tra essi; in questo caso infatti, la strategia di scelta viene utilizzata per selezionare il *cluster* che, a parità di numero di vicini, possieda un tempo di utilizzo inferiore. L'introduzione di queste condizioni, tende a restituire aree con forme meno regolari ma comunque sufficientemente compatte, come mostrato in Figura 4.8.

4.4 Euristica sviluppata

La strategia di *mapping* finale è illustrata nello pseudo-codice contenuto nel Listato 4.3.

```
1  Input: Applicazione da allocare, stato del MPSoC
2  Output: Lista dei cluster a cui sono assegnate le sub-appli\-cazioni
3  begin
4          aree=rilevaAree();
5          areaScelta=minArea(applicazione.size);
6          min=areaScelta[i].activityTime
7
8          for(i=0; i<numClusters; i++)
9          begin
10                 if(areaScelta[i].activityTime<min)
11                         primoCluster=areaScelta[i];
12         end
13         applicazione.subApplicazioni[0].clusterAssegnato=primoCluster
14         i=0
15         while(numeroApplicazioniAssegnate<applicazione.size)
16         begin
17                 if(sceltaMin)
18                         clustersUtili=minNumeroVicini(areaScelta);
19                         sceltaMin=false;
20                         sceltaMax=true;
21                 if(sceltaMax)
22                         clustersUtili=minNumeroVicini(areaScelta);
23                         sceltaMax=false;
24                         sceltaMin=true;
25
26                 if(clustersUtili.size>1)
27                         clusterAssegnato=min(clustersUtili.activitytime)
28            else
29                         clusterAssegnato=clustersUtili
30                 applicazione.subApplicazioni[i].clusterAssegnato=clusterScelto
31                 i=i+1;
32         end
33 end
```

Listato 4.3: Pseudo codice dell'euristica sviluppata

Nonostante le prove effettuate durante lo sviluppo della strategia abbiano portato allo sviluppo di una soluzione efficace però, essa risulta comunque scarsamente flessibile a causa dell'impossibilità di dare maggiore o minore priorità ad una delle due metriche utilizzate all'interno della stessa. La strategia modificata secondo le nuove condizioni, illustrato nel Listato 4.3, ha gli stessi dati in entrata ed in uscita della versione riportata nel Listato 4.2. Essa compie le stesse operazioni iniziali (linee 4 e 5) per rilevare le aree di *cluster* liberi. La prima differenza con la soluzione precedente è nella scelta del cluster iniziale. Secondo quanto sviluppato infatti, essa viene ora effettuata andando a scegliere il *cluster* appartenente all'area selezionata col minor tempo di attività (linee 6-13). Al *cluster* così selezionato viene quindi assegnata la prima sub-applicazione (linea 14). Le linee dalla 16 alla 35 contengono il ciclo di assegnamento delle rimanenti sub-applicazioni. La scelta dei *cluster* in alternanza viene effettuata in modo analogo a quanto fatto nel Listato 4.2 (linee 19-26); le differenze invece risiedono nelle linee da 28 a 33, dove viene scelto il *cluster* da utilizzare qualora vi fossero più *cluster* candidati. La scelta viene dunque effettuata coerentemente con la strategia sviluppata, andando a scegliere il *cluster* tra quelli candidati con minor tempo di attività. Una volta assegnate tutte le sub-applicazioni il ciclo si conclude e l'algoritmo termina.

4.5 Adattamenti alle euristiche esistenti

Durante lo sviluppo dell'euristica descritta nella Sezione 4.4, si è anche proceduto ad adattare opportunamente le euristiche utilizzate come confronto, al fine di renderle compatibili col modello di applicazione introdotto nel Capitolo 3. I dettagli delle modifiche sono discussi nei paragrafi seguenti: nel **Paragrafo 4.5.1** vengono introdotti gli adattamenti effettuati all'euristica PL, nel **Paragrafo 4.5.2** vengono invece discusse le modifi-

che apportate all'euristica LEC-DN. L'adattamento dell'euristica Random Walk (RW) è invece introdotto nel **Paragrafo 4.5.3**

4.5.1 Modifiche all'euristica PL

L'euristica PL, introdotta nel Paragrafo 2.1.5, utilizza un modello di applicazione di tipo master/slave e un modello di MPSoC analogo a quello utilizzato dall'euristica sviluppata in questo lavoro. Per poter utilizzare correttamente l'euristica si è reso dunque necessario adattarla al modello di applicazione introdotto nel Capitolo 3 e modificare la fase di assegnamento iniziale del processo master. Inoltre, a differenza di quanto introdotto in [11], [12] e [13], l'euristica lavora a livello di sub-applicazioni anzichè a livello di processi. Per quanto riguarda l'adattamento al **modello di applicazione**, si è scelto di segnalare come sub-applicazione *master* la prima sub-applicazione che deve essere eseguita, mentre, per quanto riguarda l'**assegnamento del processo master**, si è scelto di assegnare, coerentemente con l'euristica sviluppata, la sub-applicazione al *cluster* con minor tempo di attività.

4.5.2 Modifiche all'euristica LEC-DN

L'euristica LEC-DN ha richiesto meno adattamenti rispetto all'euristica PL in quanto sia il modello di applicazione che le metriche utilizzate dall'euristica stessa, risultano compatibili con il modello di applicazione sviluppato per l'euristica oggetto di questo lavoro di tesi. In particolare, il modello di applicazione utilizzato da tale euristica risulta identico a quanto introdotto nel Capitolo 3, pertanto l'unica modifica effettuata riguarda la scelta del *cluster* a cui assegnare una sub-applicazione. La modifica effettuata riguarda l'utilizzo della tecnica Nearest Neighbor (NN) laddove vi sia da allocare una sub-applicazione che richiede di comunicare con una sola altra sub-applicazione. Nell'algoritmo originale infatti, la scelta veni-

va fatta con la tecnica NN mentre, nella versione implementata, qualora risultino più *clusters* disponibili a pari distanza, viene scelto quello con minor tempo di attività. Tale modifica è stata apportata anche nel caso in cui una sub-applicazione comunichi con almeno altre due sub-applicazioni. La condizione in questo caso viene verificata allorquando vengano trovati due *clusters* con identico volume di traffico. Benchè tale condizione si verifichi molto raramente, questo accorgimento è stato comunque implementato per evitare l'insorgere di situazioni impreviste.

4.5.3 Modifiche all'euristica RW

L'euristica RW ha richiesto un consistente lavoro di adattamento per quanto riguarda il modello di applicazione; esso infatti risulta radicalmente diverso dal modello introdotto nel Capitolo 3 in quanto l'euristica RW è stata ideata per applicazioni strutturate ad albero e suddivise in processi di comunicazione ed elaborazione. Tale tipo di modello identifica dunque il processo radice dell'albero come processo master e l'euristica prevede l'assegnamento di tale processo a un Processing Element (PE) che successivamente ha il compito di eseguire il *mapping* di tutti i processi slave. L'implementazione di tale euristica lavora, come nei casi precedenti, su sub-applicazioni e su *clusters*. Non viene effettuata alcuna distinzione tra processi di comunicazione e processi di elaborazione e viene indicata come sub-applicazione master la prima sub-applicazione dell'applicazione che viene eseguita. La fase di assegnamento della sub-applicazione master ad un *cluster* viene quindi effettuata utilizzando come metrica il tempo di attività dei *clusters* scegliendo quello con il minor valore in assoluto. La fase di assegnamento delle successive sub-applicazioni viene invece effettuata seguendo l'algoritmo originale (vedi Paragrafo 2.1.8) con l'aggiunta, qualora vi siano più *clusters* candidati per l'esecuzione di una sub-applicazione, della condizione che permette di scegliere il *cluster* con tempo di attività

minore.

In questo capitolo è stata introdotta la strategia di *mapping* sviluppata al fine di raggiungere l'obiettivo principale del lavoro presentato in questa tesi. Tale strategia è stata sviluppata in modo incrementale andando a coprire un requisito alla volta. Il lavoro effettuato ha dato buoni risultati in termini di compattezza e regolarità della forma delle aree ottenute in fase di assegnamento rispettando tutti i vincoli posti in fase di progettazione della strategia.

Nel capitolo successivo verranno introdotti il simulatore utilizzato per svolgere i test sulla strategia sviluppata e i test effettuati al fine di validare il funzionamento della stessa.

Capitolo 5

Risultati

Nei Capitoli 3 e 4 sono stati introdotti, rispettivamente, i modelli di architettura e di applicazione (Capitolo 3) e la strategia di *mapping* elaborata (Capitolo 4). In questo capitolo vengono introdotti i test effettuati al fine di validare sia i modelli che la strategia di *mapping* ed i risultati ottenuti. Tali test sono stati effettuati implementando un simulatore dell'architettura, che ha reso possibile sia la verifica del funzionamento della strategia di *mapping*, tramite l'utilizzo dei modelli di applicazione ed architettura elaborati, che il confronto con le tre strategie di *mapping* individuate nel Capitolo 2 ovvero Path Load (PL), Low Energy Consumption - Dependences Neighborhood (LEC-DN) e Random Walk (RW).

Al fine di coprire esaurientemente il lavoro fatto per verificare i modelli e la strategia elaborata, il capitolo è strutturato in sei Sezioni:

- nella Sezione 5.1 viene introdotto il simulatore dell'architettura implementato al fine di eseguire i test. In Appendice A ne viene fornita invece una descrizione dettagliata;
- nella Sezione 5.2 vengono descritti gli esperimenti effettuati per verificare il corretto funzionamento della strategia di *mapping* sviluppata in questo lavoro di tesi;

- nella Sezione 5.3 l'euristica viene confrontata, in assenza di guasti, con le euristiche PL, LEC-DN e RW utilizzando il programma *Bench-Mark* e simulando l'esecuzione di 100, 500 e 1000 applicazioni;
- nella Sezione 5.4 vengono introdotti i risultati dei medesimi esperimenti descritti nella Sezione 5.3 effettuati introducendo, rispettivamente, 2, 4 e 8 guasti permanenti all'interno dell'architettura;
- nella Sezione 5.5 viene analizzato, tramite l'utilizzo del Simulatore, il comportamento dell'euristica sviluppata in presenza di guasti transitori nell'architettura;
- nella Sezione 5.6 vengono, infine, analizzati in modo critico i risultati ottenuti nei test effettuati.

5.1 Simulatore

In questo paragrafo viene presentato il simulatore implementato per permettere la verifica dei modelli creati ed il confronto tra la strategia di *mapping* e le strategie esistenti. Una descrizione dettagliata del simulatore è riportata in Appendice A, all'interno della quale è possibile vedere nel dettaglio come il programma è stato implementato.

La decisione di implementare un simulatore dell'architettura è stata presa in quanto, all'inizio del lavoro di tesi, non vi era a disposizione alcun tipo di simulatore dell'intera architettura p2012.

Èsso è stato quindi implementato al fine di effettuare simulazioni a livello comportamentale. L'implementazione è stata effettuata utilizzando il linguaggio C++ arricchito con la libreria SystemC e dalla sua estensione SystemC-TLM, che aggiunge il supporto al Transaction Level Modeling (TLM) necessario al fine di ottenere una corretta implementazione di una simulazione comportamentale.

Dal simulatore è stato poi derivato un secondo programma, chiamato *Benchmark*, che permette l'esecuzione di test comparativi tra la strategia sviluppata e le tecniche individuate per il confronto nel Capitolo 2. I dettagli dell'implementazione del *Benchmark* sono riportati all'interno dell'Appendice A.

5.1.1 Struttura generale

Il simulatore è stato impostato in modo tale da permettere anche la simulazione di architetture con un numero differente di risorse, in termini di numero di *clusters* presenti sul dispositivo da simulare, rispetto alla versione dell'architettura p2012 presa come riferimento e descritta nel dettaglio in [6]. Per questo motivo, e per rendere possibile la specifica del comportamento dell'architettura in termini di guasti e applicazioni eseguite durante la simulazione, la struttura data all'implementazione del simulatore è quella mostrata in Figura 5.1.

Il simulatore risulta quindi composto da tre parti principali:

- una parte che ha il compito di leggere i file forniti in input al simulatore e di inizializzare le strutture dati sulla base di quanto contenuto nei file (parte "*Architecture and programs description*");
- una parte contenente l'implementazione in System-C dei componenti facenti parte dell'architettura (parte "*Architecture implementation*");
- Una parte contenente il motore del simulatore che, utilizzando le due parti precedenti, si occupa di eseguire la vera e propria simulazione.

Le tre parti del programma interagiscono tra loro andando a formare il flusso di esecuzione mostrato in Figura 5.2.

I passi eseguiti dal programma sono semplici ed il primo di essi consiste nella lettura dei files contenenti le informazioni riguardanti la struttura dell'architettura da simulare, la struttura delle applicazioni che verranno

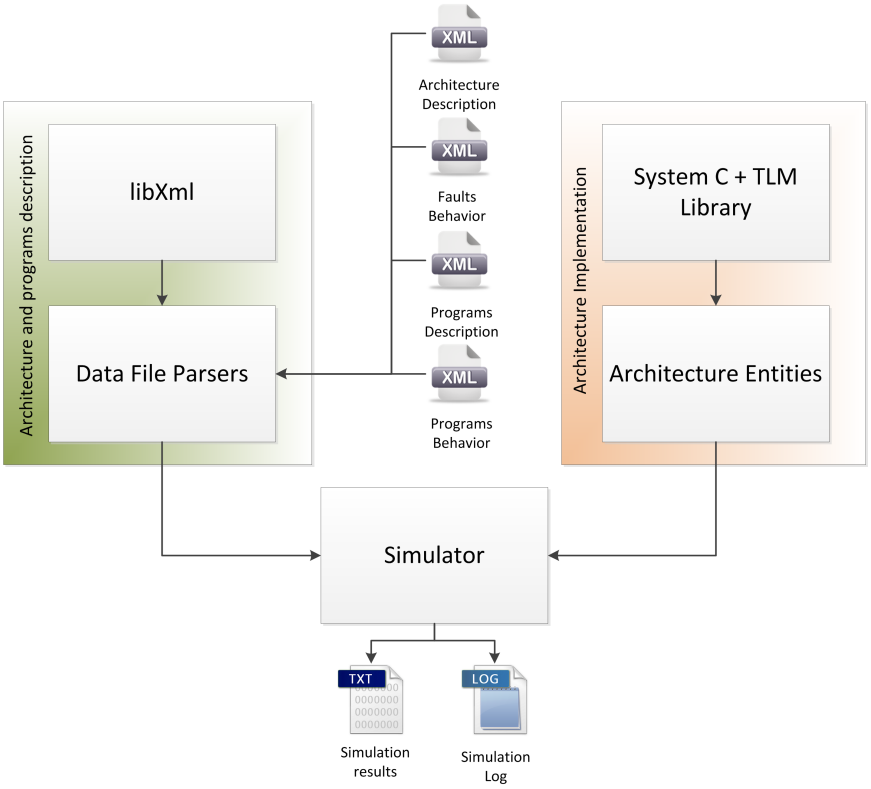


Figura 5.1: Struttura generale del simulatore

eseguite (è anche possibile attivare un generatore casuale di applicazioni interno al simulatore) e il comportamento dei guasti. Una volta lette le informazioni, vengono quindi inizializzate le strutture dati interne e viene creata l'istanza dell'architettura che verrà utilizzata nel passo successivo per la simulazione dell'esecuzione delle applicazioni. Durante l'esecuzione della simulazione vengono raccolte tutte le informazioni necessarie alla creazione dei file di output, che rappresenta l'ultima operazione effettuata dal programma. Per poter specificare i files dati da utilizzare, è possibile utilizzare, nella linea di comando del programma, i parametri riportati nella Tabella A.1 riportata in Appendice A.

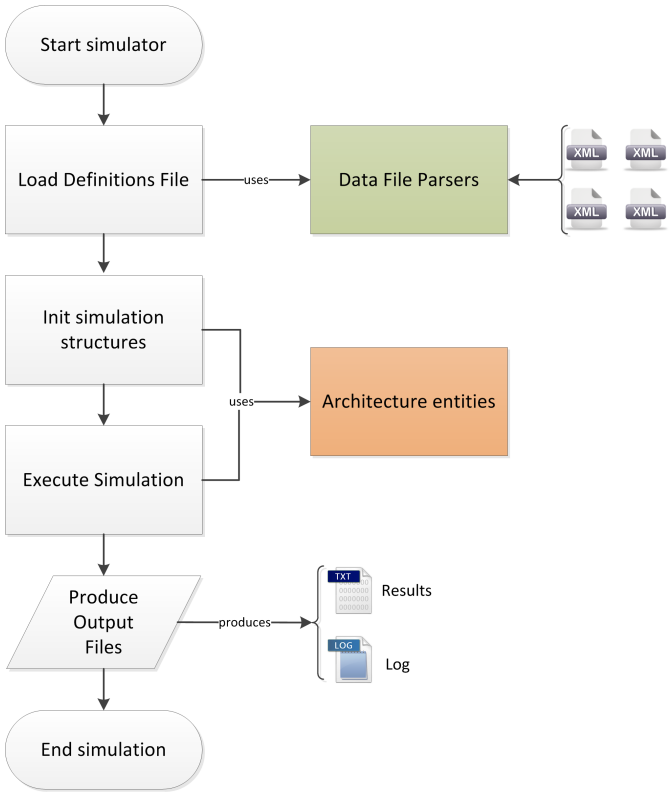


Figura 5.2: Flusso di esecuzione del simulatore

5.1.2 Architettura SystemC

Mentre nel Paragrafo 5.1.1 è stato introdotto il funzionamento del simulatore, in questo paragrafo vengono presentate le entità SystemC implementate per simulare i vari componenti della piattaforma secondo quanto descritto in [6] e modellizzato nel Capitolo 3. Il modello SystemC dell'architettura è stato implementato utilizzando l'estensione TLM, che permette di modellare le comunicazioni tra entità operando a livello di transazione e consentendo quindi di realizzare un simulatore a livello comportamentale.

