POLITECNICO DI MILANO

Facoltà di Ingegneria dell'Informazione

Corso di Laurea Specialistica in Ingegneria Informatica

A^2B: Application to Bitstream

Flusso semi-automatico per la creazione di MPSoC riconfigurabili

Relatore: Prof. Donatella SCIUTO

Correlatore: Ing. Christian PILATO

Tesi di Laurea di:

Gianluca DURELLI

Matricola n. 751301

Anno Accademico 2010–2011

Sommario

Negli ultimi anni la complessità dei sistemi informatici è aumentata esponenzialmente e tecnologie come le architetture multicore e sistemi eterogenei hanno iniziato ad essere adottate su larga scala. Fino a pochi anni or sono queste tecnologie erano riservate alle fasce di mercato più alte oppure alle soluzioni professionali, mentre al giorno d'oggi sono disponibili anche nelle fasce di mercato più economiche sistemi che utilizzano fino a quatto oppure otto core [1].

L'elevata complessità di questi sistemi si ripercuote direttamente sul loro flusso di sviluppo; nei primi anni in cui questi sistemi venivano sviluppati, tale flusso consisteva nella progettazione del circuito su carta per poi passare direttamente alla sua prototipazione e alla successiva fase di test per verificarne la correttezza. Tuttavia, proprio a causa dell'elevata complessità di questi sistemi, ci si è subito resi conto che questo flusso di lavoro risultava insostenibile per via dell'aumento esponenziale dello spazio di ricerca. Questa necessità di avere strumenti capaci di assistere i progettisti ha dato incentivo alla ricerca nel campo; i lavori svolti hanno permesso di sviluppare tecniche e strumenti in grado di semplificare notevolmente il lavoro dei progettisti; recentemente IBM ha rilasciato un video per celebrare i 50 anni del settore [2].

Il continuo progresso tecnologico inoltre ha permesso a questo settore di ricerca di essere sempre particolarmente attivo con l'obiettivo di proporre nuove tecniche capaci di assistere il progettista nel fronteggiare i nuovi problemi che esso deve affrontare durante lo sviluppo dei sistemi hardware più recenti. In questo scenario un ruolo di primaria importanza è riservato ai sistemi riconfigurabili, in particolare alle Field Programmable Gate Array (FPGA). Questi dispositivi hanno assunto con il passare del tempo una sempre maggiore importanza per lo sviluppo di sistemi embedded; la loro capacità di essere riconfigurati, sia dopo la fabbricazione, sia durante l'esecuzione, permette a questi dispositivi di simulare il sistema in fase di sviluppo senza effettivamente realizzarne un prototipo fisico, procedimento che si rivelerebbe molto più costoso. Questo permette di effettuare una fase di test per verificare la correttezza del design proposto contenendo di molto i costi; un errore scoperto in fase di test su un

dispositivo riconfigurabile infatti può essere risolto correggendo l'errore nel circuito e riconfigurando il dispositivo per effettuare un nuovo test, mentre nel caso di prototipazione su chip fisico si dovrebbe affrontare di nuovo il design e la fase di produzione. La capacità di essere riconfigurati, inoltre, rende questi dispositivi fondamentali per la fase di esplorazione dello spazio di ricerca in quanto permette di generare diverse implementazioni del sistema in via di sviluppo e di testarne direttamente il comportamento. La caratteristica delle FPGA di poter essere riconfigurate anche dinamicamente permette inoltre a questi dispositivi di poter essere usati all'interno di un sistema come acceleratori hardware a fianco di un generico General Purpose Processor (GPP) per fornire la possibilità di accelerare uno specifico algoritmo qualora ve ne sia la necessità; sistemi di questo tipo iniziano ad essere presenti negli ambienti di ricerca soprattutto in campo robotico [3, 4, 5, 6], ma se ne ipotizza una possibile introduzione anche sul mercato nel campo per esempio dei dispositivi mobili [7]. In questo tipo di sistemi, il GPP svolge il doppio ruolo di unità processante e di controllore dell'intero sistema; qualora esso identifichi un vantaggio, in termini di prestazioni o di consumo di potenza, nell'eseguire un particolare algoritmo sulla FPGA gli è sufficiente riconfigurarla per svolgere la funzione richiesta. Tuttavia, nonostante questi dispositivi riconfigurabili diminuiscano i costi di sviluppo per i dispositivi hardware ed il loro time-to-market, uno dei loro maggiori svantaggi è l'alta esperienza richiesta nel campo dell'hardware design e nei Hardware Description Languages (HDLs) per riuscire a programmarli. Questo inconveniente limita fortemente la loro diffusione in quanto ne limita l'utilizzo ai soli designer hardware, quando invece anche un programmatore potrebbe trarre enormi vantaggi da un'eventuale implementazione hardware dell'algoritmo che sta sviluppando. Per questo motivo le ricerche in questo settore sono concentrate nel provare a ridurre lo sforzo richiesto per lavorare con questi dispositivi, provando a sollevare lo sviluppatore dall'onere di scrivere ogni singola parte del sistema direttamente in HDL. Molti lavori affrontano la problematica di come sia possibile ottenere una descrizione in HDL dell'applicazione in fase di sviluppo a partire da una sua descrizione in un qualche linguaggio di alto livello (generalmente C, C++ o Fortran). Lo stato dell'arte degli strumenti che affrontano questo problema, chiamato comunemente High Level Synthesis (HLS), pur con qualche restrizione è in grado di generare un codice HDL funzionante a partire da un linguaggio di alto livello; le restrizioni generalmente imposte da questi strumenti riguardano il sottoinsieme dei costrutti del linguaggio di origine che è possibile utilizzare per codificare l'algoritmo. Una volta ottenuto il codice HDL corrispondente all'applicazione in analisi è possibile integrarlo all'interno del sistema in fase di sviluppo, operazione normalmente effettuata

in modo manuale.