In Figura 5.3, è riportata il modello SystemC del "livello Network-on-Chip (NoC)"dell'architettura.

Oltre ai componenti individuati nel Capitolo 3, nell'implementazione System-C del livello NoC dell'architettura è presente un nuovo elemento,

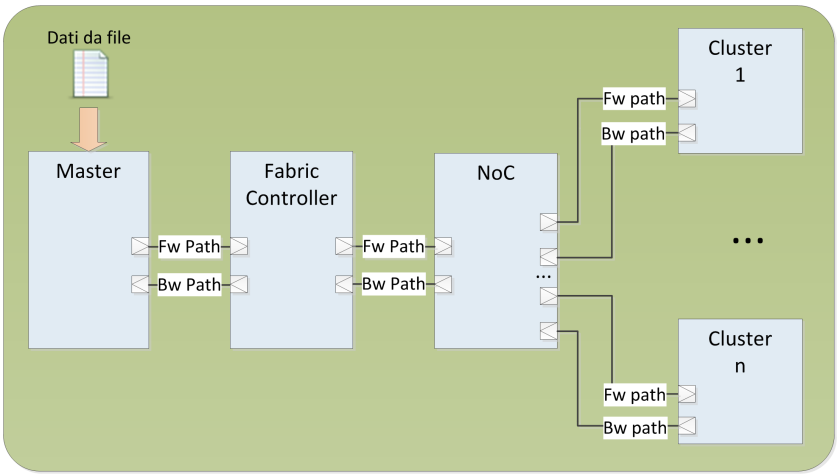


Figura 5.3: Struttura delle entità System-C

necessario al fine della corretta gestione della simulazione: il *master*. Tale entità è responsabile della gestione della simulazione e pertanto risulta il punto di ingresso dei dati letti dai files forniti in *input* all'interno della simulazione. Il compito del *master* non è solo quello di effettuare l'*issue* delle applicazioni nel momento in cui esse vanno eseguite, ma anche quello di gestire la simulazione dei guasti all'interno dei componenti dell'architettura e di raccogliere tutte le statistiche di esecuzione della simulazione al fine di redarre un file di *report* su quanto accaduto durante l'esecuzione della simulazione stessa. Da quanto descritto nel Capitolo 3, la gestione del System-on-Chip (SoC) p2012 è affidata al *fabric controller*, pertanto il *master* è stato collegato solo ad esso. Il *fabric controller* è poi collegato alla NoC che a sua volta possiede i collegamenti con tutti i clusters presenti all'interno dell'architettura. I collegamenti tra le entità sono realizzati con una coppia di *socket*:

- un *socket* di tipo *initiator*, per le comunicazioni in uscita (fwpath in Figura 5.3);
- un *socket* di tipo *target*, per le comunicazioni in entrata (bwpath in Figura 5.3).

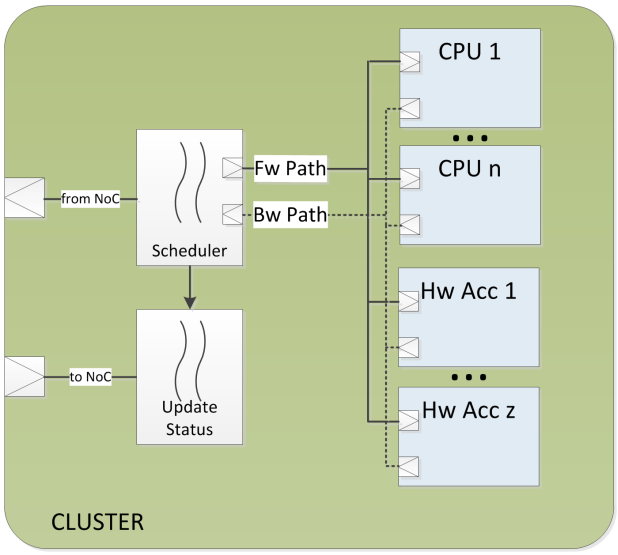


Figura 5.4: Struttura interna dell'entità di tipo *cluster*

Ogni entità contiene al suo interno diversi *thread*, responsabili della gestione dei compiti che quell'attività ha all'interno dell'architettura. Le entità di tipo *Cluster*, rispetto a *master*, NoC e *Fabric Controller*, hanno una struttura più complessa e contengono, al loro interno, la modellazione SystemC dei *cluster*, così come raffigurata in Figura 5.4.

Anche l'entità *cluster* è stata modellata secondo quanto emerso dall'analisi fatta nel Capitolo 3. Ogni singolo *cluster* contiene al suo interno un numero di CPU pari a N_{CPU} e un numero di acceleratori hardware pari a N_{AA} (vedi Tabella 3.1). Sia gli acceleratori hardware che le CPU sono a loro volta delle entità SystemC che vengono gestite dal *thread* implementante lo *scheduler* con le stesse modalità di comunicazione utilizzate a livello *cluster*. Il thread di *scheduling*, inoltre, si occupa di risvegliare il *thread* di aggiornamento dello stato del *cluster* ("*Update status*"in Figura 5.4) che, utilizzando il Bw-path, invia al *fabric controller* le informazioni relative allo stato di avanzamento dell'applicazione in esecuzione sul *cluster* e le informazioni aggiornate relative allo stato delle risorse presenti all'interno dello stesso.

Il funzionamento della simulazione, sulla base delle entità introdotte,

può essere quindi riassunto nel modo seguente:

- il *master* invia i dati di un'applicazione o di un guasto dal fabric controller;
- il *fabric controller* riceve i dati: se si tratta di un guasto allora inoltra le informazioni sul guasto al *cluster* che contiene il Processing Element (PE) interessato dal guasto tramite la NoC, altrimenti procede con il *mapping* dell'applicazione ricevuta;
- una volta effettuato il *mapping* dell'applicazione, i dati delle sub-applicazioni vengono inviati ai *cluster* selezionati dalla strategia di *mapping* tramite NoC;
- la NoC, ad ogni messaggio ricevuto, legge la destinazione e inoltra il messaggio al corretto destinatario;
- il *cluster*, nel caso vengano ricevute informazioni su guasti da applicare al suo interno procede con l'aggiornamento dei dati e invia un messaggio di stato delle risorse, tramite NoC, al *fabric controller*;
- nel caso invece venga ricevuta una sub-applicazione, lo *scheduler* si occupa di gestire correttamente l'esecuzione dei processi che compongono la sub-applicazione. Ogni volta che un processo viene completato, vengono inviati eventuali messaggi di avviso di completamento del processo ai *cluster* che contengono sub-applicazioni i cui processi sono bloccati dal processo appena eseguito. Una volta terminata l'esecuzione viene inviata una notifica al *fabric controller*;
- il *cluster controller* riceve la notifica di esecuzione della sub-applicazione, verifica lo stato dell'applicazione a cui la sub-applicazione appartiene e, nel caso in cui un'applicazione risulti completamente eseguita, invia un messaggio di corretta esecuzione al *master*, che si occupa di rilevare le statistiche dell'applicazione eseguita;

- il ciclo viene ripetuto fino a quando il *master* non riceve notifica di corretta esecuzione di tutte le applicazioni previste dalla simulazione.

All'interno della struttura SystemC così definita sono stati quini implementati i modelli di *scheduling* e *mapping* introdotti nei Capitoli 3 e 4.

Per quanto riguarda la parte relativa allo *scheduling* dei *thread* delle sub-applicazioni, esso è stato implementato all'interno delle entità *Cluster* secondo il modello introdotto nella Sezione 3.4. La strategia di *mapping* elaborata nel Capitolo 4, unitamente alle strategie utilizzate per il confronto ed opportunamente modificate come descritto nella Sezione 4.5, è stata invece implementata all'interno dell'entità *fabric controller*.

5.2 Casi di studio effettuati per verificare il corretto funzionamento della strategia sviluppata

L'implementazione del simulatore introdotto nella Sezione 5.1 ha reso possibile lo svolgimento dei test per verificare il comportamento della strategia di *mapping* elaborata in questo lavoro di tesi.

La prima parte degli esperimenti è stata condotta per verificare il corretto funzionamento della strategia di *mapping*. Per verificare il corretto funzionamento di quanto introdotto nel Capitolo 4, sono stati condotti tre distinti esperimenti:

- un esperimento volto a verificare il corretto funzionamento della scelta delle aree in cui allocare le applicazioni;
- un esperimento volto a verificare l'effettiva uniformità di utilizzo dei clusters appartenenti al SoC;
- un esperimento volto a verificare l'effettivo funzionamento dei meccanismi di sincronizzazione tra processi appartenenti a diverse sub-applicazioni e legati da vincoli di precedenza.

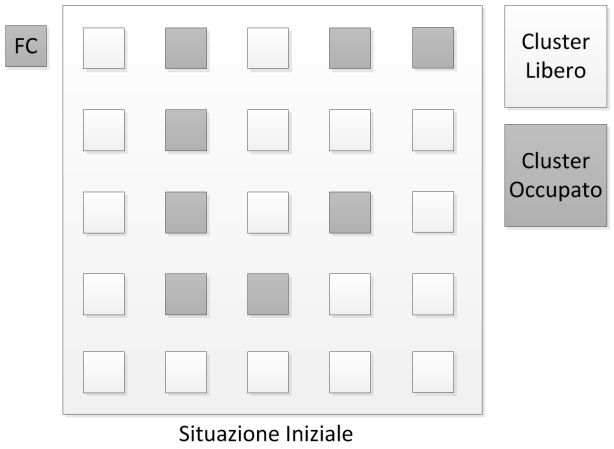


Figura 5.5: Situazione iniziale del primo esperimento generata dal Simulatore

5.2.1 Primo Esperimento

Il primo esperimento è stato svolto utilizzando il Simulatore. Sono state fornite allo stesso dieci applicazioni e si è utilizzata una configurazione del SoC composta da 25 *clusters* identici. Lo scopo dell'esperimento è di verificare che le applicazioni vengano mappate sul SoC secondo la metodologia descritta nel Capitolo 4. Il SoC è stato inoltre pre-inizializzato andando a inizializzare in modo randomico i tempi di attività per ciascun *cluster* e andando a simulare l'occupazione di alcuni degli stessi in modo da effettuare i test partendo da una possibile situazione di carico del dispositivo. In Figura 5.5 viene quindi illustrata la situazione iniziale generata dal simulatore e dalla quale l'esperimento è partito. L'evoluzione dell'intero esperimento è invece riportata in Figura 5.6. All'arrivo di ogni applicazione è stato riportato lo stato del SoC immediatamente precedente il *mapping* della stessa e lo stato immediatamente successivo, per permettere di verificare che l'applicazione venga assegnata correttamente all'area più piccola che la può contenere. Quanto mostrato in Figura 5.6 conferma il corretto funzionamento dell'euristica per quanto concerne la scelta delle aree all'interno delle quali allocare le applicazioni.

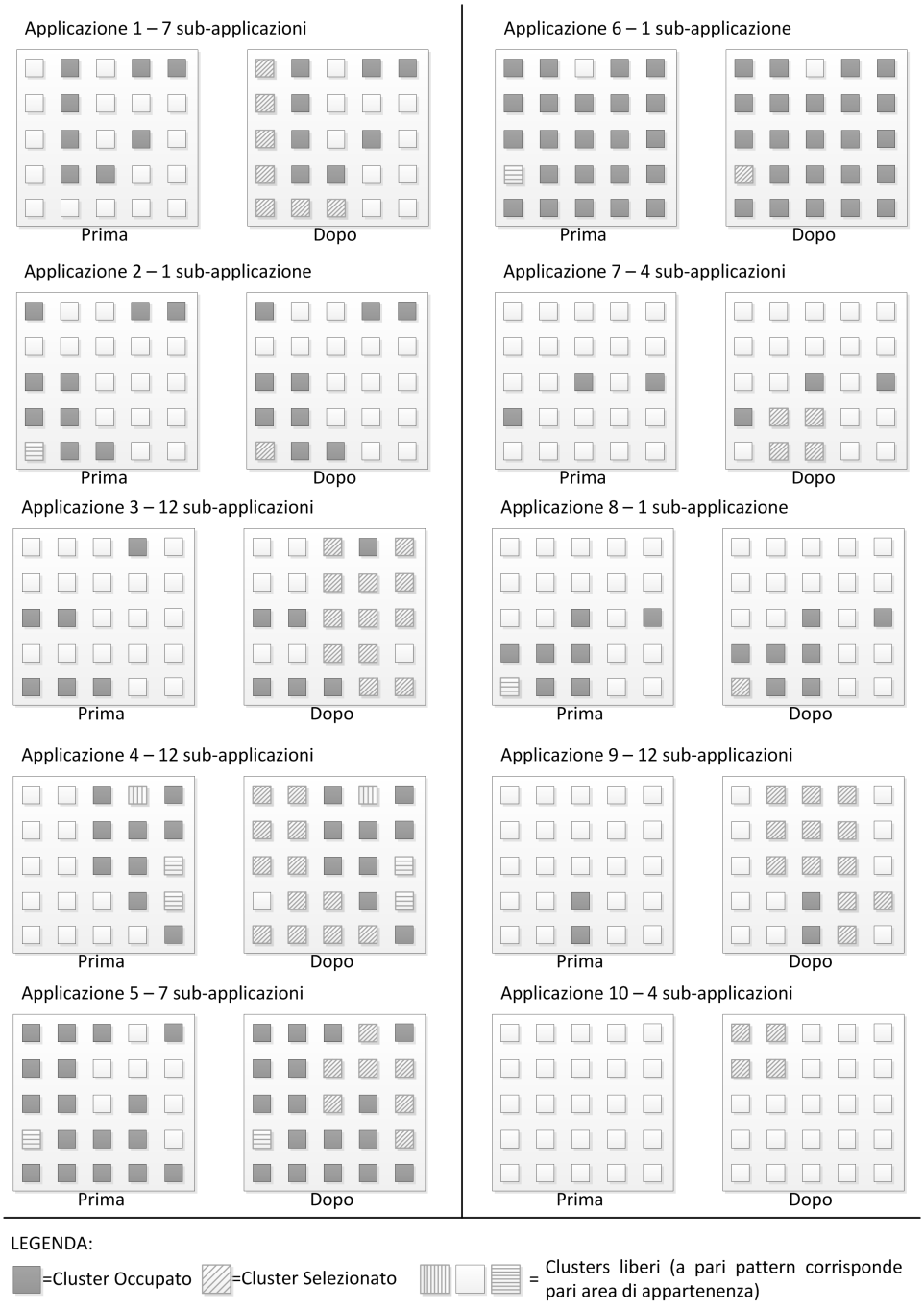


Figura 5.6: Evoluzione del primo esperimento

L'allocazione dell'applicazione 2 e delle applicazioni 5 e 8 confermano la scelta dell'area più piccola che può contenere un'applicazione. L'allocazione delle applicazioni 4,9 e 10 conferma, invece, come in una situazione di sistema scarico, pur mantenendo come obiettivo primario l'uniformità dell'utilizzo dei *cluster*, si riescano ad ottenere allocazioni con forma regolare. Particolarmente significativo è il caso dell'applicazione 10, che viene allocata in maniera ideale.

Sono ovviamente presenti anche casi in cui la forma ottenuta è molto irregolare, come ad esempio nell'allocazione dell'applicazione 1 dove i *cluster* con tempo di utilizzo minore risultano essere quelli lungo i bordi e la particolare forma dell'area di *cluster* liberi impedisce uno sviluppo regolare dell'assegnamento.

5.2.2 Secondo Esperimento

Verificato il corretto funzionamento della scelta delle aree con l'esperimento descritto nel Paragrafo 5.1.1, è stato impostato un secondo esperimento per verificare l'efficacia delle metriche volte a mantenere un utilizzo uniforme dei *cluster*. Per ricavare questi dati sono state eseguite tre simulazioni utilizzando il programma *BenchMark* (vedi Appendice A). Nella prima simulazione sono state eseguite 100 applicazioni, nella seconda 500 e nella terza 1000. Tutte le applicazioni sono state inizializzate in modo randomico e per ogni simulazione sono stati rilevati i tempi di attività di ciascun *cluster*, la media dei tempi di attività e la deviazione standard dei tempi di occupazione in percentuale rispetto alla media di esecuzione. I dati statistici ottenuti sono riportati in Tabella 5.1, mentre i tempi finali dei *clusters* sono riportati in Tabella 5.2. Il tempo di esecuzione di una singola applicazione viene calcolato come $T_{exec} = T_{end} - T_{arrival}[ms]$ e quindi, detta $A_{i,j}$ una generica applicazione iin esecuzione sul *cluster* j, e detto $T_{i,j}$ il suo tempo di esecuzione, il tempo di attività totale di un generico *cluster*

Tabella 5.1: Risultati statistici del secondo esperimento

Numero di applicazioni	**Tempi medi esecuzione [ms]**	σ%
100 Applicazioni	3272	6,32
500 Applicazioni	15220	2,04
1000 Applicazioni	27913	0,88

j può essere calcolato come indicato nell'equazione 5.1.

I risultati mostrati in Tabella 5.1 confermano il buon funzionamento della strategia anche dal punto di vista della distribuzione del carico di lavoro su tutti i *clusters* del SoC. Il dato più importante riguarda il constante calo della deviazione standard percentuale rispetto alla media dei tempi di attività dei *cluster* che indica che maggiore è il numero di applicazioni eseguite e più l'uso dei *cluster* risulterà uniforme.

$$T_{\substack{Exec \\ j}} = \sum_{i} A_{i,j}[ms] \tag{5.1}$$

I dati in Tabella 5.1 confermano inoltre i dati dei tempi di attività finali riportati in Tabella 5.2, mostrando come effettivamente, con l'aumentare del numero di applicazioni, il tempo di attività dei *cluster* tende ad essere molto vicino al tempo medio di attività, segno di una distribuzione uniforme del carico di lavoro.

Di particolare rilevanza risulta inoltre il dato della deviazione standard calcolata durante l'esecuzione di 1000 applicazioni che risulta essere dello 0,88% rispetto alla media dei tempi di attività.

5.2.3 Terzo esperimento

Il terzo esperimento del primo gruppo è stato effettuato con lo scopo di andare a verificare l'effettivo funzionamento dei meccanismi di sincronizzazione durante l'esecuzione di un'applicazione composta da due

Tabella 5.2: Uptime finali dei clusters dopo l'esecuzione del secondo esperimento

Cluster	**Uptime 100 App. [ms]**	**Uptime 500 App. [ms]**	**Uptime 1000 App. [ms]**
1	3267	15044	28026
2	2991	14913	27447
3	3279	14947	27903
4	3298	15588	27716
5	3366	14876	27632
6	3131	15270	27835
7	3368	15493	27829
8	3363	15135	27583
9	3383	14757	27926
10	2949	15132	27573
11	3100	15230	27675
12	3078	15302	27842
13	3202	15110	27762
14	3165	15087	27677
15	2912	14666	28022
16	3052	15506	27892
17	3114	15307	27828
18	3093	15405	27904
19	3188	14998	28052
20	3218	14955	27793
21	3424	14429	28131
22	3016	15289	27632
23	2863	14657	28120
24	2847	15185	27198
25	3415	15349	27930
Tempi Medi	3178	151052	27807

sub-applicazioni con dipendenze tra *thread*. In particolare, è stata simulata, mediante il Simulatore dell'architettura, l'esecuzione di un'applicazione composta da 2 sub-applicazioni i cui *task-graph* sono riportati in Figura 5.7

I processi appartenenti alle due sub-applicazioni sono stati inoltre dimensionati, in termini di durata e dipendenze, in modo tale da evitare

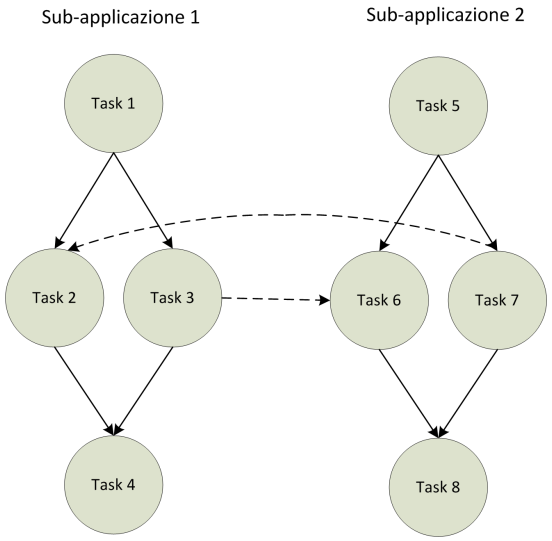


Figura 5.7: Task-graph dell'applicazione utilizzata per il terzo esperimento del primo gruppo

deadlock, ovvero il fenomeno di blocco dell'applicazione dovuto alla mutua attesa di due *thread*. Al fine di verificare il corretto funzionamento della sincronizzazione è stato esaminato il *file* di log prodotto dal simulatore, ricercando i messaggi prodotti dall'entità *cluster* qualora vengano rilevati problemi di dipendenze non soddisfatte che portano all'impossibilità di eseguire un processo. In particolare sono stati rilevati i tempi di fine esecuzione dei *thread* 1 e 5, i tempi di inizio e fine esecuzione dei *thread* 3,4,6 e 7 e i tempi di inizio esecuzione dei processi 4 e 8. Tali tempi sono riportati in tabella 5.3 e confermano il corretto funzionamento del simulatore per quanto riguarda i meccanismi di sincronizzazione tra processi.

I risultati mostrati in tabella mostrano come sia il *thread* 2 della sub-applicazione 1 che il *thread* 7 della sub-applicazione 2 vengano correttamente eseguiti solo dopo che tutti i *thread* da cui dipendono sono terminati. L'eventuale ritardo che compare tra il tempo finale di un *thread* bloccante e il tempo iniziale dei *thread* sbloccati da esso, presente qualora *thread* bloccante e *thread* sbloccati appartengano a sub-applicazioni diverse, dipende

Tabella 5.3: Tempi di esecuzione dei thread dell'applicazione utilizzata durante il terzo esperimento del primo gruppo

Id Thread	Istante iniziale [ms]	Istante finale [ms]	Sub-applicazione
1	20	53	1
2	88	254	1
3	55	86	1
4	256	307	1
5	30	45	2
6	97	264	2
7	47	78	2
8	266	371	2

dal tempo di trasferimento dei dati attraverso la NoC.

5.3 Confronto tra la strategia sviluppata e altre strategie di mapping in assenza di guasti

Il secondo gruppo di esperimenti effettuati riguarda il confronto tra la strategia di *mapping* sviluppata in questo lavoro di tesi e altre strategie presenti in letteratura ed analizzate nel Capitolo 2. Le strategie con cui è stato operato il confronto sono PL (Vedi Paragrafo 2.1.5 e [11], [12] e [13]), LEC-DN (Vedi Paragrafo 2.1.7 e [14] e [7]) e RW (Vedi Paragrafo 2.1.8 e [3]). Per effettuare il confronto sono state eseguite delle simulazioni utilizzando il programma *BanchMark* (vedi Appendice A) con test da 10, 100, 500 e 1000 applicazioni su un'architettura contenente 25 *cluster* identici. Ciascun test è stato ripetuto 10 volte per poter ottenere dei valori affidabili. Sono state quindi confrontate le medie aritmetiche dei tempi medi di esecuzione, i tempi medi di attività dei *cluster* e la deviazione standard, sia in valori assoluti che in valori percentuali rispetto ai tempi medi di attività dei *cluster*.

La sezione è quindi divisa in due paragrafi:

- nel Paragrafo 5.3.1 vengono riportati i dati riguardanti i tempi medi di esecuzione;
- nel Paragrafo 5.3.2 vengono invece riportati i dati riguardanti la deviazione standard dei tempi medi di attività.

5.3.1 Tempi di esecuzione

Il primo confronto effettuato tra le quattro strategie di *mapping* riguarda i tempi medi di esecuzione dei *cluster*, calcolati come illustrato nell'Equazione 5.1. È molto importante infatti che i tempi di esecuzione della strategia sviluppata restino in linea con i tempi di esecuzione delle altre strategie, in modo tale da non aumentare l'*overhead* dovuto al calcolo del *mapping*, allungando, di conseguenza, i tempi di esecuzione delle applicazioni. In Tabella 5.4 sono riportati i tempi di esecuzione medi rilevati nei test effettuati.

Gli stessi tempi sono stati poi riportati nel grafico visibile in Figura 5.8. Come visibile in Tabella 5.4 e in Figura 5.8, i tempi di esecuzione della

Tabella 5.4: Tempi medi di esecuzione degli esperimenti di confronto tra le quattro tecniche considerate

Tecnica	**10 App. [ms]**	**100 App. [ms]**	**500 App. [ms]**	**1000 App. [ms]**
Lavoro svolto	1087,30	11989,50	59932,40	119923,90
PL	1087,00	12106,50	59953,00	120021,80
LEC-DN	1214,90	12136	60108,80	120084,80
RW	1231,40	12311	60061	120117

tecnica sviluppata in questo lavoro di tesi e delle tecniche di confronto, adattate per l'esecuzione col modello di applicazione esposto nel Capitolo 3 non presentano differenze sostanziali.

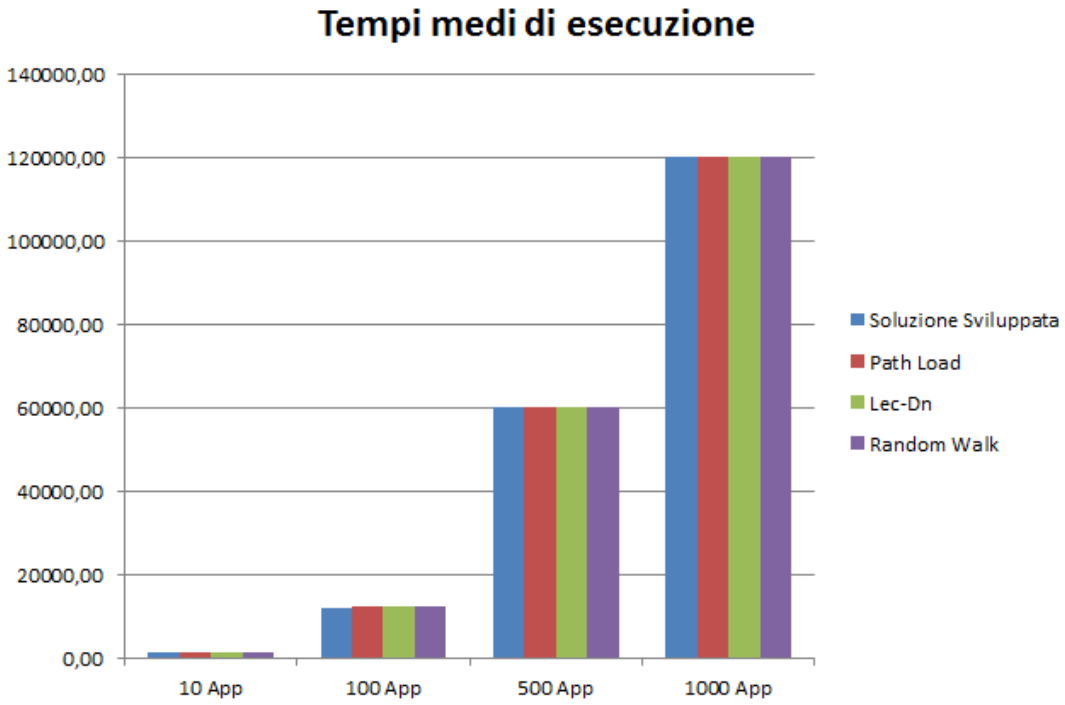


Figura 5.8: Grafico dei tempi di esecuzioni medi

Infatti, come riportato in Tabella 5.5, le variazioni in percentuale dei tempi di esecuzione delle tecniche di confronto, rispetto alla tecnica sviluppata, risultano significativi solo con poche applicazioni dove, inoltre, risultano essere anche più lente fino all'11%. Anche con l'aumento del numero di applicazioni la strategia sviluppata risulta essere più veloce delle altre tecniche, ma la differenza si riduce sensibilmente non superando mai il 2,6% (RW con 100 applicazioni) fino a quasi azzerarsi nel caso di 1000 applicazioni, dove la differenza massima è dello 0,15%.

5.3.2 Deviazione standard

Confrontati i tempi medi di esecuzione, si è proceduto al calcolo della deviazione standard dei tempi medi di occupazione dei *cluster* per verificare il comportamento delle quattro tecniche rispetto ai tempi di utilizzo

Tabella 5.5: Variazione in percentuale dei tempi di esecuzione media rispetto alla soluzione sviluppata

Tecnica	**10 App.** [$\sigma_{\%}$]	**100 App.** [$\sigma_{\%}$]	**500 App.**[$\sigma_{\%}$]	**1000 App.**[$\sigma_{\%}$]
Soluzione sviluppata	-	-	-	-
PL	-0,1	0,9	0,03	0,08
LEC-DN	11,7	1,2	0,4	0,13
RW	13,2	2,6	0,2	0,15

effettivo dei *clusters* stessi. Il valore della deviazione standard dei tempi medi di attività dei *cluster*, se rapportato con i valori medi stessi, indica infatti quanto varia l'utilizzo delle risorse. Un valore alto della deviazione standard indica un utilizzo non uniforme dei *cluster*, mentre un valore basso indica che, mediamente, tutti i cluster hanno tempi di attività simili.

Tabella 5.6: Deviazione standard dei tempi medi di attività dei cluster in valori assoluti

Tecnica	**10 App. [ms]**	**100 App. [ms]**	**500 App. [ms]**	**1000 App. [ms]**
Soluzione sviluppata	182,34	269,45	236,84	258,85
PL	203,65	817,90	1903,66	4224,75
LEC-DN	206,31	449,54	1700.62	3333,98
RW	221,05	1059,64	4188,34	7909,95

Tabella 5.7: Deviazione standard dei tempi medi di attività dei cluster in percentuale rispetto alla media dei tempi di attività

Tecnica	**10 App.** [$\sigma_{\%}$]	**100 App.** [$\sigma_{\%}$]	**500 App.** [$\sigma_{\%}$]	**1000 App.** [$\sigma_{\%}$]
Soluzione sviluppata	22,54	5,64	1,61	0,90
PL	25,21	17.07	12,87	14,74
LEC-DN	25,45	9,43	11,53	11,64
RW	27,20	22,18	28,37	27,61

Nelle Tabelle 5.6 e 5.7, sono riportati i valori della deviazione standard sia assoluti (Tabella 5.6) che in percentuale rispetto alla media dei tempi

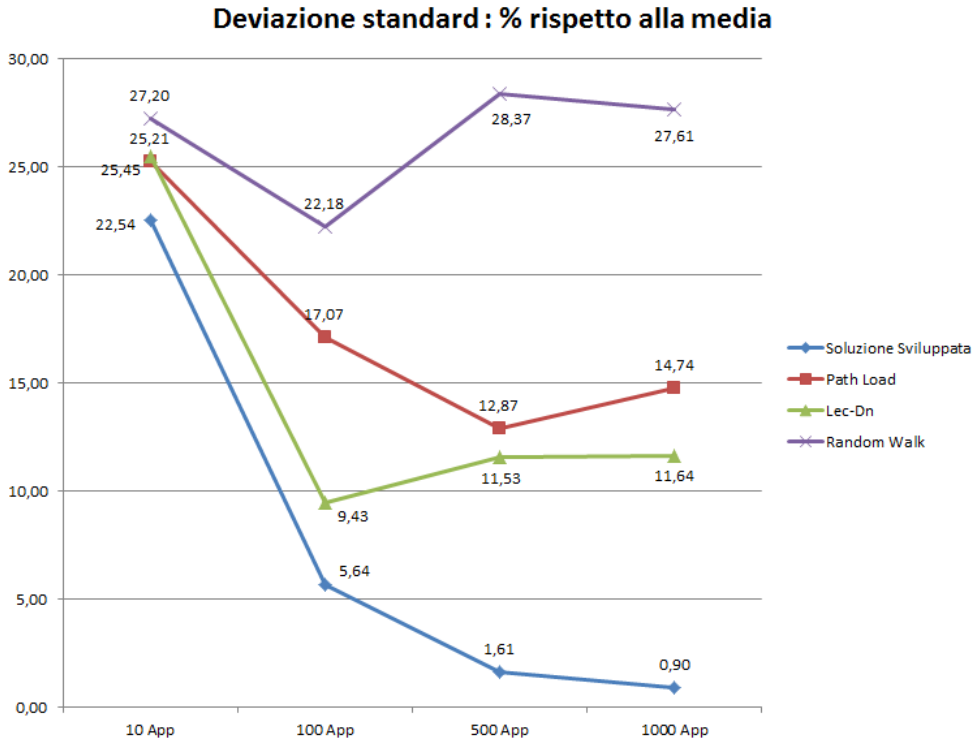


Figura 5.9: Grafico della deviazione standard in percentuale rispetto alla media dei tempi di attività

di attività dei clusters (Tabella 5.7), mentre, in Figura 5.9 è visualizzato il grafico realizzato con i dati riportati in Tabella 5.7.

Dal confronto delle deviazioni standard emerge una netta differenza tra le varie tecniche di *mapping* confrontate. Da un punto di vista dei valori assoluti della deviazione standard, riportati in Tabella 5.6, si nota come la soluzione proposta in questo lavoro di tesi mantenga un valore assoluto costante in tutti e quattro gli esperimenti, mentre le altre tre tecniche tendono ad aumentare tale valore con l'aumentare delle applicazioni eseguite. La Tabella 5.7 e il grafico riportato in Figura 5.9 illustrano invece i valori della deviazione standard percentuali rapportati alla media dei tempi medi di attività dei *cluster*. I dati mostrano come la soluzione proposta abbia una percentuale in costante diminuzione, conseguenza del mantenimento

di un valore assoluto della deviazione standard costante, che risulta di molto inferiore alle percentuali relative alle tre tecniche con cui viene operato il confronto. Per quanto riguarda PL e LEC-DN, si osserva che la percentuale diminuisce comunque con l'aumentare delle applicazioni, seppur essa tenda ad aumentare in valore assoluto. Rimangono invece abbastanza costanti i valori per la tecnica RW, segno di un importante aumento, in valore assoluto, dell'indicatore.