Tuttavia le applicazioni in oggetto spesso non beneficiano da una completa esecuzione sull'acceleratore hardware; nella maggioranza dei casi tali applicazioni possono essere divise in sottoparti, chiamate comunemente task; ognuno di questi task può beneficiare o meno di un'implementazione hardware. Un'altra branca di ricerca, indicata con il termine Hw/Sw Co-design, si occupa appunto di identificare le parti di un'applicazione che beneficiano di un'implementazione hardware andando così ad identificare un partizionamento tra i task che andranno poi eseguiti sull'acceleratore hardware e quelli che sarà più opportuno eseguire su di un GPP. I task che faranno parte della partizione che andrà eseguita in hardware possono quindi essere tradotti in HDL sfruttando gli strumenti di HLS introdotti in precedenza.

In ultima analisi, la ricerca si è concentrata anche nel fornire framework che affiancassero lo sviluppatore lungo il completo flusso di sviluppo al fine di ridurre al minimo gli sforzi che devono essere da lui compiuti. Questi strumenti hanno l'obbiettivo di riunire le fasi descritte in precedenza al fine di automatizzarle completamente prendendosi cura della gestione dell'intero flusso delle informazioni all'interno dello strumento. Il generico flusso di lavoro proposto da questi framework consiste nel ricevere in ingresso un'applicazione codificata in un linguaggio di alto livello, una descrizione dell'architettura di riferimento e un insieme di scelte progettuali dello sviluppatore; il framework realizza in modo automatico, o semi automatico, le fasi di Hw/Sw Codesign e di HLS occupandosi inoltre di generare le necessarie interfacce per l'integrazione delle parti generate nell'architettura di riferimento. L'output generalmente fornito da questi strumenti è un sistema embedded pronto per essere implementato su di un dispositivo, eventualmente riconfigurabile, per la fase di test. Questi framework assumono un ruolo fondamentale nello sviluppo di sistemi embedded in quanto riescono a velocizzarne enormemente il processo di sviluppo ed a semplificare enormemente l'integrazione delle varie parti del sistema; in questo modo il designer si può concentrare solamente sul trovare l'implementazione migliore del sistema che sta sviluppando usando questi tool come un importantissimo supporto alla fase di esplorazione delle soluzioni disponibili.

Sebbene questi strumenti riescano effettivamente a velocizzare il processo di sviluppo dei sistemi embedded uno dei loro maggiori inconvenienti risiede nel fatto che il loro flusso di sviluppo non permette di sfruttare a pieno le potenzialità messe a disposizione dai nuovi dispositivi. Tecnologie come la riconfigurabilità dinamica parziale sebbene non siano una novità, sono tornate in auge a fronte dello sviluppo tecnologico degli ultimi anni; tale sviluppo ha permesso di avere a disposizione dispositivi riconfigurabili molto più ampi e dalle prestazioni molto

più elevate di quelle un tempo disponibili. Questi sviluppi hanno dato nuova linfa alla ricerca in questo ambito, ed i framework sviluppati in precedenza non sono più in grado di assistere lo sviluppatore nell'affrontare i nuovi problemi che nascono dalla disponibilità di queste nuove soluzioni, quali ad esempio riuscire a gestire in modo intelligente la riconfigurazione del dispositivo.

In particolare le nuove correnti di ricerca sono focalizzate nello sfruttare la riconfigurazione per sviluppare sistemi in grado di supportare, grazie alla riconfigurazione, più applicazioni in esecuzione in modo concorrente. Lo scopo di questo lavoro è colmare il vuoto tra i framework proposti nello stato dell'arte e il rinnovato interesse nello studio dei dispositivi riconfigurabili. L'obiettivo di questo lavoro di tesi è quello di sviluppare un framework in grado di assistere lo sviluppatore nello sviluppo di un sistema embedded eterogeneo; tale sistema deve essere in grado di sfruttare la riconfigurabilità per fornire un set di caratteristiche quali la gestione a runtime di più processi.

In questo lavoro verranno analizzate le problematiche che questo framework deve affrontare e si identificheranno caratteristiche che devono essere supportate dal framework e dall'architettura che viene creata dallo stesso. Questo lavoro propone in primo luogo un framework che supporti tutto quanto finora specificato ed inoltre propone un'architettura aderente alle richieste evidenziate al fine di poter testare il framework con diversi casi di studio. I contributi principali di questo lavoro di tesi sono quindi:

- la definizione di un'architettura riconfigurabile che possa utilizzare i core generati tramite la fase di HLS automatica;
- l'implementazione di un framework automatico per sviluppare applicazioni su questa piattaforma al fine di supportare l'esecuzione concorrente delle stesse; in particolare ci si è concentrati sulla automazione della creazione delle interfacce dei core hardware, dei driver di comunicazione tra i diversi elementi del sistema e dell'integrazione dei core generati nel sistema;
- la creazione di un livello di gestione software del dispositivo volto a garantire l'esecuzione corretta delle applicazioni gestendo a runtime le dipendenze tra i vari task e decidendo come riconfigurare i core nel miglior modo possibile in relazione alle diverse condizioni di lavoro nelle quali si trova il dispositivo.

I risultati forniti in questo lavoro, infine, in primo luogo validano il framework su di un caso di studio mostrandone la sua semplicità ed efficacia; inoltre mostrano come il framework sviluppato possa essere utile in fase di esplorazione dello spazio delle soluzioni; e in ultimo di come sia possibile generare sistemi che eseguano più applicazioni in modo concorrente sfruttando le capacità di riconfigurazione del dispositivo.