5.4 Confronto tra la strategia sviluppata e altre strategie di mapping in presenza di guasti permanenti

In questa sezione, vengono presentati i risultati ottenuti eseguendo i test di confronto della Sezione 5.3 in presenza di guasti permanenti, con lo scopo di a verificare l'impatto che essi hanno sui tempi di esecuzione delle applicazioni e sull'uniformità d'uso dei *cluster*. Utilizzando la stessa metodologia dei test descritti nella Sezione 5.2, sono stati quindi ri-eseguiti i test con 2, 4 e 8 guasti permanenti all'interno dell'architettura. La suddivisione della sezione è quindi la seguente:

- nel Paragrafo 5.4.1 vengono riportati i risultati degli esperimenti eseguiti con 100 applicazioni;
- nel Paragrafo 5.4.2 sono riportati i risultati raccolti eseguendo i test con 500 e 1000 applicazioni unitamente al confronto del comportamento della strategia sviluppata nelle varie situazioni analizzate.

5.4.1 Verifica della corretta inizializzazione del SoC ed esperimenti con 100 applicazioni

Come per il Simulatore, anche per il *BenchMark* (Descritto in Appendice A) è stato verificato l'effettivo funzionamento delle politiche implementate.

```
BenchMark::Initialize() - MATRIX INITIALIZATI(
1 0 0 0 1
0 0 0 0 1
0 0 0 0 0
0 F 1 0 F
1 1 0 0 1
BenchMark::Initialize() - Initializing 100 ra
```

Figura 5.10: Risultati dell'inizializzazione del SoC in presenza di due guasti

In particolare il primo passo svolto è stato quello di verificare che durante l'esecuzione del programma venissero effettivamente marcati come guasti un numero di guasti pari a quelli specificati da linea di comando.

Tale verifica è stata effettuata andando ad esaminare i risultati di inizializzazione del SoC all'interno il file di log generato durante l'esecuzione di un'istanza di *BenchMark* contenente 2 guasti permanenti. In Figura 5.10 sono dunque riportati i risultati estrapolati dal file di Log. Il SoC è rappresentato come una matrice quadrata con un numero di *cluster* pari a quanto specificato da linea di comando. Ciascun *cluster* viene rappresentato con un carattere che può assumere i seguenti valori:

- 0 se il *cluster* è libero;
- 1 se il *cluster* è occupato;
- F se il *cluster* è guasto.

I dati riportati in Figura 5.10 confermano la corretta inizializzazione del SoC e, conseguentemente, la correttezza dell'esecuzione del *BenchMark*.

Effettuata la verifica del *Benchmark* è stato effettuato il primo gruppo di esperimenti volti a confrontare le euristiche considerate con quanto sviluppato in questo lavoro di tesi. Gli esperimenti sono stati effettuati andando ad eseguire 100 applicazioni generate dal *BenchMark* su un SoC contenente 25 *cluster*. Sono state quindi eseguite quattro istanze del *BenchMark* specificando per ciascuna un numero diverso di guasti: 0 per la prima istanza, 2

per la seconda istanza, 4 per la terza istanza e 8 per la quarta istanza. Per ciascuna istanza è stato raccolto il valore della deviazione standard percentuale rispetto alla media dei tempi medi di attività dei *cluster*, ottenendo i dati riportati in Tabella 5.8 ed il cui grafico è illustrato in Figura 5.11.

Come per i test senza guasti, anche in questo caso viene misurata la deviazione standard percentuale rispetto alla media dei tempi medi di attività dei cluster perchè essa rappresenta un indice per misurare il *load balancing* dei *cluster*.

Tabella 5.8: Deviazione standard dei tempi medi di attività dei clusters in percentuale rispetto alla media dei tempi di attività dopo l'esecuzione di 100 applicazioni e in presenza di 2,4 e 8 guasti permanenti

Tecnica	**0 Guasti** [$\sigma_{\%}$]	**2 Guasti** [$\sigma_{\%}$]	**4 Guasti** [$\sigma_{\%}$]	**8 Guasti** [$\sigma_{\%}$]
Soluzione sviluppata	5,64	5,36	7,52	10,29
PL	17,07	20,73	20,33	22,58
LEC-DN	12,43	14,73	16,35	20,56
RW	27,18	23,57	22,85	19,79

I dati ottenuti dagli esperimenti mostrano una crescita costante del valore percentuale della deviazione standard della strategia sviluppata sino al 10,29% dell'esperimento con 8 guasti. Lo stesso andamento è presente anche nei valori relativi alle euristiche LEC-DN e PL, che passano, rispettivamente, dal 12,43% e dal 17,07% dell'esperimento con 0 guasti, al 20,56% ed al 22,58% dell'esperimento con 8 guasti. L'andamento dei valori relativi all'euristica RW risulta invece opposto: si passa infatti dal 27,18% dell'esperimento con 0 guasti al 19,79% dell'esperimento con 8 guasti. Questo andamento dei dati indica che un numero significativo di guasti influenza pesantemente la scelta dei *cluster* nell'utilizzo di euristiche completamente deterministiche mentre, invece, la componente di scelta casuale del *cluster* presente all'interno di RW permette di variare maggiormente i *cluster* utilizzati portando dunque alla diminuzione della deviazione standard a valori

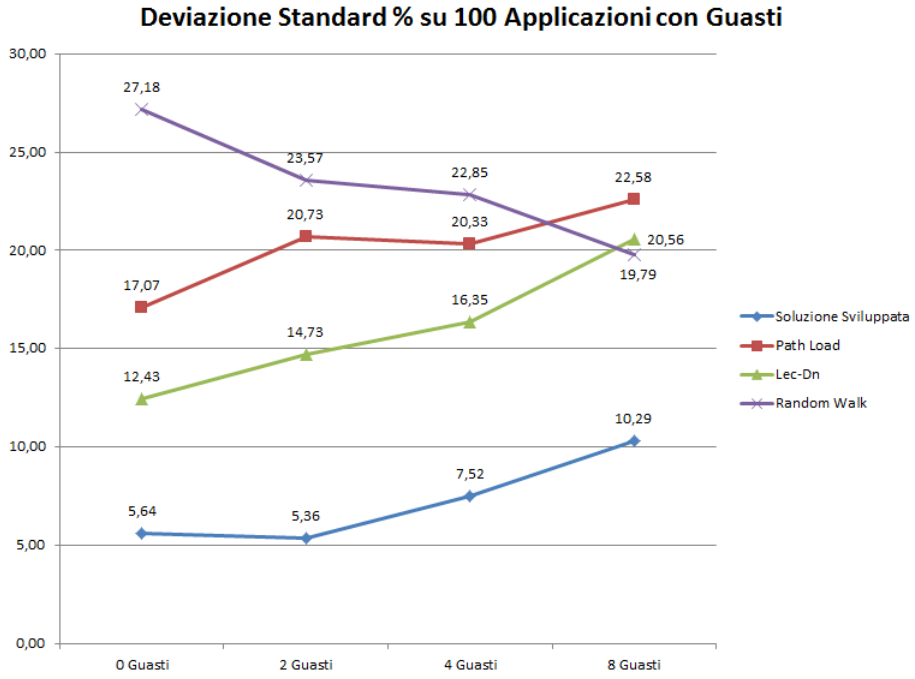


Figura 5.11: Confronto della deviazione standard percentuale rispetto alla media con 100 applicazioni e 0,2,4 e 8 guasti permanenti

in linea a quelli delle altre euristiche.

5.4.2 Esperimenti con 500 e 1000 applicazioni

Al fine di approfondire l'andamento della deviazione standard, gli stessi test illustrati nel Paragrafo 5.4.1 sono stati ripetuti con 500 e 1000 applicazioni. I dati raccolti da questi esperimenti sono riportati nelle Tabelle 5.9 e 5.10 e nei relativi grafici di Figura 5.12. I test con 500 e 1000 applicazioni confermano l'andamento generale rilevato con i test effettuati con 100 applicazioni. Anche in questi test i valori della deviazione standard presentano un andamento crescente per la strategia sviluppata, PL e LEC-DN mentre presentano un andamento opposto per l'euristica RW.

Per quanto riguarda gli esperimenti con 500 applicazioni,il valore della deviazione standard percentuale per la tecnica sviluppata sale dal 1,61%

Tabella 5.9: Deviazione standard dei tempi medi di attività dei clusters in percentuale rispetto alla media dei tempi di attività dopo l'esecuzione di 500 applicazioni e in presenza di 0, 2, 4 e 8 guasti permanenti

Tecnica	**0 Guasti** [$\sigma_{\%}$]	**2 Guasti** [$\sigma_{\%}$]	**4 Guasti** [$\sigma_{\%}$]	**8 Guasti** [$\sigma_{\%}$]
Soluzione sviluppata	1,61	4,67	7,56	10,46
PL	12,87	21,19	19,62	20,28
LEC-DN	11,53	12,66	14,82	16,56
RW	28,37	26,95	23,40	20,40

Tabella 5.10: Deviazione standard dei tempi medi di attività dei clusters in percentuale rispetto alla media dei tempi di attività dopo l'esecuzione di 1000 applicazioni e in presenza di 0, 2 ,4 e 8 guasti permanenti

Tecnica	**0 Guasti** [$\sigma_{\%}$]	**2 Guasti** [$\sigma_{\%}$]	**4 Guasti** [$\sigma_{\%}$]	**8 Guasti** [$\sigma_{\%}$]
Soluzione sviluppata	0,90	4,68	6,54	10,42
PL	14,74	19,49	20,63	20,06
LEC-DN	11,64	13,70	16,95	22,12
RW	27,61	25,78	20,75	20,06

dell'esperimento con 0 guasti al 10,46% dell'esperimento con 8 guasti, il valore per l'euristica PL sale dal 12,87% al 20,28% ed il valore per l'euristica LEC-DN sale dal 11,53% al 16,56%. I valori per l'euristica RW scendono invece dal 28,37% dell'esperimento con 0 guasti al 20,40% dell'esperimento con 8 guasti. L'andamento per gli esperimenti con 1000 applicazioni è del tutto analogo a quello degli esperimenti con 500 applicazioni, confermando così il comportamento delle euristiche in presenza di guasti permanenti.

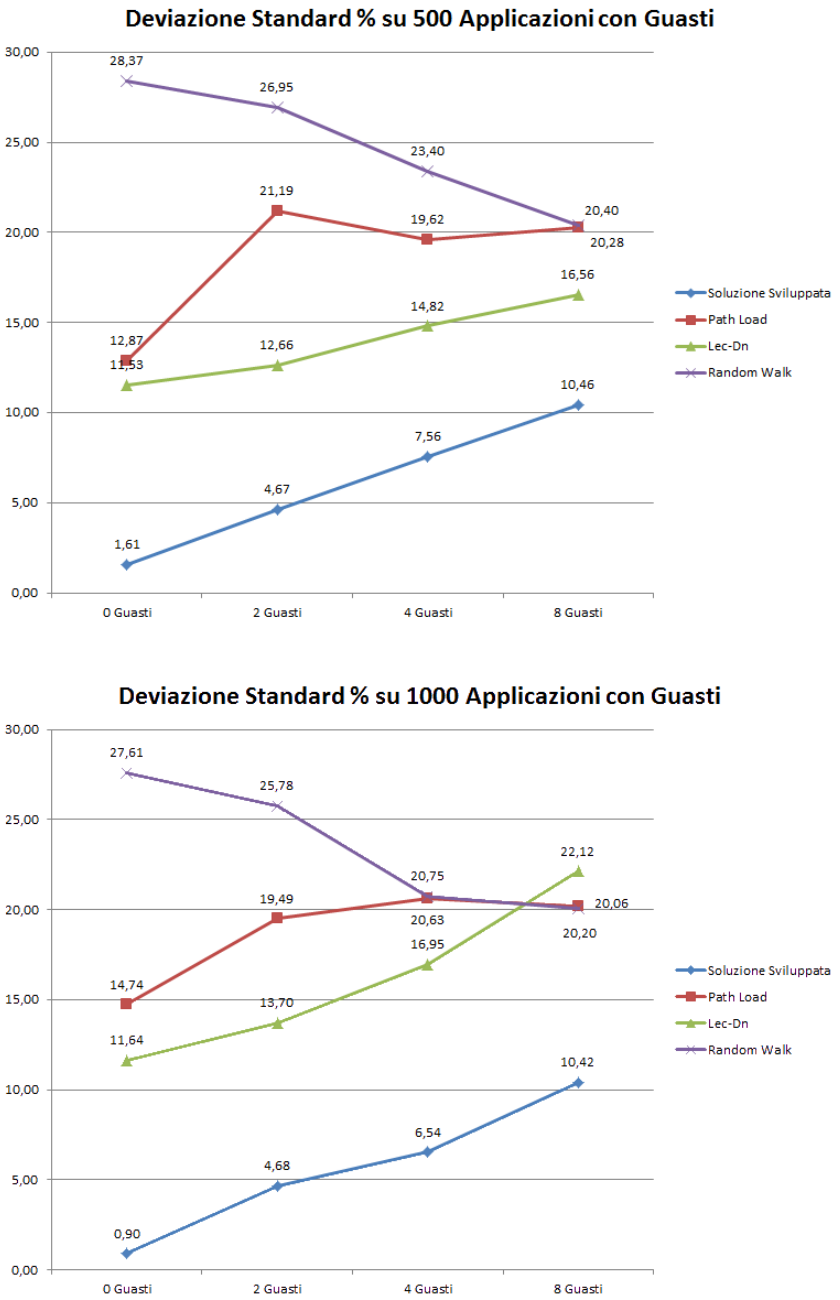


Figura 5.12: Confronto della deviazione standard percentuale rispetto alla media con 500 e 1000 applicazioni e con 0,2,4 e 8 guasti permanenti

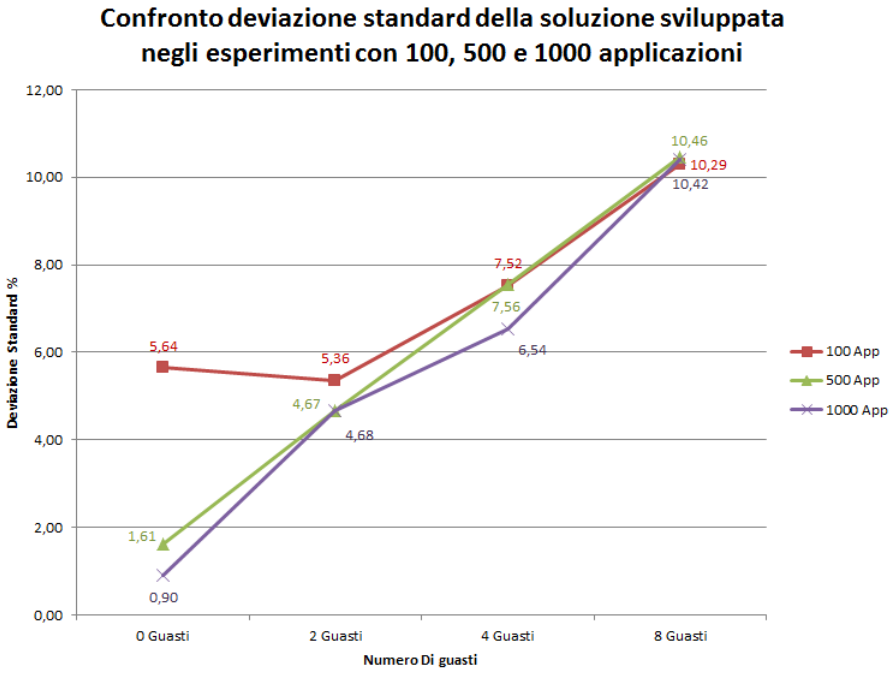


Figura 5.13: Confronto dei valori della deviazione standard dell'euristica sviluppata negli esperimenti con 100, 500 e 1000 applicazioni

A conclusione della sezione, viene presentato un grafico comparativo dei valori della deviazione standard percentuale relativo alla soluzione sviluppata in questa tesi. Il grafico, riportato in Figura 5.13 permette di confermare che l'andamento del valore della deviazione standard percentuale, in caso di guasti permanenti, dipenda solo marginalmente dal tempo di attività dei *cluster* ma risulti influenzato quasi unicamente dal numero di guasti.

5.5 Comportamento della strategia in presenza di guasti transitori

In questa sezione vengono presentati i risultati riguardanti il comportamento della strategia sviluppata in questo lavoro di tesi in caso di gua-

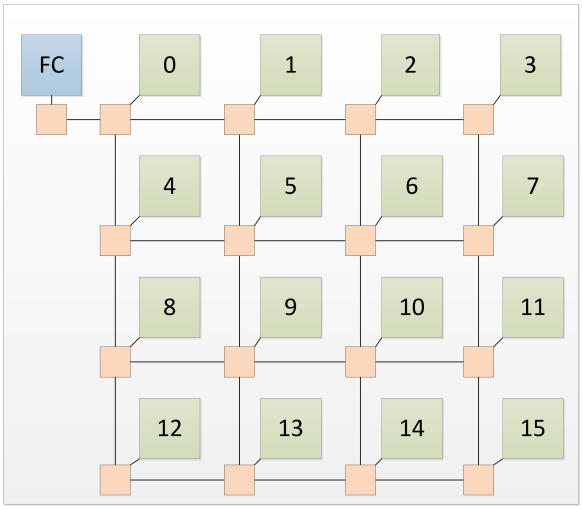


Figura 5.14: Architettura utilizzata dal simulatore durante la verifica dell'euristica con guasti transitori

sto transitorio ai *cluster*. In questa sezione viene quindi presentato l'esperimento effettuato al fine di verificare il corretto adattamento dell'esecuzione delle applicazioni qualora si verifichino guasti temporanei ai *cluster* appartenenti al SoC. La sezione si articola su un unico paragrafo, all'interno del quale viene esposto l'esperimento effettuato.

5.5.1 Descrizione dell'esperimento effettuato in presenza di guasti transitori

L'esperimento è stato effettuato al fine di verificare la corretta esclusione dei *cluster* momentaneamente non disponibili durante la fase di *mapping* dell'applicazione. Per effettuare la verifica è stato utilizzato il programma Simulatore, utilizzando il modello base dell'architettura p2012 descritto in [6] e raffigurato in Figura 5.14 e utilizzando un set di 13 esecuzioni di un'applicazione composta da 2 sub-applicazioni. I tempi di arrivo delle 13 esecuzioni dell'applicazione sono riportati in Tabella 5.11. Per quanto riguarda il guasto, si è scelto di specificare un guasto transitorio del *cluster* numero 5 della durata di 190 ms con istante iniziale a 63ms e istante finale a 253ms.

Tabella 5.11: Tempi di arrivo delle esecuzioni dell'applicazione utilizzata nell'esperimento

Numero esecuzione	**T di arrivo [ms]**
1	10
2	40
3	70
4	100
5	130
6	160
7	190
8	210
9	360
10	390
11	410
12	420
13	430

La verifica è stata effettuata eseguendo due simulazioni, una simulazione effettuata senza il guasto transitorio e una simulazione con il guasto. I risultati delle due simulazioni, in termini di *mapping* delle applicazioni sui *cluster*, sono stati poi confrontati prendendo lo stato del SoC prima, durante e dopo l'accadere del guasto per quanto riguarda la simulazione all'interno del quale esso è avvenuto, e prendendo i corrispondenti stati per quanto riguarda la simulazione senza guasto. In Figura 5.15 sono riportati gli stati del SoC utilizzati per effettuare il confronto. Nella parte sinistra della stessa, sono riportati gli stati del SoC rilevati nella simulazione senza guasto, nella parte destra invece, sono riportati gli stati del SoC rilevati nella simulazione con guasto temporaneo. In Figura 5.15 a è riportato lo stato del SoC al tempo t=40 ms. In entrambe le simulazioni il *mapping* delle prime due esecuzioni dell'applicazione è identico, in quanto, non essendosi ancora verificato il guasto temporaneo, la strategia sviluppata lavora in modo identico in entrambi i casi. In Figura 5.15 b invece, viene riportato lo stato del dispositivo a un tempo t=70 ms. Essendo il guasto apparso al tempo t=63 ms, il risultato delle due simulazioni risulta diverso. La simulazione

Figura 5.15: Risultati delle simulazioni senza guasto transitorio (a sinistra) e con guasto transitorio (a destra).

senza guasto infatti, porta all'utilizzo del cluster 5 assegnandolo a una sub-applicazione appartenente alla terza esecuzione dell'applicazione mentre,

la simulazione con guasto, porta ad assegnare la suddetta sub-applicazione al cluster 8, andando di fatto a confermare la corretta esclusione del cluster guasto dal *mapping*. La parte c della Figura 5.15 mostra invece lo stato del SoC al tempo t=430 ms. Ovviamente, a causa del guasto avvenuto in precedenza, il *mapping* delle applicazioni risulta differente ma è importante notare come, anche nella simulazione con guasto, essendo il cluster 5 tornato disponibile, esso venga effettivamente utilizzato al fine di eseguire le applicazioni denotando il corretto comportamento della strategia sviluppata.

5.6 Analisi dei risultati

In questa sezione vengono analizzati i risultati dei test presentati nelle Sezioni 5.2, 5.3, 5.4 e 5.5. L'analisi dei risultati è stata strutturata su quattro paragrafi; nel **Paragrafo 5.6.1** viene effettuata l'analisi dei risultati degli esperimenti volti a determinare la correttezza del funzionamento del simulatore; nel **Paragrafo 5.6.2** viene introdotta l'analisi degli esperimenti eseguiti per verificare la correttezza della strategia sviluppata in questo lavoro di tesi mentre nel **Paragrafo 5.6.3** viene introdotta l'analisi dei risultati ottenuti con gli esperimenti in cui la strategia sviluppata è stata confrontata con le euristiche PL, LEC-DN e RW senza la presenza di guasti. Nel **Paragrafo 5.6.4** Viene infine presentata l'analisi dei risultati ottenuti con gli esperimenti di confronto in presenza di guasti.

5.6.1 Analisi dei risultati di correttezza dell'implementazione del simulatore

Un risultato fondamentale da ottenere, al fine di validare i risultati numerici ottenuti, è stato verificare il corretto funzionamento del simulatore. Per ottenere questo risultato è stato svolto un unico esperimento, descritto

nel Paragrafo 5.2.3, con lo scopo di verificare la corretta sincronizzazione tra le varie sub-applicazioni qualora esistano delle dipendenze tra i loro processi. Uno scenario simile permette di validare non solo il corretto funzionamento della parte di sincronizzazione, ma anche il corretto funzionamento della lettura e dell'interpretazione dei dati forniti al simulatore, fornendo così una validazione generale del modello utilizzato e del simulatore stesso. I risultati riportati da tale esperimento, illustrati in Tabella 5.3, confermano il corretto funzionamento della simulazione permettendo di considerare validi i risultati ottenuti da tutte le simulazioni effettuate al fine di validare la strategia di *mapping* sviluppata in questo lavoro di tesi.

5.6.2 Analisi dei risultati di funzionamento

Per quanto riguarda il corretto funzionamento della strategia di *mapping* sviluppata, esso viene confermato, in assenza di guasti, dagli esperimenti presentati nella Sezione 5.2. Il corretto funzionamento dell'euristica in presenza di guasti transitori è invece provato con l'esperimento descritto nella Sezione 5.5. Il **primo esperimento**, descritto nel Paragrafo 5.2.1 Mostra come effettivamente le aree vengano rilevate e scelte come voluto. Alcune situazioni emerse durante l'esecuzione dell'esperimento, ed illustrate in Figura 5.6, mostrano come tutti gli obiettivi di corretta allocazione delle sub-applicazioni vengano raggiunti. Le seguenti situazioni, in particolare, risultano significative:

- L'assegnamento dell'**applicazione 2** dimostra come, in presenza di diverse aree libere, venga scelta l'area più piccola che possa contenere l'applicazione;
- L'assegnamento delle **applicazioni 4 e 9** dimostra che la forma ottenuta durante l'assegnamento di applicazioni composte da molte sub-applicazioni, nonostante i vincoli sul tempo di attività dei *clu-*

ster, risulta abbastanza regolare qualora la forma dell'area ospitante lo permetta.

- L'assegnamento dell'**applicazione 10** invece, dimostra come, a Multi-Processor SoC (MPSoC) scarico, la forma dell'area dei *cluster* utilizzati sia invece regolare.

Nonostante l'effettivo funzionamento dell'euristica secondo il comportamento desiderato, esso è fortemente limitato qualora dovessero essere eseguite delle applicazioni con un numero di sub-applicazioni maggiore della grandezza dell'area libera più grande oppure maggiori del numero di *cluster* presenti all'interno del MPSoC utilizzato.

Nel primo caso si è agito andando ad inserire l'applicazione all'interno di una coda di attesa ed eseguendola nel momento in cui vi sia un'area capace di contenerla. Questo comportamento è conseguenza della non-considerazione di eventuali priorità delle applicazioni in entrata nel sistema. Nel caso in cui però si voglia implementare una versione della strategia che supporti le priorità, questo comportamento non è più accettabile qualora arrivasse un'applicazione ad alta priorità per la quale vi siano sufficienti risorse disponibili ma non vi sia un'area sufficientemente grande da contenerla. In questo caso l'inserimento all'interno della coda di applicazioni non è tollerabile e la strategia di *mapping* andrebbe migliorata adattandola alla gestione di questa situazione.

Nel secondo caso invece, non sono state considerate applicazioni più grandi del MPSoC durante la simulazione. Si è data infatti per scontata una fase di compilazione e *profiling* del codice che permetta di ottenere applicazioni in grado di essere eseguite sul MPSoC utilizzato. Nel caso però in cui la strategia elaborata debba essere utilizzata senza queste fasi, le applicazioni troppo grandi verrebbero inevitabilmente scartate e pertanto la gestione di questo caso dovrebbe essere aggiunta alla strategia stessa.

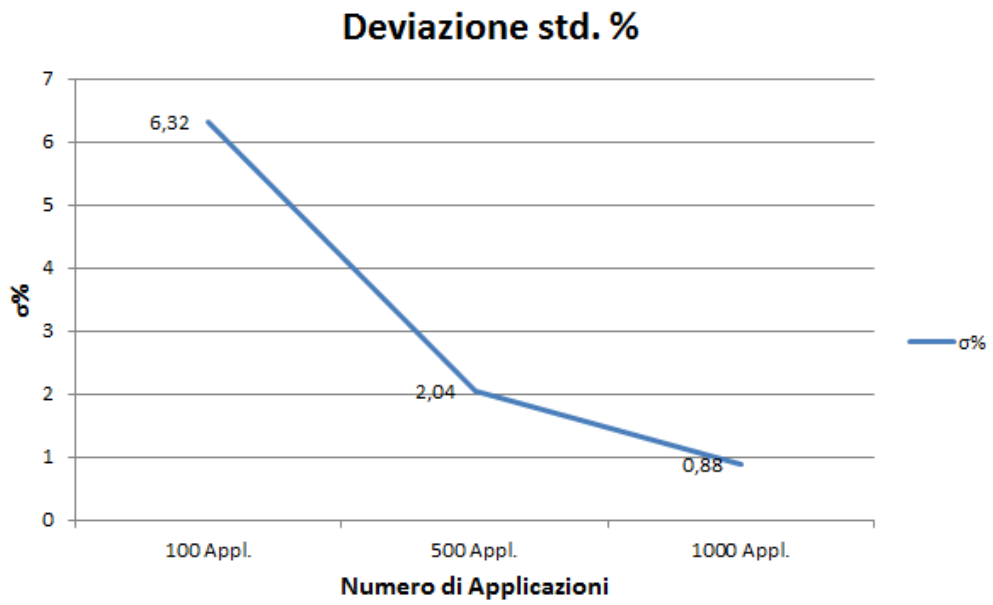


Figura 5.16: Grafico raffigurante i risultati riportati in Tabella 5.1

Nel **Secondo esperimento**, descritto nel Paragrafo 5.3.2 è stato provato il corretto funzionamento delle metriche volte a mantenere uniforme l'uso dei *cluster* presenti all'interno del sistema utilizzato. La simulazione dell'esecuzione di 100, 500 e 1000 applicazioni ha prodotto i risultati riportati in Tabella 5.1.

Il dato principale esaminato è la deviazione standard rapportata ai tempi medi di attività dei *cluster* ed espressa in percentuale: più basso infatti è questo valore e più i tempi di attività dei clusters sono simili, andando quindi ad uniformare l'utilizzo degli stessi e a massimizzarne la vita media. I risultati in Tabella 5.1 e qui riportati in Figura 5.16 dimostrano che maggiore è il numero delle applicazioni eseguite e minore è il valore della deviazione standard, andando a verificare il raggiungimento dell'obiettivo preposto. Significativo infatti, risulta il valore di 0,88% ottenuto nella simulazione dell'esecuzione di 1000 applicazioni anche se, in generale, tale valore non è mai superiore al 7% (6,32% nella simulazione dell'esecuzione

di 100 applicazioni) andando a confermare ulteriormente la bontà del lavoro sviluppato. Come però mostrato durante la descrizione dello sviluppo della strategia di *mapping* nel Capitolo 4, e confermato dagli esperimenti illustrati nel Paragrafo 5.2.1, le metriche di uniformità causano una certa irregolarità nelle forme delle aree di *cluster* che eseguono un'applicazione e al conseguente aumento del rischio di frammentazione del dispositivo e dei tempi di esecuzione delle applicazioni.

Per quanto riguarda il comportamento in presenza di guasti transitori invece, esso è stato testato nell'esperimento descritto nella Sezione 5.5. I risultati dell'esperimento, illustrati in Figura 5.15, confermano come un cluster che presenti un guasto temporaneo venga correttamente escluso dall'esecuzione dell'applicazione, andando a confermare il comportamento desiderato dell'euristica.

5.6.3 Analisi dei risultati di confronto con le altre euristiche in assenza di guasti

I risultati degli esperimenti della Sezione 5.2 hanno mostrato il corretto funzionamento della strategia di *mapping* sviluppata in questo lavoro di tesi. Essa è stata poi confrontata con le euristiche PL, LEC-DN e RW per verificarne le prestazioni sia dal punto di vista temporale, che dal punto di vista dell'uniformità di utilizzo dei clusters. Al fine di provare le prestazioni è stato eseguito l'esperimento descritto nella Sezione 5.3. Dal punto di vista dei **tempi di esecuzione**, la tecnica sviluppata risulta paragonabile all'implementazione delle tre euristiche utilizzate per il confronto, descritta nella Sezione 4.5. I tempi di esecuzione delle simulazioni con 10, 100, 500 e 1000 applicazioni, riportati in Tabella5.4 e in Figura 5.8 , risultano infatti leggermente inferiori a quelli delle tre euristiche confermando l'effettività della soluzione sviluppata. I risultati dal punto di vista dell'uniformità invece, riportati in Tabella 5.7 dimostrano come la strategia sviluppata sia

sensibilmente migliore delle tre euristiche di confronto, in particolare con l'esecuzione di numerose applicazioni, dove la deviazione standard, in percentuale rispetto alla media dei tempi di attività dei *cluster* è praticamente nulla per l'euristica sviluppata mentre si attesta attorno all'11% per PL, al 14% per LEC-DN e al 27% per RW. I risultati ottenuti sono dunque molto buoni ma sono stati ottenuti da simulazioni che non consideravano la possibilità di guasti all'interno dell'architettura.

5.6.4 Analisi dei risultati di confronto con le altre euristiche in presenza di guasti

Gli esperimenti eseguiti in assenza di guasti hanno confermato il buon comportamento della strategia sviluppata rispetto alle soluzioni pre-esistenti. Per comprendere meglio il comportamento della tecnica sviluppata sono inoltre stati eseguiti test di confronto in presenza di guasti permanenti al fine di andare a investigare il deterioramento delle prestazioni della soluzione proposta in termini di uniformità di utilizzo dei *cluster*. Tali esperimenti sono descritti nella Sezione 5.4 e sostanzialmente consistono nella ri-esecuzione dei test descritti nella Sezione 5.3 in presenza di 0, 2, 4 e 8 guasti. I risultati ottenuti, illustrati nelle Tabelle 5.8, 5.9 e 5.10 e nei corrispondenti grafici delle Figure 5.11 e 5.12 mostrano un aumento della deviazione standard percentuale man mano che aumentano i guasti presenti nell'architettura. Dai dati si nota in particolare come sia ininfluente il numero delle applicazioni eseguite sia per la strategia di *mapping* sviluppata, che per le tre euristiche di confronto. I valori ottenuti dal lavoro sviluppato in questa tesi vedono la deviazione standard percentuale rapportata alla media salire a circa 5% nel caso di due guasti, al 7,5% circa nel caso di quattro guasti fino ad arrivare al 10,4% circa nel caso di otto guasti. Tutti questi valori risultano inferiori rispetto a quanto rilevato dalle simulazioni delle altre euristiche, provando la bontà del lavoro svolto anche in presenza di

più guasti permanenti.

In questo capitolo è stato introdotto tutto il lavoro svolto al fine di validare la strategia di *mapping* elaborata in questo lavoro di tesi. Utilizzando i modelli di applicazione e architettura sviluppati nel Capitolo 3 è stato implementato un Simulatore della piattaforma p2012 che ha permesso l'implementazione della strategia di *mapping* sviluppata introdotta nel Capitolo 4. Nel simulatore è stato inoltre implementato un programma *BenchMark*, descritto dettagliatamente in Appendice A, che ha permesso il confronto tra la strategia sviluppata e le euristiche esistenti ed analizzate nel Capitolo 2. Con questi strumenti sono stati effettuati tutti i test descritti in questo capitolo che hanno permesso di verificare la correttezza e le prestazioni del lavoro svolto in questa tesi e di trarre le dovute conclusioni.

Nel capitolo successivo verranno quindi introdotte le considerazioni finali, sulla base dei risultati esposti in questo capitolo e verranno inoltre introdotti i possibili sviluppi futuri a cui il lavoro svolto può andare incontro.

Capitolo 6

Conclusioni e sviluppi futuri

Nel Capitolo 5 sono stati illustrati i test effettuati al fine di verificare il funzionamento della strategia di *mapping* elaborata in questo lavoro di tesi. Tale strategia è stata inoltre confrontata con altre euristiche presente in letteratura al fine di saggiarne le prestazioni. In questo capitolo vengono effettuate le considerazioni finali sull'interno lavoro presentato in questa tesi, con lo scopo di evidenziare se e come gli obiettivi preposti sono stati raggiunti, eventuali debolezze della strategia e gli sviluppi che in futuro possono portare ad un miglioramento della stessa. Il capitolo è quindi organizzato in due sezioni:

- nella Sezione 6.1 vengono illustrate le conclusioni finali basate sui risultati ottenuti nei test;
- nella Sezione 6.2 vengono effettuate delle proposte di sviluppi futuri della strategia al fine di migliorarne le prestazioni ed estenderne i casi di utilizzo.