Il resto della trattazione è organizzata nel modo seguente:

- il Capitolo 1 introduce gli aspetti ed i problemi comuni ai framework volti allo sviluppo dei sistemi embedded e fornisce la definizione dei diversi passi che un flusso di sviluppo per questi sistemi deve affrontare;
- il Capitolo 2 presenta le definizioni relative ai concetti utili per la comprensione del lavoro in oggetto ed analizza i problemi che si hanno nelle varie fasi del flusso di sviluppo presentando le tecniche volte a risolverli;
- il Capitolo 3 presenta lo stato dell'arte analizzando i framework al momento disponibili ed analizzandone le caratteristiche alla luce degli obiettivi che si pone questo lavoro;
- il Capitolo 4 descrive l'approccio proposto per lo sviluppo del framework in oggetto evidenziando i problemi più importanti che deveno essere considerati e proponendo la soluzione scelta; più in dettaglio il capitolo tratta i problemi quali ad esempio il supporto che deve essere fornito dall'architettura sottostante, le caratteristiche che devono essere esposte allo sviluppatore ed i problemi relativi alla gestione runtime del sistema;
- il Capitolo 5 presenta quindi in dettaglio l'architettura hardware che viene proposta per testare il framework sviluppato; questo capitolo mostra inoltre la struttura dei cores che dovranno essere generati dal framework ed in quale modo è quindi possibile integrarli nel sistema in fase di sviluppo; infine il capitolo fornisce dettagli riguardo al modello di comunicazione che si è scelto di utilizzare;
- il Capitolo 6 descrive i dettagli implementativi del framework illustrando come sono stati affrontati i problemi evidenziati nel Capitolo 4;
- il Capitolo 7 approfondisce i dettagli implementativi relativi alla gestione run-time del sistema generato con il framework proposto;

- il Capitolo 8 mostra quindi alcuni risultati volti a dimostrare l'efficacia del flusso proposto mostrandone la sua facilità di utilizzo e analizzando le performance del sistema generato;
- infine il Capitolo 9 presenta le conclusioni e delinea alcuni sviluppi futuri che nascono da questo lavoro di tesi.

Indice

Elenco delle figure

Elenco delle tabelle

Elenco degli algoritmi

Elenco delle abbreviazioni

ASIC Application Specific Integrated Circuit . . . 3

CFG Control Flow Graph . . . 24

DAG Directed Acyclic Graph . . . 14

DFG Data Flow Graph . . . 72

FPGA Field Programmable Gate Array . . . 109

FSM Finite State Machine . . . 26

GCC GNU Compiler Collection . . . 24

GPP General Purpose Processor . . . 27

GPU Graphical Processing Unit . . . 3

HDL Hardware Description Language . . . 59

HLS High Level Synthesis . . . 24

HTG Hierarchical Task Graph . . . 15

ILP Integer Linear Programming . . . 18

KL Kernighan/Lin . . . 19

KPN Kahn Process Network . . . 27

MPSoC Multi Processors System on Chip . . . 108

NoC Network on Chip . . . 31

RTL Register Transfer Level . . . 35

XPS Xilinx Platform Studio . . . 27

Capitolo 1

Introduzione

Questo capitolo introduce gli argomenti discussi in questo lavoro. La Sezione 1.1 si descrive come il continuo sviluppo tecnologico, spinto da un'incessante ricerca per ottenere migliori performance, stia portando cambiamenti radicali nel campo delle architetture hardware e dei requisiti a loro richiesti; cambiamenti che si ripercuotono sugli strumenti necessari allo sviluppo di tali architetture. Dopo questa introduzione viene fornita la descrizione di un generico flusso per lo sviluppo di sistemi embedded che sarà usato come riferimento nel resto del lavoro; verranno presentati in questa sezione tutti i passi e i problemi che questo flusso deve affrontare; passi che verranno poi approfonditi ognuno in una propria sezione nel Capitolo 2.

1.1 Introduzione ai sistemi multiprocessore

I progressi tecnologici degli ultimi anni hanno determinato un punto di svolta per quanto riguarda i paradigmi di computazione; le classiche architetture single-core sono state progressivamente abbandonate e sostituite da quelle multi-core. Questo cambiamento è stato causato dal continuo aumento delle performance richieste ai sistemi informatici unitamente al fatto che tali performance non potevano più essere garantite dalle architetture single-core in quanto oramai giunte al loro limite fisico in termini di potenza dissipata, calore generato e frequenza di clock [9]. Date queste condizioni un cambio di direzione verso i sistemi multi-core si è reso obbligatorio ed è stato senza dubbio favorito dai progressi tecnologici quali la continua riduzione della dimensione dei transistor; analizzando i dati delle microarchitetture prodotte da Intel negli ultimi anni, riportati in Tabella 1.1, risulta evidente quanto affermato.

Tabella 1.1: Processo di fabbrica delle microarchitetture Intel [8, 10].

Microarchitectura	**Anno di rilascio**	**Processo di fabbrica**
Core	2006	65 nm
Nehalem	2008	45 nm
Sandy Bridge	2011	32 nm
Haswell	2013	22 nm

Il cambio nel processo di fabbrica dei processori e la loro continua miniaturizzazione ha permesso di condensare su di un singolo chip più elementi processanti. Questo ha spinto la ricerca ad esplorare architetture sempre più complesse e con un numero sempre maggiore di core in grado di accelerare le applicazioni al fine di garantire le performance loro richieste; Intel ha proposto un'architettura ad 80 core [11], AMD propone in commercio sistemi fino a 16 core, mentre progetti più ambiziosi hanno portato alla creazione di architetture con 1024 core [12]. Questa continua miniaturizzazione dei transistor ha quindi permesso di aggregare sempre più componenti su di un singolo chip andando quindi a realizzare sistemi chiamati appunto Multi Processors System on Chip (MPSoC). La famiglia delle architetture multiprocessore si divide in due grandi tipologie di architetture caratterizzate l'una dall'avere gli elementi processanti tutti uguali tra loro, identificate appunto come architetture omogenee, l'altra dal possedere una differente tipologia di elementi processanti e generalmente differenti caratteristiche nella gestione della memoria, identificate con il termine di architetture eterogenee. Le soluzioni sino qui riportate fanno tutte riferimento al caso di un'architettura omogenea, ma anche per

il caso di architetture eterogenee si è assistito ad una continua ricerca nel settore per ottenere prestazioni sempre migliori; basti pensare al sempre maggiore utilizzo di Graphical Processing Units (GPUs) come acceleratori per applicazioni anche non relative alla grafica [13].

Lo stesso andamento si è avuto nel campo dei dispositivi riconfigurabili, ovvero tutta quella famiglia di dispositivi che non hanno una funzionalità decisa al momento della fabbricazione, ma che possono essere di volta in volta configurati secondo le necessità dell'utente, eventualmente anche durante l'esecuzione. La loro dimensione ora non è più un vincolo così stringente da permettergli di realizzare solo sistemi relativamente semplici. Questi dispositivi possono ora essere utilizzati per realizzare anche sistemi molto complessi quali ad esempio architetture multi-core molto avanzate o addirittura architetture con un numero di elementi processanti che può adattarsi al carico di lavoro del sistema [14] realizzando così sistemi MP-SoC riconfigurabili a runtime. Il continuo aumento della complessità dei sistemi che possono essere realizzati con questi dispositivi tuttavia deve essere sostenuto da strumenti che semplifichino lo sforzo che viene richiesto allo sviluppatore che altrimenti risulterebbe difficilmente sostenibile. Lo stato attuale di questi strumenti supporta appieno diverse fasi di sviluppo per i dispositivi Application Specific Integrated Circuit (ASIC), ovvero dispositivi volti a realizzare una singola specifica applicazione. Tuttavia lo sviluppo di queste applicazioni su dispositivi di tipo ASIC presenta due principali problematiche. In primo luogo sebbene, come spiegato in precedenza, i transistor continuino a diminuire la loro dimensione è altresì vero che per realizzare tramite ASIC applicazioni sempre più grandi sarà necessaria sempre più area. Questo fatto rappresenta un problema in quanto, sebbene su un singolo *die* di silicio ci stiano sempre più componenti, non è detto che il *die* possa comunque accomodare tutti quelli necessari; inoltre al crescere della dimensione del *die* da utilizzare aumenta la probabilità che esso sia affetto da impurità che invalidano il dispositivo finale o ne rendono una parte inutilizzabile [15]. In secondo luogo il flusso di sviluppo per questi dispositivi non permette una fase di test costante delle funzionalità del circuito in fase di sviluppo al fine di verificarne la corrispondenza ai requisiti di progetto; questa fase di test è rimandata alla fine del ciclo di sviluppo dopo aver realizzato un prototipo del dispositivo.

In questo ambito stanno diventando sempre più importanti i dispositivi a logica programmabile come le Field Programmable Gate Array (FPGA); questi dispositivi hanno il vantaggio di garantire un ciclo di sviluppo più rapido in quanto rendono possibile un test costante dei componenti in via di sviluppo durante l'intera fase di progettazione permettendo quindi di risolvere ed individuare immediatamente eventuali incongruenze rispetto alle specifiche origi-

narie. Il problema della dimensione del *die* disponibile non viene risolto dalla tecnologia in se, ma dal fatto che questi dispositivi permettono di cambiare durante l'esecuzione la funzionalità svolta da una parte del dispositivo stesso. In questo modo è possibile dividere l'applicazione da implementare in vari task per fare in modo che solamente alcuni di essi siano configurati sul dispositivo e che nuovi task possano essere configurati quando necessario. Questa caratteristica permette inoltre di poter aggiornare l'applicazione da implementare senza passare per un altro ciclo di produzione come avverrebbe nel caso degli ASIC. Un altro vantaggio è che questi dispositivi permettono la riconfigurazione del sistema riuscendo quindi ad adattare la loro configurazione allo stato attuale in cui si trova a lavorare il sistema riuscendo quindi ad implementare, ad esempio, politiche e tecniche di tolleranza ai guasti [16]. Oltre che per attuare politiche di tolleranza ai guasti è possibile pensare di utilizzare questi dispositivi al fine di implementare sistemi che adattino il numero di elementi processanti a seconda delle richieste che il sistema si trova a processare.

Sorge quindi la problematica di come sia possibile gestire il dispositivo a runtime per riuscire ad implementare politiche di scheduling dei task che tengano conto delle problematiche della riconfigurazione al fine di ottimizzare ad esempio il tempo di esecuzione dei vari task nel sistema. Queste politiche devono tenere conto che allo stato attuale la riconfigurazione è un'operazione spesso lenta e quindi potrebbe non risultare sempre vantaggiosa. Un'altra problematica è il fatto che il modulo necessario per effettuarla è spesso unico e quindi si può effettuare una sola riconfigurazione alla volta; questo unitamente al problema precedente fa in modo che le politiche debbano riuscire a runtime a capire quale sia la strategia migliore da utilizzare scegliendo se e quando effettuare la riconfigurazione di un modulo. Lo stato dell'arte dei framework per il supporto allo sviluppo di questi dispositivi tuttavia non è ancora completo e spesso sono disponibili flussi di lavoro che supportano la riconfigurazione ma che non permettono di realizzare la gestione del sistema a runtime al fine di supportare dispositivi in grado di eseguire più task in maniera concorrente. Inoltre i principali problemi che questi framework mostrano allo stato attuale è nella mancanza nell'automazione di alcuni passi fondamentali quali la creazione dei core hardware e le loro interfacce e nell'effettuare l'integrazione con il sistema complessivo, ovvero la FPGA ed eventuali componenti ad essa collegati, come può essere nel caso di lavoro su di una scheda di sviluppo. Spesso gli strumenti disponibili al giorno d'oggi effettuano automaticamente solamente alcuni di questi passi, quelli che offrono l'integrazione dell'intero sistema non effettuano la generazione automatica dei core, mentre chi effettua la generazione automatica dei core non consente l'integrazione automatica

dell'intero sistema a livello di scheda di sviluppo, ma si concentra solamente nello sviluppare il sistema limitatamente alla sola FPGA. Vi è quindi la mancanza di uno strumento che assista il designer hardware in ogni fase dello sviluppo del sistema.