6.1 Conclusioni

Il lavoro descritto in questa tesi, come introdotto nella Sezione 1.1 si è posto due obiettivi principali:

- l'introduzione di un modello di architettura Multi-Processor SoC (MPSoC) basato sulla piattaforma p2012 di S.T. Microelectronics e di un modello di applicazione più completo rispetto al modello *master/slave* utilizzato in letteratura (vedi Capitolo 2)
- L'elaborazione di una strategia base di assegnamento dinamico che permetta di minimizzare i tempi di esecuzione delle applicazioni evitando l'invecchiamento precoce dei componenti del MPSoC causato da un uso intensivo e squilibrato degli stessi

Al fine di raggiungere questi obiettivi, è stata effettuata l'analisi di diverse tecniche di *mapping* dinamico, valutando pregi, difetti e applicabilità delle stesse alla piattaforma p2012, anche in relazione al tipo di applicazione su cui questo lavoro di tesi si è focalizzato. L'analisi delle tecniche esistenti, introdotta nel Capitolo 2, ha evidenziato inoltre la necessità di introdurre ex-novo, oltre al modello di architettura, anche un modello di applicazione che permettesse la modellazione di applicazioni complesse, scomponibili a loro volta in sub-applicazioni composte da *thread* eseguibili in parallelo. Le tecniche analizzate infatti fanno uso quasi esclusivamente di un modello di applicazione di tipo *master/slave*, non adatto agli scopi che il lavoro presentato in questa tesi si è preposto.

Nel Capitolo 3 sono quindi stati introdotti i modelli di architettura e di applicazione utilizzati successivamente per la validazione della strategia di *mapping* sviluppata. Il modello di applicazione è stato sviluppato basandosi sull'architettura p2012 si S.T. Microelectronics (descritta in [6]) mentre il modello di applicazione è stato elaborato partendo dal modello *master/slave* ed estendendolo secondo i requisiti individuati.

Sulla base dei modelli introdotti è stata quindi progettata la strategia di *mapping* dinamico, rappresentante l'obiettivo principale del lavoro svolto. Essa è stata introdotta nel Capitolo 4 ed è stata sviluppata utilizzando come capi saldi gli obiettivi di minimizzazione dei tempi di esecuzione delle applicazioni e di utilizzo uniforme, in termini di tempo di attività, dei *cluster* facenti parte dell'architettura utilizzata come piattaforma di test.

Al fine di validare i modelli elaborati e la strategia di *mapping* dinamico sono stati elaborati un Simulatore dell'architettura e un programma *Benchmark*, che hanno permesso l'esecuzione dei test descritti nel Capitolo 5. Tali test spaziano dalla verifica dei modelli al confronto della strategia sviluppata con alcune delle strategie di *mapping* dinamico presenti in letteratura (In particolare con le strategie Path Load (PL), Low Energy Consumption - Dependences Neighborhood (LEC-DN) e Random Walk (RW)). I risultati, analizzati nella Sezione 5.6, permettono di concludere che il lavoro svolto in questa tesi raggiunge tutti gli obiettivi preposti. I risultati di test di funzionamento del simulatore di architettura, illustrati nel Paragrafo 5.6.1 hanno permesso di verificare la correttezza dei modelli elaborati. È stata in particolare verificata l'effettività del modello di applicazione elaborato, che raggiunge lo scopo di riuscire a rappresentare applicazioni complesse andando così oltre al tipico paradigma *master/slave* utilizzato in letteratura. Anche dal punto di vista della strategia di *mapping* sviluppata è possibile concludere che il lavoro soddisfa i requisiti posti. L'analisi dei risultati dei test, effettuata nei Paragrafi 5.6.1, 5.6.2 e 5.6.3 mostra che non solo la strategia sviluppata riesce a raggiungere gli obiettivi, ma anche che essa presenta tempi di elaborazioni simili alle strategie presenti in letteratura.

È quindi possibile concludere che il lavoro svolto rappresenta una base di partenza per quanto riguarda l'elaborazione di strategie complesse di *mapping* dinamico di applicazioni fortemente parallele su sistemi MPSoC che prendano in considerazione anche il problema dell'affidabilità del di-

spositivo su cui vengono implementate.

6.2 Sviluppi Futuri

L'ambito di ricerca in cui questo lavoro di tesi si è posto è vasto e recente. Il lavoro svolto in questa tesi può rappresentare una base di partenza per l'elaborazione di strategie di *mapping* più complete e complesse. Pur raggiungendo gli obiettivi preposti, la strategia sviluppata in questo lavoro di tesi non è esente da difetti, come esplicato nella Sezione 5.6, che ne impediscono l'utilizzo al di fuori dell'ambito in cui essa è stata sviluppata. Il limite maggiore della strategia, è rappresentato dal criterio di scelta utilizzato per l'assegnazione di una sub-applicazione a un *cluster* che è strutturato su due fasi ben distinte (vedi Sezione 4.4):una per ogni metrica utilizzata. In questo modo non è possibile definire un'unica funzione di costo sulla base della quale andare a scegliere il cluster senza dover attraversare entrambe le fasi di selezione. Una prima miglioria alla strategia potrebbe dunque essere quella della creazione di una funzione di costo che sia combinazione lineare delle due metriche adottate. In questo modo non solo si snellirebbe la fase di scelta del *cluster* ma sarebbe possibile effettuare un bilanciamento dei pesi da dare alle due metriche sulla base delle esigenze dello specifico ambito. Un ulteriore vantaggio deriverebbe inoltre dalla struttura a combinazione lineare: l'inserimento di nuove metriche di valutazione all'interno della funzione di costo risulterebbe in questo modo più semplice. Un'altra importante direzione di evoluzione della strategia potrebbe essere rappresentata dall'adattamento della stessa ad altri ambiti di utilizzo operativo: la strategia è stata infatti pensata per essere adottata in contesti di calcolo parallelo ed intensivo. Essa risulterebbe pertanto inutilizzabile negli altri tipi di ambienti di esecuzione (ad esempio in ambito *real-time*) e quindi potrebbe essere interessante inserire delle metriche atte

ad adattare la strategia ad ambiti per cui essa non è stata pensata al fine di verificarne il comportamento.

La strategia di *mapping* però non rappresenta l'unica parte del lavoro sviluppabile ulteriormente. Data la vastità dell'ambito di ricerca, sono state volutamente tralasciate, come introdotto nel Capitolo 1, le problematiche dell'implementazione di meccanismi di tolleranza ai guasti all'interno dei *cluster*, considerandole implementate. L'implementazione effettiva delle stesse potrebbe essere un'ulteriore sviluppo al lavoro effettuato. La strategia sviluppata inoltre, agisce a livello Network-on-Chip (NoC). Durante lo sviluppo del modello di architettura, è stato estrapolato un modello di esecuzione delle sub-applicazioni all'interno dei *cluster* che prevede lo *scheduling* dei *thread* di cui esse si compongono utilizzando la semplice tecnica First In First Out (FIFO). Un ulteriore sviluppo potrebbe quindi essere l'implementazione di una strategia di *schdeuling* più complessa che permetta di migliorare quanto fatto a livello NoC con la strategia di *mapping* dinamico sviluppata in questo lavoro di tesi.

Appendice A

Dettagli sul simulatore dell'architettura

Nel Capitolo 5 è stato introdotto il simulatore di architettura p2012. In questa Appendice vengono forniti i dettagli di implementazione del simulatore stesso unitamente a una descrizione dettagliata del programma *Benchmark* creato sulla base del simulatore stesso con lo scopo di confrontare la strategia di *mapping* sviluppata in questo lavoro di tesi con le strategie PL, LEC-DN e RW. Entrambi i programmi sono stati implementati utilizzando il linguaggio di programmazione C++ e la libreria SystemC, unitamente alla sua estensione Transaction Level Modeling (TLM) che ha permesso la creazione di una simulazione comportamentale. La trattazione dei dettagli dei due programmi è suddivisa in due sezioni; nella **Sezione A.1** vengono introdotti i dettagli di implementazione del simulatore, tralasciando quanto già introdotto nella Sezione 5.1 mentre nella **Sezione A.2** viene introdotta la descrizione completa del programma *Benchmark*.

A.1 Simulatore

Nella Sezione 5.1 è stato introdotto il simulatore dell'architettura p2012 implementato al fine di validare modelli e strategia di *mapping* elaborati in questo lavoro di tesi. In questa sezione vengono introdotti i dettagli di implementazione del simulatore descrivendo, in particolare, il formato dei *file* utilizzati, le strutture dati utilizzate internamente ed i risultati prodotti dal programma stesso. La descrizione dei dettagli di implementazione di articola quindi su 3 paragrafi:

- nel Paragrafo A.1.1 vengono introdotti i comandi impartibili al simulatore tramite linea di comando;
- nel Paragrafo A.1.2 vengono introdotti il formato dei *file* utilizzato e l'implementazione del modello di applicazione sviluppato nel Capitolo 3 tramite la descrizione delle strutture dati utilizzate;
- nel Paragrafo A.1.3 vengono infine introdotti i *file* di output prodotti dall'esecuzione del simulatore.

A.1.1 Comandi impartibili al Simulatore

Come introdotto nella Sezione 5.1, il Simulatore accetta diversi comandi che permettono di specificare opzioni quali i nomi dei *files* di input da utilizzare e la risoluzione temporale della simulazione. In Tabella A.1 sono riportate tutte le opzioni specificabili e il loro significato.

Tabella A.1: Elenco dei comandi impartibili al Simulatore

Comando	**Signficato**
a	Permette di specificare il nome del file eXtensible Markup Language (XML) contenente la definizione dell'architettura. Se non viene specificato verrà utilizzato come nome di default *architecture.xml*
p	Permette di specificare il nome del file XML contenente la dichiarazione dei programmi. Se non viene specificato verrà utilizzato *program.xml* come nome di default.
b	Permette di specificare il nome dei *file* contenenti il comportamento delle applicazioni e dei guasti. Il nome va specificato **senza** estensione. I due *file* devono inoltre essere memorizzati con i suffissi "App"per le applicazioni e "Flt"per i guasti. Se l'opzione non viene specificate viene utilizzato come nome di default *behavior*
t	Permette di specificare la risoluzione della simulazione, in millisecondi
d	Permette di abilitare i messaggi di *debug*
h	Visualizza le istruzioni di utilizzo del comando per lanciare il Simulatore

A.1.2 Implementazione modello e strutture dati

Come evidenziato in Figura 5.1 e descritto nel Paragrafo 5.1.1, il simulatore, per funzionare correttamente, necessita di alcuni file contenenti i dati

necessari alla corretta inizializzazione delle strutture dati utilizzate per implementare i modelli presentati nel Capitolo 3. In questo paragrafo vengono quindi descritti i formati dei file di informazioni utilizzati e le strutture dati utilizzate per implementare i modelli introdotti nel Capitolo 3.

Files dati

Per inizializzare correttamente il simulatore, devono essere forniti quattro *file* in formato XML contenenti le seguenti informazioni:

- la caratterizzazione dell'architettura in termini di risorse disponibili. Questa opzione è stata implementata per poter utilizzare il simulatore anche con configurazioni dell'architettura p2012 diverse da quella standard riportata in [6];
- la descrizione dei programmi: per ogni programma deve essere fornita la descrizione caratterizzata dai parametri introdotti nel Capitolo 3, È inoltre stato implementato anche un generatore casuale di applicazioni qualora si vogliano effettuare test senza avere a disposizione un file di descrizione delle applicazioni;
- la descrizione del comportamento dei programmi contenente l'elenco dei programmi da eseguire in fase di simulazione ed il loro tempo di arrivo;
- la descrizione del comportamento dei guasti all'interno dell'architettura; con questo file è possibile specificare dove e quando un guasto può accadere, rendendo così possibile testare il comportamento dell'architettura e dell'euristica anche in presenza di uno o più guasti.

Per ciascuno di questi *files* è stato quindi creato uno schema XML che permette di definire esattamente la struttura dei dati contenuti in esso.

Lo schema del **file di descrizione dell'architettura** è illustrato in Figura A.1; esso è stato creato in modo tale da poter assegnare tutti i parametri

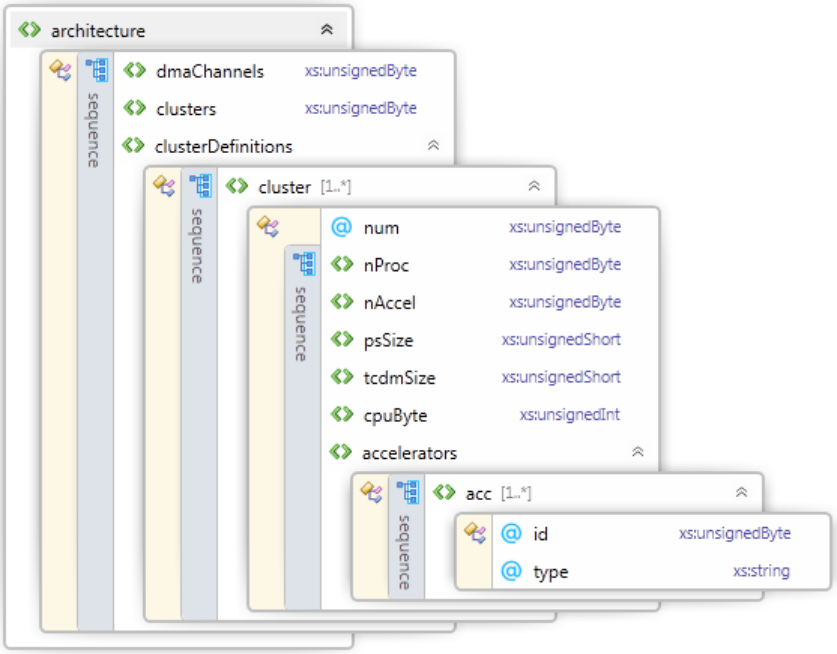


Figura A.1: Schema XML per il file di definizione dell'architetura

di descrizione dell'architettura individuati nella Sezione 3.2, oltre a permettere la specifica del numero di clusters e delle risorse presenti su di essi.

La struttura dello schema è la seguente:

- i primi due parametri definiti, **dmaChannels** e **clusters** permettono di specificare, rispettivamente, il numero di canali Direct Memory Access (DMA) utilizzati per le comunicazioni tra *fabric controller* e *cluster* (il valore predefinito è 3) e il numero di *cluster* presenti all'interno dell'architettura (valore predefinito 16);

- dopo i due parametri precedenti vengono specificati i *cluster*. Nella lista **clusterDefinition** devono essere definiti un numero di elementi di tipo **cluster** pari al valore del parametro **cluster**. Per ciascun *cluster*, deve poi essere specificato l'attributo **num**, che identifica il *cluster* e i parametri **nProc, nAccel, psSize, tcdmSize** e **cpuByte**, che speci-

ficano rispettivamente numero di CPU presenti nel *cluster*, numero di acceleratori configurati, grandezza delle due aree di memoria P$ e TCDM e, infine, il *byte* di configurazione delle CPU, in termini di unità funzionali presenti, come descritto nel Paragrafo 3.5.2;

- per ogni acceleratore configurato, è infine presente una lista di elementi di tipo **acc** contenenti identificativo dell'acceleratore (attributo **id**) e tipo dell'acceleratore (attributo **type**).

Lo schema di definizione delle **applicazioni** risulta più complicato, in quanto per ogni applicazione deve essere definita l'eventuale suddivisione in sub-applicazioni (un'applicazione non divisa in sub-applicazioni viene descritta come un'applicazione contenente un unica sub-applicazione) e per ogni sub-applicazione deve essere definito il *task-graph*. Lo schema utilizzato è quello rappresentato in Figura A.2.