Questo lavoro ha come l'obiettivo quello di sviluppare un framework che fornisca il completo supporto allo sviluppo di un sistema riconfigurabile in grado di eseguire più applicazioni in modo concorrente. Il flusso in oggetto, così come rappresentato in Figura 1.1, parte da una descrizione delle applicazioni da implementare in un linguaggio di alto livello (in codice C), da una descrizione dell'architettura di riferimento (in formato XML) e da un insieme di direttive fornite in ingresso dallo sviluppatore configurare il flusso; l'output prodotto dal framework è il bitstream, o la serie di bitstream nel caso di sistema riconfigurabile, pronti ad essere usati sul dispositivo; tale bitstream inoltre conterrà il sistema di gestione a runtime per lo scheduling e la riconfigurazione dei vari task.

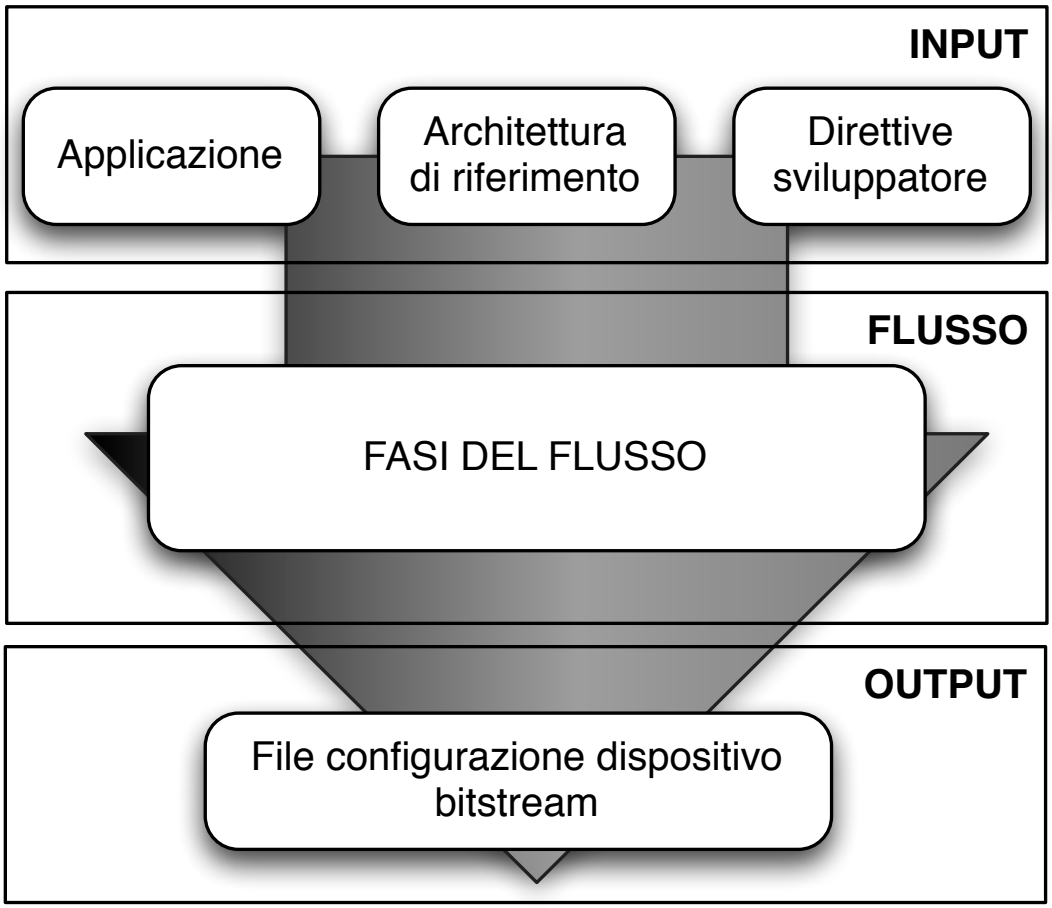

Figura 1.1: Struttura del flusso automatico.

I contributi principali di questo lavoro di tesi sono quindi:

- la definizione di un'architettura riconfigurabile che possa utilizzare i core generati tramite la fase di High Level Synthesis (HLS) automatica;
- l'implementazione di un framework automatico per sviluppare applicazioni su questa piattaforma al fine di supportare l'esecuzione concorrente delle stesse; in particolare ci si è concentrati sulla automazione della creazione delle interfacce dei core hardware, dei driver di comunicazione tra i diversi elementi del sistema e dell'integrazione dei core generati nel sistema;
- la creazione di un livello di gestione software del dispositivo volto a garantire l'esecuzione corretta delle applicazioni gestendo a runtime le dipendenze tra i vari task e decidendo come riconfigurare i core nel miglior modo possibile in relazione alle diverse condizioni di lavoro nelle quali si trova il dispositivo.

Prima di addentrarci nella descrizione del lavoro nella Sezione 1.2 verrà fornita una descrizione ad alto livello della struttura dei framework per lo sviluppo di sistemi embedded.

1.2 Il flusso di sviluppo di un MPSoC

Alcuni framework di sviluppo che assistono il progettista in ogni fase dello sviluppo sono stati proposti in letteratura e verranno illustrati in dettaglio nel Capitolo 3. In Figura 1.2 è invece rappresentato un flusso generico che risponde alla necessità di assistere il progettista nelle varie fasi del ciclo di sviluppo; nella figura sono evidenziati gli aspetti caratteristici di tale flusso quali gli input e gli output e vengono identificate le varie fasi. Un flusso di questo genere può essere automatizzato in tutto o in parte; nel primo caso il framework non ha bisogno di intervento da parte dello sviluppatore tranne che nella preparazione dei file di input; sarà poi il flusso stesso a prendersi cura di tutti i passi da compiere per adattare i prodotti intermedi agli input delle fasi successive al fine di ottenere gli output desiderati; tuttavia uno strumento di questo tipo è praticamente impossibile da realizzare in quanto lo spazio delle soluzioni da analizzare sarebbe troppo vasto e quindi alcune scelte sono in genere delegate al progettista. Verranno ora analizzati in dettaglio le varie parti di Figura 1.2 elencandone le principali caratteristiche.

Input

1. *Applicazione*: Rappresenta l'insieme di algoritmi che si vogliono portare sul dispositivo, descritti in un linguaggio di alto livello (C, C++, o altro);

2. *Architettura target*: Consiste in una descrizione ad alto livello (ad esempio XML) del modello dell'architettura da implementare; contiene informazioni come ad esempio:

   - Numero di processori SW e loro caratteristiche;
   - Numero di dispositivi riconfigurabili, e loro caratteristiche;
   - Memorie e loro specifiche;
   - Elementi di comunicazione;
   - Descrizione delle interconnessioni;

3. *Direttive dello sviluppatore*: Insieme di direttive volte ad adattare il flusso del framework alle necessità dello sviluppatore. Fanno parte di questa categoria ad esempio la scelta di come assegnare i vari task ai processori oppure altri dettagli, quali ad esempio le stime relative ai tempi di esecuzione o risposta dei vari task e componenti.

Fasi del flusso

1. *Rappresentazione interna*: l'applicazione in ingresso necessita di essere codificata in una qualche rappresentazione intermedia per poter essere manipolata dal flusso di sviluppo. La codifica presa in considerazione in questo lavoro è la rappresentazione in forma di Task Graph [17, 18] , tale specifica verrà approfondita nella Sezione 2.2;

2. *Identificazione dei task*: in questa fase si va a dividere l'applicazione di ingresso in un insieme di task; tale divisione può essere effettuata manualmente o con metodi automatici; i task individuati a questo passo sono gli elementi alla base delle fasi successive del flusso;

3. *Partizionamento Hw/Sw ed assegnamento dei task ai processori*: in questa fase si va ad identificare una partizione tra i task per decidere quali traggono vantaggio da un'implementazione hardware dedicata e quali invece risulta più opportuno eseguire su di un General Purpose Processor (GPP) e si assegnano quindi i vari task alle risorse computazionali disponibili sul dispositivo; generalmente viene anche calcolato lo schedule di come verrà

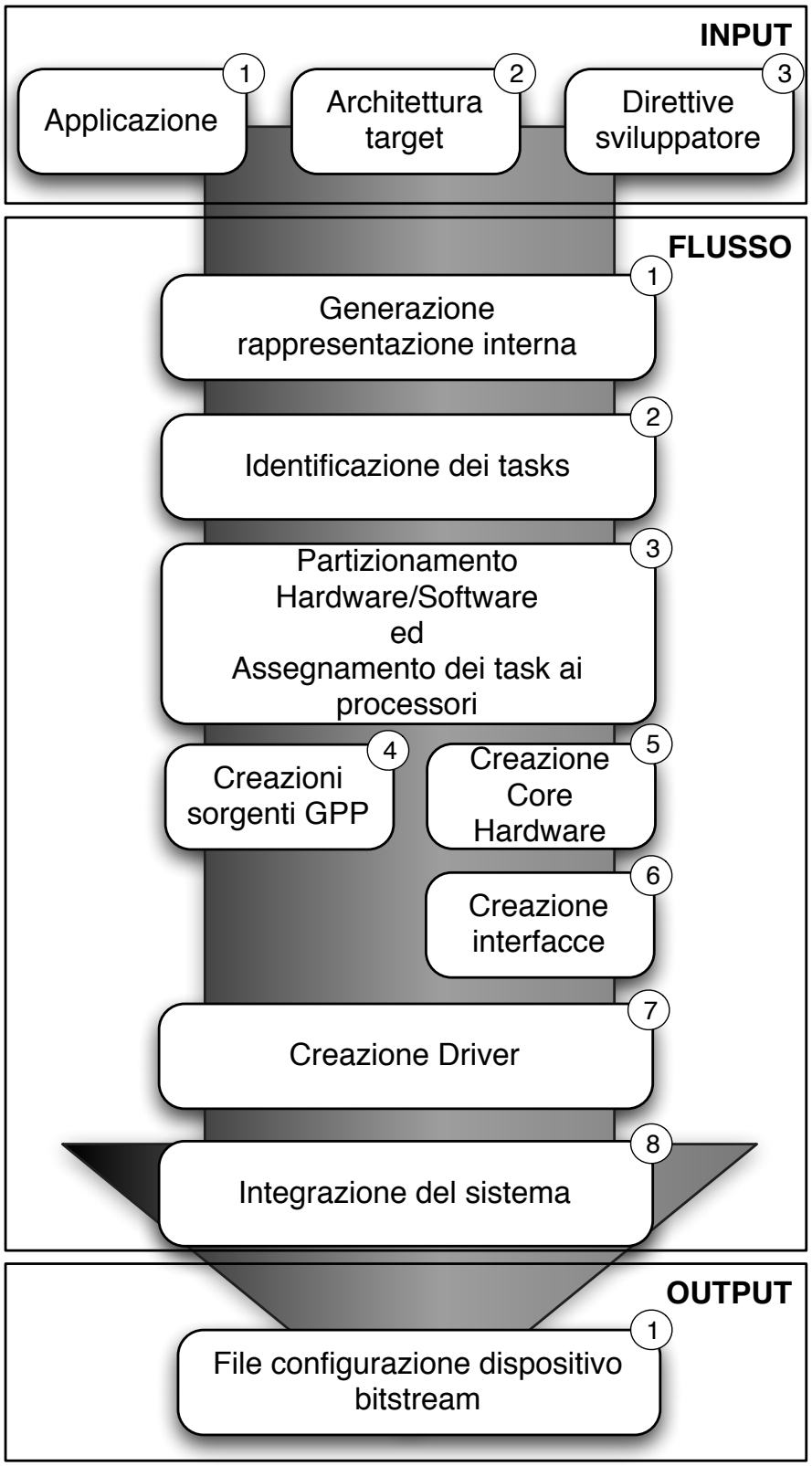

Figura 1.2: Detteglio delle fasi del flusso di sviluppo.