La lista di elementi di tipo **application** definisce l'elenco delle applicazioni utilizzate per la simulazione e ha come unico vincolo quello di non poter essere vuota. Ogni elemento **application** definisce quindi un'applicazione ed è strutturato nel seguente modo:

- per ogni applicazione vengono specificati inizialmente l'identificativo dell'applicazione, mediante il parametro **id** e il numero di clusters richiesti, mediante il parametro **reqClusters**;

- vengono poi specificate le sub-applicazioni mediante gli elementi del tipo **subApplications**. Devono essere presenti un numero di elementi **subApplications** pari al valore del parametro **requClusters**;

- per ogni sub-applicazione vengono definiti i parametri **id, reqFus** e **reqCpu** che specificano, rispettivamente, identificativo della sub-applicazione, le unità funzionali richieste e il numero di Processing Element (PE) richiesti;

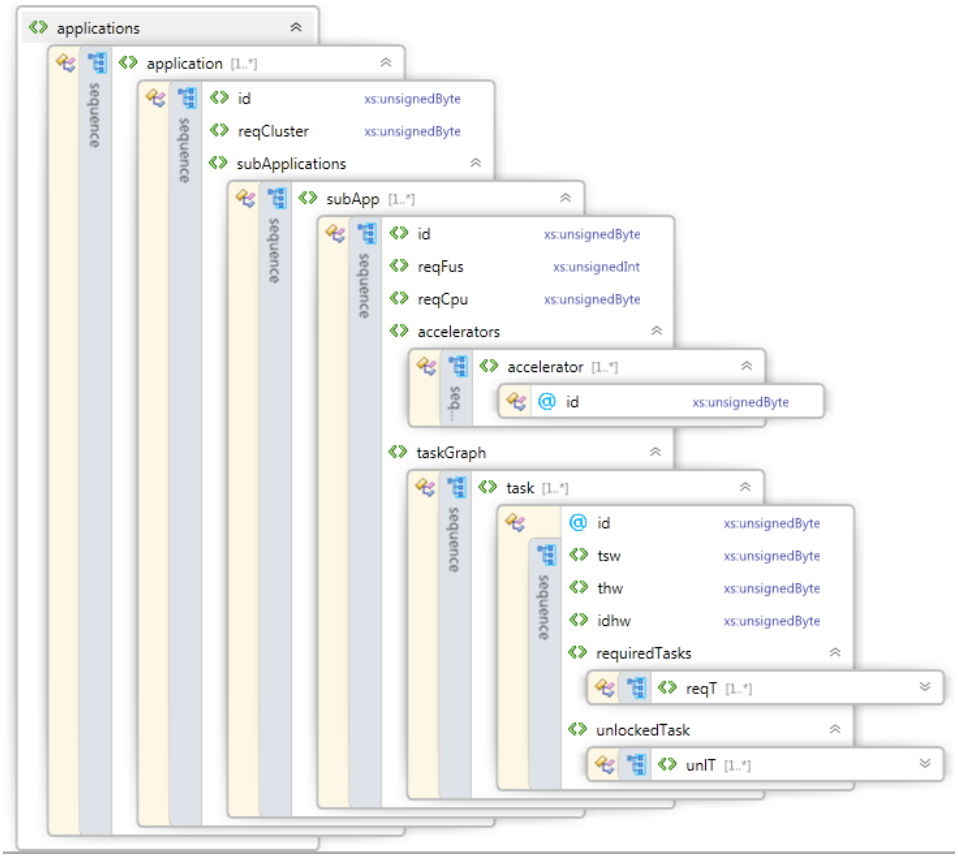


Figura A.2: Schema XML per il file di definizione delle applicazioni

- vengono quindi introdotte, per ogni sub-applicazione, la lista degli acceleratori *hardware* che possono essere utili all'esecuzione della sub-applicazione (lista di elementi **accelerator**) e il *task-graph* della sub-applicazione, definito come lista di elementi di tipo **task**;
- per ogni acceleratore viene specificato solo l'attributo **id**, che serve per identificare quale acceleratore viene richiesto;
- per ogni elemento **task** della lista **taskGraph** vengono invece specificati l'attributo **id**, che contiene l'identificativo del processo, i parametri **tsw, thw** e **idHw**, che servono per specificare i tempi di esecuzione in *software*, i tempi di esecuzione in *hardware* e l'acceleratore sul quale il processo può essere eseguito in *hardware* e infine vengono specifi-

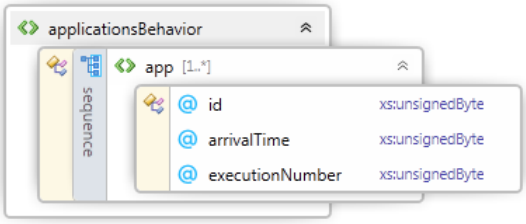


Figura A.3: Schema XML per il file di definizione del comportamento delle applicazioni

cate le liste dei processi predecessori (lista **requiredTasks**) e dei processi successori (lista **unlockedTasks**) all'interno del *task-graph* della sub-applicazione.

Lo schema di definizione del **comportamento delle applicazioni** risulta invece molto più semplice rispetto ai due schemi precedenti. Esso è illustrato in Figura A.3.

Lo schema è composto dalla sola lista **applicationsBehavior** composta da elementi di tipo **app**, ciascuno dei quali ha tre attributi:

- l'attributo **id**, che identifica l'applicazione da eseguire;
- l'attributo **executionNumber**, che identifica il numero di esecuzione dell'applicazione, necessario al simulatore per poter riconoscere tra esecuzioni diverse della stessa applicazione;
- l'attributo **arrivalTime** che contiene l'istante in cui l'applicazione deve essere eseguita dal simulatore.

Lo schema di definizione del **comportamento dei guasti** ricalca, dal punto di vista della struttura, lo schema di definizione del comportamento delle applicazioni.

Esso è illustrato in Figura A.4 e contiene una lista di elementi **fault**, denominata **faultList** che a loro volta contengono l'attributo **id**, che contiene l'identificatore univoco del guasto e 5 parametri:

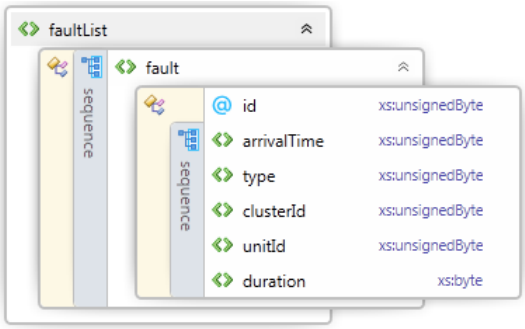


Figura A.4: Schema XML per il file di definizione del comportamento dei guasti

- **arrivalTime**, che contiene il tempo di arrivo, in millisecondi dall'inizio della simulazione, del guasto;
- **type** identifica il tipo di guasto ed ha due possibili valori *0* nel caso di guasto permanente e *1* nel caso di guasto transitorio;
- **clusterId**, che contiene l'identificativo del *cluster* sul quale di verifica il guasto;
- **unitId**, che contiene l'identificativo del PE interno al *cluster* sul quale si verifica il guasto;
- **duration**, che contiene la durata del guasto in millisecondi, nel caso il guasto sia transitorio, mentre contiene il valore -1 in caso di guasto permanente.

Strutture Dati

Per rendere utilizzabili dal programma le informazioni contenute all'interno dei *file* XML sono state implementate le corrispondenti strutture in linguaggio C++. In questo sotto-paragrafo vengono illustrate le strutture utilizzate dal simulatore per memorizzare i dati letti dai *file* XML. Per quanto riguarda l'**architettura** del System-on-Chip (SoC), sono state utilizzate quattro strutture, riportate in Figura A.5.

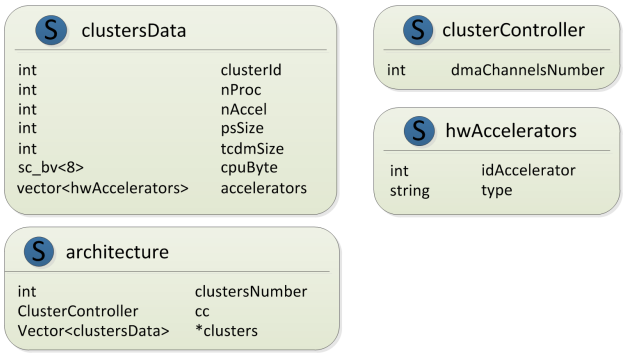


Figura A.5: Strutture C++ utilizzate per i dati dell'architettura

La struttura principale rimane **Architecture**, che contiene al suo interno informazioni sul numero di *cluster* (**clustersNumber**), un campo di tipo **clusterController**, che contiene i dati del *cluster controller* (**cc**) e un vettore di tipo **clustersData**, che contiene le informazioni di ogni *cluster* (**cluster**). La struttura **clusterController** contiene, nella versione implementata, un unico campo, contenente il numero di canali DMA (**dmaChannelsNumber**) a disposizione del componente per comunicare con i clusters mentre la struttura **clustersData** contiene tutte le informazioni necessarie a caratterizzare un *cluster* quali identificativo del *cluster* stesso (**clusterId**), numero di processori presenti (**nProc**), grandezza, in KiloBytes, delle memorie a disposizione dei processori (**pSSize** e **tcdmSize**), byte di configurazione dei processori (**cpuByte**), così come descritto nel Paragrafo 3.5.2. È infine indicato il numero di acceleratori (**nAccel**) presenti, unitamente a un vettore di elementi **hwAccelerators** (**accelerators**), che contengono a loro volta l'identificativo di ciascun acceleratore (**idAccelerator**) ed il tipo (**type**).

Le strutture utilizzate per memorizzare le informazioni sulle applicazioni, sono invece riportate in Figura A.6. Anche per le applicazioni sono state utilizzate una serie di strutture tra loro correlate. La struttura principale risulta essere, in questo caso, la struttura **application**, che contiene l'identificativo dell'applicazione (**applicationId**), in numero di cluster richie-

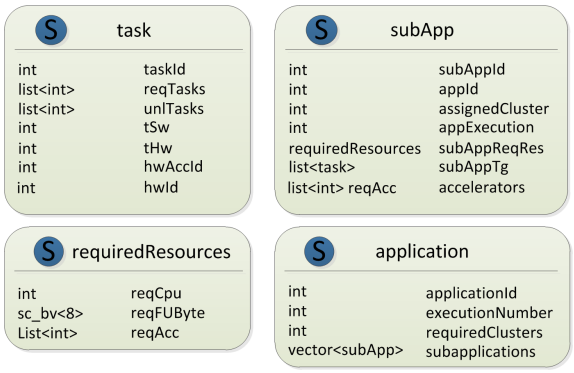


Figura A.6: Strutture C++ utilizzate per i dati delle applicazioni

sti (**requiredClusters**) e un vettore di elementi **subApp** (**subApplications**), che contengono le informazioni di ogni sub-applicazione in cui l'applicazione è composta. La struttura **subApp** contiene quindi l'identificativo dell'applicazione a cui appartiene (**appId**), l'identificativo della sub-applicazione (**subAppId**), l'identificativo del *cluster* a cui viene assegnata (*assignedCluster*, campo inizializzato dal *dispatcher*), un elemento di tipo **requiredResources** (**subAppReqRes**), che contiene informazioni sulle risorse richieste dalla sub-applicazione e una lista di elementi di tipo **task** (**subAppTg**), che rappresenta il *task-graph* dell'applicazione. Nella struttura **requiredResources** sono contenuti i dati riguardanti le risorse richieste dalla sub-applicazione, utili al *dispatcher* per poter eseguire il *mapping* delle sub-applicazioni. Queste informazioni comprendono il numero di CPU richieste (**reqCpu**), il byte delle unità funzionali richieste (**reqFUByte**, il cui formato è riportato nel Paragrafo 3.5.2) e una lista contenente gli identificativi di tutti gli acceleratori richiesti (**reqAcc**). La struttura **task**, invece, contiene tutte le informazioni riguardanti un singolo processo, in particolare sono presenti l'identificativo del processo stesso (**taskId**), la lista dei predecessori (**reqTasks**), la lista dei successori (**unlTasks**), il tempo di esecuzione, in millisecondi, in *software* (**tSw**) e, infine, il tempo di esecuzione in *hardware* (*tHw*) e l'identificativo dell'acceleratore richiesto(*hwId*).

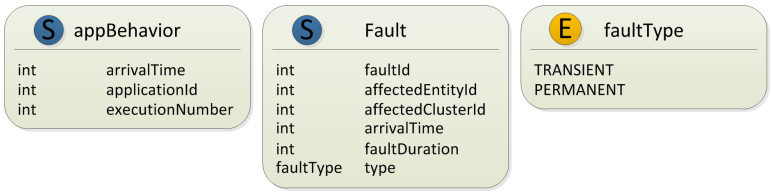


Figura A.7: Strutture C++ utilizzate per i dati sul comportamento delle applicazioni e dei guasti

Per quanto riguarda il comportamento delle applicazioni, invece, i relativi dati sono memorizzati all'interno di strutture di tipo **appBehavior**, rappresentate in Figura A.7.

La struttura risulta molto semplice e composta solo da tre campi: il campo **arrivalTime**, che contiene il momento di arrivo dell'applicazione, in millisecondi; il campo **applicationId**, che contiene l'identificativo dell'applicazione da eseguire ed il campo **executionNumber**, che contiene il numero di esecuzione dell'applicazione.

I dati sui guasti vengono invece memorizzati utilizzando una struttura ed un'enumerazione, chiamata **faultType** che permette di identificare il tipo di guasto. Essa contiene due valori, il valore *PERMANENT*, che identifica un guasto permanente e il valore *TRANSIENT*, che identifica un guasto temporaneo. I dati sui guasti sono invece contenuti all'interno della struttura **fault**, che contiene l'identificativo del guasto (**faultId**), *cluster* e PE affetti dallo stesso (*affectedClusterId* e *affectedEntityId*), tempo di avvenimento del guasto (**arrivalTime**), tipo del guasto (**type**) e durata del guasto (**faultDuration**), campo che verrà inizializzato a -1 nel caso il guasto sia di tipo permanente.

A.1.3 Output di simulazione

In questo paragrafo vengono descritti i file di output prodotti dal simulatore. Ogni simulazione infatti produce i seguenti file di output:

- un file di log contenente tutte le informazioni prodotte dal simulatore durante la sua esecuzione;
- un file riassuntivo con le statistiche di esecuzione per ciascuna applicazione;
- una serie di script *GNUPlot* che permettono di creare i grafici di esecuzione dei processi delle sub-applicazioni sui vari PE presenti in un *cluster*.

File di log

Il file di log è stato implementato per permettere di ricostruire l'esatta cronologia di quanto accade durante la simulazione. Esso contiene una copia esatta dei messaggi che vengono visualizzati a terminale durante l'esecuzione del programma. Un esempio del file di Log è riportato in figura A.8.

Come è possibile notare in Figura A.8, la struttura dei messaggi è molto regolare e permette di andare a identificare velocemente la funzione all'interno della quale il messaggio è stato notificato, l'entità che ha notificato il messaggio e l'istante temporale in cui il messaggio stesso è stato inviato. Ogni riga infatti inizia con un codice che rappresenta l'identificativo dell'entità responsabile dell'invio del messaggio, seguito dal tempo di simulazione e dal messaggio vero e proprio.

Statistiche riassuntive

Il file *executionInfo.txt* è il file di output più importante prodotto dal simulatore, esso infatti contiene tutte le informazioni di esecuzione di ogni applicazione. La struttura del file è regolare e le informazioni sono riportate in ordine cronologico dalla prima applicazione eseguita all'ultima. Per prima cosa vengono quindi stampati i dati riguardanti l'applicazione, ov-

```
105 - AT 206 ms================= END Cluster::verifyRequisites - 105 ================================
105 - AT 206 ms - SOFTWARE EXECUTION FOR TASK 4
105 - CPU 0occupied
105 - AT 206 ms - free cpu found 1
105 - AT 206 ms - Task assigned to CPU number 1
105 - AT 206 ms ============= Cluster::sendMessageToExecutionUnit for cluster 105=================
105 - AT 206 ms - Transaction in Cluster created, send it to NoC
105 - AT 206 ms Transaction:0x428a130
1001 - AT 206 ms ======================= clusterCPU::nb_transport_fw START =======================
1001 - AT 206 ms TASK ASSIGNED TO CLUSTER, INFO:
1001 - AT 206 ms TASK ID: 4
1001 - AT 206 ms TSW: 30
1001 - AT 206 ms CPU NUMBER: 1
1001 - AT 206 ms NOTIFY tlm::TLM_UPDATED and trigger event
Transaction 0x428a130 complete
1001 - AT 206 ms ======================= clusterCPU::nb_transport_fw END =======================
105 - AT 206 ms ============= END Cluster::sendMessageToExecutionUnit-105=================
105 - AT 206 ms - Sending task to cpu, task number: 1
1001 - AT 206 ms CPU number 1, executing task 4, in software. tsw=30ms
1000 - AT 208 ms SENDING TO CLUSTER CONTROLLER THE EXECUTION COMPLETE ADVICE
1000 - AT 208 ms============= ClusterCPU::sendMessageToCC =================
1000 - AT 208 ms Transaction in Cluster created, send it to NoC
1000 - AT 208 ms Transaction:0x428a260
112 - AT 208 ms=================Cluster::nb_transport_fw - 112 ================================
1000 - AT 208 ms Transaction 0x428a260 complete
112 - AT 208 ms - ID TASK EXECUTED: 8
112 - AT 208 ms - TASKS EXECUTED: 2
112 - AT 208 ms - TASKS TOTAL NUMBER: 3
112 - AT 208 ms=================Cluster::sendSyncMessage - 112 ================================
SENDING SYNC MESSAGES FOR TASK 8
112 - AT 208 ms================= END Cluster::sendSyncMessage - 112 ================================
112 - AT 208 ms - All parallel tasks executed, notify it
112 - AT 208 ms - Task executed, notify it
112 - AT 208 ms - CPU 0 freed, at time: 208 ms
112 - AT 208 ms ================= END Cluster::nb_transport_fw - 112 ================================
```

Figura A.8: Esempio di file di Log

vero identificativo della stessa, numero di esecuzione e numero di sub-applicazioni presenti. Quindi, per ogni sub-applicazione, vengono stampate le informazioni riguardanti il *cluster* a cui sono state assegnate, i tempi di invio, di inizio esecuzione e di fine esecuzione e viene anche riportato l'*overhead* temporale di comunicazione, inteso come tempo di trasferimento di dati e codice dal *fabric controller* al *cluster* designato attraverso la NoC. Infine, per ogni processo di ogni sub-applicazione vengono riportati il tipo di esecuzione ottenuta, il PE su cui il processo è stato assegnato, l'istante temporale di inizio esecuzione e quello di fine esecuzione. Un esempio di quanto riportato nel file delle statistiche è visibile in Figura A.9.

```
==========================APPLICATION EXECUTION INFORMATIONS=======================
ID APPLICATION: 41
EXECUTION NUMBER: 41
SUBAPPLICATIONS NUMBER: 4

SUBAPPLICATION 1=======================================
ASSIGNED TO CLUSTER: 1
SENDED AT t=5ms
STARTED AT t=25ms
ENDED AT t=85ms

COMMUNICATION OVERHEAD:20ms

SUBAPPLICATION TASKS INFORMATIONS======================
NUMBER OF TASKS: 3

TASK NUMBER: 1
SOFTWARE EXECUTED
ASSIGNED TO CPU: 0
STARTED AT t=25ms
ENDED AT t=45ms
```

Figura A.9: Esempio di file delle statistiche

Script Gnuplot

I file contenti script GNUPlot rappresentano il terzo e ultimo tipo di *output* prodotto dal simulatore. Essi rappresentano graficamente quanto contenuto nel file delle statistiche riassuntive, permettendo di visualizzare su un diagramma di Gantt l'andamento temporale dell'esecuzione di una sub-applicazione su un *cluster*. Viene creato uno *script* per ogni sub-applicazione eseguita su ogni cluster.