eseguita l'applicazione sul dispositivo provando a minimizzarne in tempo d'esecuzione; questo problema verrà descritto nel dettaglio nella Sezione 2.3;

4. *Creazione sorgenti GPP*: in questa fase vengono create i sorgenti software necessari ad eseguire i task assegnati ai GPP presenti sul dispositivo finale;

5. *Generazione dei core hardware*: questa fase si occupa della traduzione da codice di alto livello in Hardware Description Language (HDL); maggiori informazioni sono fornite nella Sezione 2.4;

6. *Generazione delle interfacce*: a questo punto vengono generate le interfacce necessari per la comunicazione tra i moduli hardware e la loro successiva integrazione nel sistema finale; vengono realizzate sia le interfacce tra i diversi moduli, sia quelle tra i moduli e le altre periferiche del sistema come ad esempio le interfacce verso la memoria;

7. *Creazione dei driver*: questa fase crea i driver necessari alla comunicazione tra il mondo hardware e quello software astraendo i protocolli di comunicazione così da poter essere efficientemente utilizzati;

8. *Integrazione del sistema*: infine questa fase genera i risultati finali del flusso ovvero i sorgenti per i GPPs presenti ed integra i componenti hardware nel sistema; in questa parte viene inoltre generata tutta quella parte necessaria per la gestione a runtime dell'applicazione, sia essa hardware o software.

Output

1. *MPSoC*: il risultato del flusso consiste nel sistema MPSoC pronto ad essere configurato sulla scheda; la generazione dei bitstream di configurazione è fatta generalmente con strumenti proprietari dello sviluppatore del dispositivo target (ad esempio quelli di Xilinx [19], oppure Altera [20]) e può essere o meno effettuata in maniera automatica.

1.3 Sommario

In questo capitolo si è introdotto il problema trattato in questo lavoro; si è illustrato come il continuo progresso tecnologico stia rapidamente modificando le strutture dei calcolatori incentivando la ricerca nel campo delle architetture al fine di riuscire a garantire performance sempre migliori nei diversi domini applicativi. Si è mostrato inoltre come gli strumenti ed i

framework che supportano lo sviluppo di queste architetture non siano in grado di assistere i design hardware nell'affrontare tutti i nuovi problemi relativi allo sviluppo delle architetture in oggetto, quali ad esempio la capacità di gestire efficacemente la riconfigurazione disponibile su dispositivi come le FPGA. Infine si è descritto il flusso di lavoro di questi strumenti così da introdurre i problemi che questi framework devono riuscire ad affrontare al fine di riuscire ad assistere con successo il progettista.

Capitolo 2

Definizioni

In questo capitolo vengono fornite le definizioni principali necessarie alla comprensione del resto del lavoro. La Sezione 2.1 presenta una descrizione dei dispositivi riconfigurabili e del loro funzionamento. La Sezione 2.2 fornisce la definizione formale del concetto di task graph e illustra come questo possa essere usato per rappresentare un'applicazione. La Sezione 2.3 introduce il problema del partizionamento Hw/Sw, dell'assegnamento dei task alle unità funzionali e del calcolo dello schedule per l'esecuzione dell'applicazione; fornisce da prima una definizione formale del problema ed illustra quindi le tecniche e le euristiche disponibili per risolverlo. La Sezione 2.4 discute il problema della High Level Synthesis (HLS); dopo aver introdotto il problema, questa sezione mostra come sia possibile convertire un'applicazione scritta in un linguaggio di alto livello in Hardware Description Language (HDL). Infine la Sezione 2.5 mostra quali sono le problematiche che si presentano a runtime per la gestione dei sistemi embedded presi in considerazione in questo lavoro; verranno approfondite le questioni relative alle strategie disponibili per effettuare l'assegnamento dei vari task ai differenti processori e per gestire la riconfigurazione.

2.1 FPGA

Le Field Programmable Gate Arrays (FPGAs) sono una famiglia di dispositivi programmabili utilizzati soprattutto in fase di progetto dei sistemi embedded; il termine programmabili significa che è l'utente finale a stabilire la configurazione del dispositivo. Il loro utilizzo in fase di sviluppo è suggerito dal fatto che consentono di apportare modifiche semplicemente riprogrammando il dispositivo, a differenza degli ASIC, che sono personalizzati per un solo uso particolare. Questo vantaggio si paga però in termini di prestazioni e di area occupata, d'altra parte però il costo per unità delle FPGA è più basso per bassi volumi di produzione e queste possono essere utilizzate in momenti diversi, con differenti configurazioni e quindi con elevata flessibilità. Il blocco logico base di una FPGA contiene una *Lookup Table (LUT)*, ovvero una memoria che in base ai valori assunti dai bit in ingresso restituisce un bit in uscita; la funzionalità logica implementata dalla LUT può essere modificata riconfigurando il dispositivo. Le FPGA sono costituite da tre ulteriori elementi:

- componenti logici, che rappresentano le funzionalità che la FPGA fornisce all'utente, possono essere componenti più o meno complessi a seconda che si voglia privilegiare le prestazioni in termini di velocità di esecuzione diminuendo le connessioni tra gli elementi o la flessibilità, cioè la possibilità di riutilizzare lo stesso componente in varie situazioni;
- blocchi di input/output, che permettono di interfacciare i contatti del dispositivo con i segnali interni;
- interconnessioni programmabili, con le quali si possono collegare i blocchi logici tra loro o con i blocchi di input/output; possono essere locali, interessano cioè solo una parte del dispositivo e quindi pochi componenti, oppure distribuite, sono più lunghe e danno un'elevata flessibilità.