A.2 BenchMark

Il secondo programma implementato per validare la strategia di *mapping* adottata in questo lavoro di tesi è un *BenchMark*, basato sul simulatore introdotto nella Sezioni 5.1 e A.1 e opportunamente modificato per permettere il confronto con altre tecniche di *mapping*, nella fattispecie PL, LEC-DN e RW. In questo modo è stato possibile effettuare i confronti necessari a stabilire l'effetività della strategia di *mapping* descritta nel Capitolo 4. Rispetto al Simulatore, il *BenchMark* sfrutta la possibilità di generare applicazioni tramite un algoritmo randomico permettendo così di non utilizzare

i *file* XML di specifica delle applicazioni e del loro comportamento. In Tabella A.2 sono illustrati tutti i comandi che possono essere specificati al *Benchmark* tramite linea di comando.

Tabella A.2: Elenco dei comandi impartibili al programma *BenchMark*

Comando	**Signficato**
l	Specifica il numero di clusters per lato del SoC, il valore di default è 4
a	Specifica il numero di applicazioni da generare casualmente, questo comando è obbligatorio
t	Specifica il tempo massimo di simulazione, se non viene impartito questo comando, il *BenchMark* termina con l'esecuzione dell'ultima applicazione
f	Specifica il numero di unità guaste permanentemente, se tale numero è maggiore del doppio del valore specificato con l'opzione l, il comando viene ignorato in quanto il dispositivo risulterebbe guasto in ogni componente
p	Specifica la percentuale di clusters occupati all'inizio dell'esecuzione del *BenchMark*, il valore di default è 50%
s	Specifica che tipo di strategia utilizzare per l'euristica, *n* indica l'euristica che privilegia l'uniformità d'uso dei *cluster*, *i* indica l'euristica che privilegia la vicinanza tra i clusters utilizzati
d	Abilita i messaggi di *debug* che verranno visualizzati a schermo

La descrizione del programma è articolata su due Paragrafi:

- nel Paragrafo A.2.1 viene introdotto il flusso di esecuzione del *BenchMark*, illustrando come vengono eseguite le varie prove e come vengono raccolti i dati utilizzati per confrontare le tecniche;

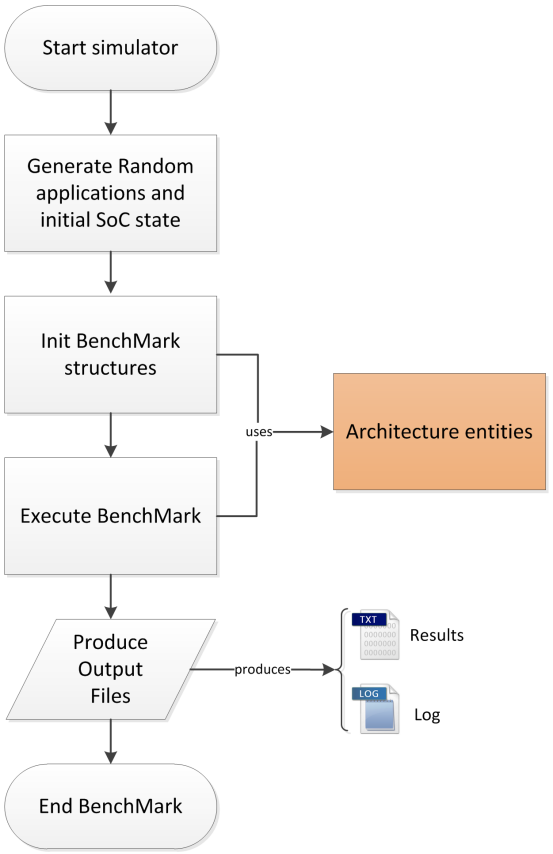


Figura A.10: Flusso di esecuzione del *BenchMark*

- nel Paragrafo A.2.2 vengono invece illustrati i file di *output* prodotti dal programma.

A.2.1 Flusso di esecuzione del BenchMark

Essendo il *BenchMark* basato sul Simulatore descritto nella sezione 5.1, la struttura interna del programma è analoga a quella del Simulatore stesso. La differenza tra *BenchMark* e Simulatore, risiede pertanto nelle operazioni svolte dai due programmi.

Il flusso di esecuzione del *BenchMark* è illustrato in Figura A.10. La principale differenza con il Simulatore risiede nella fase di inizializzazione del programma. Nel *BenchMark*, infatti, la fase di caricamento dei dati da file

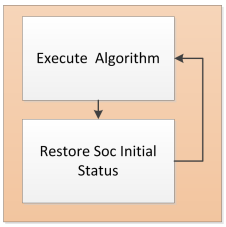


Figura A.11: Esecuzione del BenchMark

XML viene sostituita dalla fase di generazione, tramite algoritmo randomico, delle applicazioni e dello stato iniziale del SoC. In questa fase viene creato un numero di applicazioni pari a quanto specificato tramite linea di comando (vedi Tabella A.2) e viene inizializzato il SoC andando a marcare come occupati un numero di *cluster* pari alla percentuale specificata tramite linea di comando e andando ad escludere dall'esecuzione un numero di *cluster* pari al numero di guasti permanenti specificato da linea di comando. L'esecuzione prosegue poi in modo analogo a quanto fatto con la simulazione, andando a eseguire tutte le applicazioni utilizzando tutte e quattro le tecniche implementate all'interno del programma. Al termine dell'esecuzione delle tecniche, i dati raccolti durante lo svolgimento della simulazione vengono scritti sui file di *output* che rappresentano il prodotto finale dell'esecuzione del programma (vedi Paragrafo A.2.1) L'esecuzione del *BenchMark* è strutturata come riportato in Figura A.11. Per ogni algoritmo infatti, dopo l'esecuzione dello stesso vengono ripristinati lo stato iniziale del SoC e la stessa sequenza di applicazioni in modo tale che tutti gli algoritmi eseguiti vadano ad operare nella stessa situazione iniziale e con la stessa sequenza di arrivo delle applicazioni, permettendo un confronto preciso tra le quattro tecniche.

A.2.2 Output del BenchMark

Così come per il Simulatore, anche il benchMark produce una serie di *file* di output che permettono di verificare quanto accaduto durante l'e-

secuzione del programma e che permettono di andare a vedere i risultati prodotti dai quattro algoritmi. Vengono dunque generati sei diversi *file*:

- il *file* di log del programma;
- un *file* per ciascuna tecnica (per un totale di quattro *file*) contenente la cronologia di quanto accaduto durante l'esecuzione del relativo algoritmo;
- un *file* riassuntivo contenente i dati riepilogativi di tutte e quattro le tecniche ed utilizzato per effettuare le statistiche comparate.

File di Log

Il *file* di Log è analogo a quello generato dal Simulatore e illustrato in Figura A.8.

File della cronologia di esecuzione

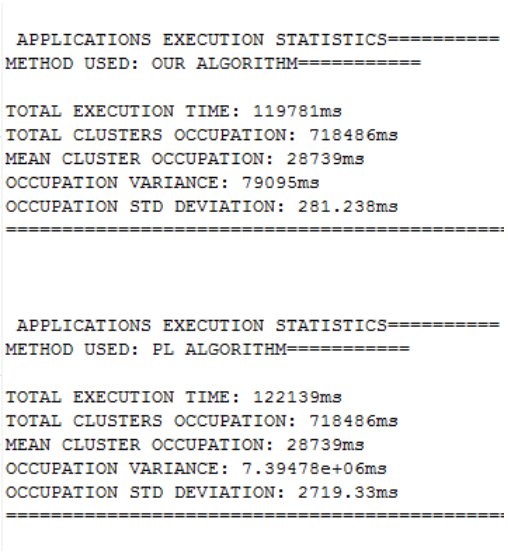
I *file* contenenti la cronologia di esecuzione rappresentano l'*output* più significativo del *BenchMark*. Viene creato infatti un *file* per ciascun algoritmo utilizzando la stessa struttura, in modo tale che sia possibile confrontare facilmente l'esecuzioni delle varie strategie di *mapping*. Un esempio del contenuto di questi *file* è visibile in Figura A.12. Per ogni applicazione eseguita vengono riportati identificativo di applicazione, tempo di arrivo e tempo di inizio dell'esecuzione, immediatamente seguiti dal numero di sub-applicazioni che compongono un'applicazione e dallo stato del SoC al momento dell'inizio dell'esecuzione. Per ogni sub-applicazione vengono riportati poi i dati identificativi e il cluster a cui la sub-applicazione è stata assegnata. Al termine dei dati delle sub-applicazioni viene poi riportato lo stato del SoC dopo il *mapping* dell'applicazione e i relativi tempi di attività di ogni *cluster*, in questo modo è possibile ricostruire la "storia"completa dell'esecuzione dell'algoritmo.

```
NEW APPLICATION READY TO BE DISPATCHED
APPLICATION INFO:
ID: 803
CURRENT TIME: 82179ms
ARRIVAL TIME: 82179
SUBAPP NUMBER: 1

CLUSTER STATUS BEFORE DISPATCHING:
 0  0  0 -1 -1
-1 -1 -1 -1 -1
 0  0 -1 -1 -1
 0 -1  0 -1  0
 0 -1 -1 -1 -1

 SUBAPPLICATION DISPATCHED: DISPATCHING INFORMATION:
 Id Subapplication: 0
 Application Id:803
 Execution Number:0
 Assigned to cluster:19

CLUSTER STATUS AFTER DISPATCHING:
CLUSTERS OCCUPIED: (-1)
 0  0  0 -1 -1
-1 -1 -1 -1 -1
 0  0 -1 -1 -1
 0 -1  0 -1 -1
 0 -1 -1 -1 -1
CLUSTERS UPTIME [ms]:
19030   18330   18580   18207   18697
18217   18991   18484   18710   18135
18432   18681   19158   18859   18341
18019   18967   18844   19438   18373
18706   18120   18281   18596   18554
```

Figura A.12: Esempio di file di cronologia

Statistiche Riepilogative

L'ultimo *file* prodotto dal programma è il *file* contenente le statistiche riepilogative dell'esecuzione degli algoritmi. Tale *file* è di fondamentale importanza per il confronto tra gli algoritmi in quanto in esso, per ogni algoritmo, vengono riportati il tempo totale di esecuzione della simulazione, il tempo medio di attività di ciascun *cluster*, la varianza e la deviazione standard dei tempi di attività dei *cluster*. Questi dati quindi permettono di verificare quanto l'algoritmo sviluppato in questo lavoro di tesi raggiunga gli obiettivi di esecuzione efficiente (tempo totale di esecuzione) e di omogeneità nell'uso dei *cluster* (deviazione standard dei tempi di attività). Un esempio del contenuto di questo *file* è riportato in Figura A.13.



```
 APPLICATIONS EXECUTION STATISTICS==========
METHOD USED: OUR ALGORITHM===========

TOTAL EXECUTION TIME: 119781ms
TOTAL CLUSTERS OCCUPATION: 718486ms
MEAN CLUSTER OCCUPATION: 28739ms
OCCUPATION VARIANCE: 79095ms
OCCUPATION STD DEVIATION: 281.238ms
=========================================



 APPLICATIONS EXECUTION STATISTICS==========
METHOD USED: PL ALGORITHM===========

TOTAL EXECUTION TIME: 122139ms
TOTAL CLUSTERS OCCUPATION: 718486ms
MEAN CLUSTER OCCUPATION: 28739ms
OCCUPATION VARIANCE: 7.39478e+06ms
OCCUPATION STD DEVIATION: 2719.33ms
=========================================
```

Figura A.13: Esempio di file riepilogativo

Bibliografia

[1] Lodewijk T. Smit, Johann L. Hurink, and Gerard J. M. Smit. Run-time mapping of applications to a heterogeneous soc. In *in Proceedings of the International Symposium on System-on-Chip (SoC 05), pp. 78–81*, 2005.

[2] Stefan Wildermann and J′urgen Teich. Run time mapping of adaptive applications onto homogeneous noc-based reconfigurable architectures. In *in Proceedings of the Conference on Field Programmable Technology (FPT 09), pp. 514–517*, 2009.

[3] Andreas Weichslgartner, Stefan Wildermann, and Jürgen Teich. Dynamic decentralized mapping of tree-structured applications on noc architectures. In *Proceedings of the Fifth ACM/IEEE International Symposium on Networks-on-Chip*, NOCS '11, pages 201–208, New York, NY, USA, 2011. ACM.

[4] Francesco Paterna, Luca Benini, Andrea Acquaviva, Francesco Papariello, Giuseppe Desoli, and Mauro Olivieri. Adaptive idleness distribution for non-uniform aging tolerance in multiprocessor systems-on-chip. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '09, pages 906–909, 3001 Leuven, Belgium, Belgium, 2009. European Design and Automation Association.

[5] Intel. High performance 32nm logic technology, 2010. [Online;].

[6] St Microelectronics, CEA. Platfrom 2012: A many-core programmable accelerator for ultra-efficient embedded computing in nanometer technology. *ST Datasheets*, 1(1):1–27, 2011.

[7] Marcelo Mandelli, Alexandre Amory, Luciano Ost, and Fernando Gehm Moraes. Multi-task dynamic mapping onto noc-based mpsocs. In *Proceedings of the 24th symposium on Integrated circuits and systems design*, SBCCI '11, pages 191–196, New York, NY, USA, 2011. ACM.

[8] Andreas Schranzhofer, Jian-Jia Chen, Luca Santinelli, and Lothar Thiele. Dynamic and adaptive allocation of applications on mpsoc platforms. In *Proceedings of the 2010 Asia and South Pacific Design Automation Conference*, ASPDAC '10, pages 885–890, Piscataway, NJ, USA, 2010. IEEE Press.

[9] Amit Kumar Singh, Thambipillai Srikanthan, Akash Kumar, and Wu Jigang. Communication-aware heuristics for run-time task mapping on noc-based mpsoc platforms. *J. Syst. Archit.*, 56:242–255, July 2010.

[10] Amit Kumar Singh, Wu Jigang, Alok Prakash, and Thambipillai Srikanthan. Efficient heuristics for minimizing communication overhead in noc-based heterogeneous mpsoc platforms. In *Proceedings of the 2009 IEEE/IFIP International Symposium on Rapid System Prototyping*, RSP '09, pages 55–60, Washington, DC, USA, 2009. IEEE Computer Society.

[11] Ewerson Carvalho, César Marcon, Ney Calazans, and Fernando Moraes. Evaluation of static and dynamic task mapping algorithms in noc-based mpsocs. In *Proceedings of the 11th international conference on System-on-chip*, SOC'09, pages 87–90, Piscataway, NJ, USA, 2009. IEEE Press.

[12] Ewerson Carvalho, Ney Calazans, and Fernando Moraes. Congestion-aware task mapping in noc-based mpsocs with dynamic workload. In *Proceedings of the IEEE Computer Society Annual Symposium on VLSI*, pages 459–460, Washington, DC, USA, 2007. IEEE Computer Society.

[13] Ewerson Carvalho, Ney Calazans, and Fernando Moraes. Heuristics for dynamic task mapping in noc-based heterogeneous mpsocs. In *Proceedings of the 18th IEEE/IFIP International Workshop on Rapid System Prototyping*, pages 34–40, Washington, DC, USA, 2007. IEEE Computer Society.

[14] Ost Mandelli, Guindani Carara, Medeiros Gouvea, and Moares. Energy-aware dynamic task mapping for noc-based mpsocs. In *Proceedings of the 2011 IEEE Symposium on Circuit and Systems (ISCAS*, pages 34–40, Washington, DC, USA, 2011. IEEE Computer Society.

[15] Marshall L. Fisher, R. Jaikumar, and Luk N. Van Wassenhove. A multiplier adjustment method for the generalized assignment problem. *Management Science*, 32(9):1095–1103, September 1986.

[16] Jingcao Hu and Radu Marculescu. Energy-aware mapping for tile-based noc architectures under performance constraints. In *Proceedings of the 2003 Asia and South Pacific Design Automation Conference*, ASP-DAC '03, pages 233–239, New York, NY, USA, 2003. ACM.

[17] Gereon Frahling and Christian Sohler. A fast k-means implementation using coresets. In *Proceedings of the twenty-second annual symposium on Computational geometry*, SCG '06, pages 135–143, New York, NY, USA, 2006. ACM.

[18] Sanjoy Dasgupta. Performance guarantees for hierarchical clustering. In *Proceedings of the 15th Annual Conference on Computational Learning*

Theory, COLT '02, pages 351–363, London, UK, UK, 2002. Springer-Verlag.

[19] Chen-Ling Chou, Ümit Y. Ogras, and Radu Marculescu. Energy- and performance-aware incremental mapping for networks on chip with multiple voltage levels. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 27(10):1866–1879, 2008.

[20] Lin Huang, Feng Yuan, and Qiang Xu. On task allocation and scheduling for lifetime extension of platform-based mpsoc designs. *IEEE Trans. Parallel Distrib. Syst.*, 22:2088–2099, December 2011.

[21] Dr. Maroun Ojail, Dr. Raphael David, Dr. Karim Ben Chehida, Dr. Yves Lhuillier, Dr. Luca Benini. Synchronous reactive fine grain tasks management for homogeneous many-core architecture. *ST Datasheets*, 1(1):144–150, 2011.

[22] St Microelectronics. *STxP70-4B processor supercore datasheet*. St Microelectronics, 2010.