Programmare una FPGA significa configurare le interconnessioni dei blocchi logici tra loro e con quelli di input/output mediante un bitstream che viene scaricato sulla scheda; tale bitstream viene generato a partire da una descrizione del dispositivo da implementare in un opportuno linguaggio di descrizione hardware, HDL, quale il VHDL o il Verilog. Esistono vari tipi di riconfigurazione possibili, in particolare si fa riferimento a riconfigurabilità di tipo totale statica o parziale dinamica. Con il termine riconfigurabilità totale statica si intende la possibilità di riconfigurare l'intero dispositivo prima del suo utilizzo al fine di modificarne al

funzionalità; il termine riconfigurabilità parziale dinamica invece si intende la possibilità di riconfigurare singole parti del dispositivo, lasciando completamente funzionanti le restanti parti che quindi possono continuare ad assolvere ai loro compiti mentre si porta avanti il processo di riconfigurazione. Sebbene questa capacità dia alle FPGA un'elevata flessibilità, tuttavia ha delle forti limitazioni relativamente al tempo necessario che serve per riconfigurare il dispositivo; un importante ramo di ricerca in questo campo è quindi focalizzato nel cercare di ottimizzare questo processo mirando a ridurre al minimo le riconfigurazioni necessarie, riutilizzando componenti già configurati quando possibile, oppure cercare di schedulare le riconfigurazioni in modo da non generare latenze nel sistema.

2.2 Task Graph

Un task graph è un grafo in cui i vari nodi rappresentano i vari task dell'applicazione; questo formalismo si presta molto bene a rappresentare un'applicazione potendo illustrare in modo molto semplice le parti tra loro indipendenti, il codice parallelo e la quantità di informazioni che vengono scambiate, nonché il flusso delle stesse. Vengono ora fornite le definizioni fondamentali relative al concetto di grafo ed altri concetti ad esso collegati per poi vedere come i vari elementi che lo compongono possono trovare una corrispondenza nel caso in cui tale grafo sia utilizzato per rappresentare un'applicazione.

Definizione 2.2.1 (Grafo) *Un grafo* G *consiste in una coppia* (V, E)*, dove* V *è un insieme finito di vertici o nodi, mentre* E *è un insieme di coppie non ordinate di vertici chiamati archi. La presenza di un elemento* $(u, v) \in E$ *implica l'esistenza di un collegamento tra i nodi* u *e* v*.*

Definizione 2.2.2 (Grafo diretto) *Un grafo* G *si dice diretto quando gli elementi di* E *sono un insieme di coppie ordinate di vertici. Un elemento* $(u, v) \in E$ *in un grafo diretto implica l'esistenza di un collegamento tra i nodi* u *e* v *tramite un arco uscente da* u *ed entrante in* v*; nella notazione grafica si indica la direzione dell'arco mediante il verso della freccia associata.*

Definizione 2.2.3 (Cammino) *Un cammino* p *di lunghezza* k *tra due nodi* u *e* u' *di un grafo G è una sequenza* $p = \langle v_0, v_1, ..., v_k \rangle$ *tale che* $v_0 = u$*,* $v_1 = u'$ *e* $(v_{i-1}, v_i) \in E$ *per* $i = 1, 2, ..., k$..

Definizione 2.2.4 (Raggiungibilità) *Un nodo* u' *si dice raggiungibile da un nodo* u *se esiste un cammino* p *che collega* u *a* u'*.*

Definizione 2.2.5 (Ciclo) *Si definisce ciclo un cammino* $p = \langle v_0, v_1, ..., v_k \rangle$ *dove* $v_0 = v_k$ *e* $v_1, v_2, ..., v_{k-1}$ *sono tra loro distinti.*

Definizione 2.2.6 (Grafo aciclico) *Un grafo* G *si dice aciclico se non contiene nessun ciclo al suo interno.*

Quelle fornite sono le definizioni basilari riguardanti la teoria dei grafi e di rilevanza per il resto del lavoro; per una trattazione più approfondita si rimanda a [21]. Di fondamentale importanza per la rappresentazione delle applicazioni sono i grafi diretti aciclici, Directed Acyclic Graphs (DAGs). Alcuni esempi di grafi sono riportati in Figura 2.1. Si parla di Task

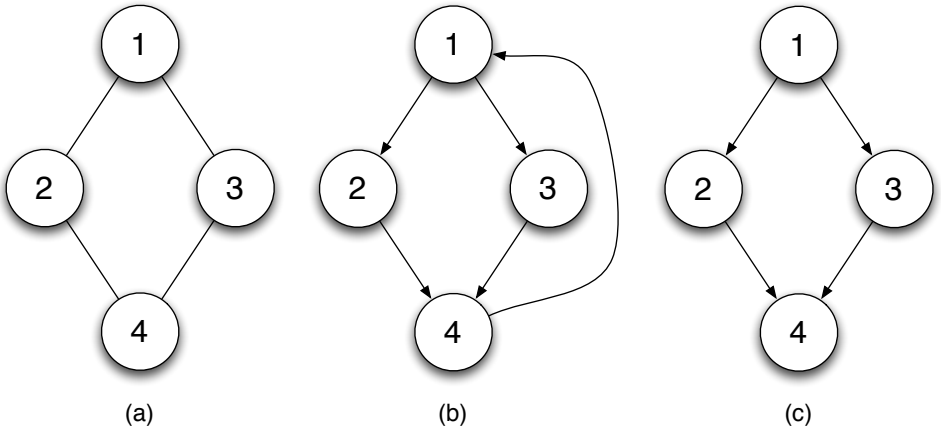

Figura 2.1: Esempio di grafi. (a) rappresenta un grafo elementare; (b) raffigura un grafo diretto con due cicli 1-2-4-1 e 1-3-4-1; (c) infine rappresenta un DAG.

Graph, nella forma di DAG, quando i nodi del grafo rappresentano un insieme delle istruzioni dell'applicazione in analisi, ognuno di questi nodi rappresenta un task dell'applicazione. Gli archi in questo caso rappresentano le dipendenze tra i vari task ed a questi è possibile associare un peso che identifica ad esempio la mole di dati trasferiti tra due task. Data la natura sequenziale del codice di un'applicazione risulta evidente che il DAG si presta ad una sua rappresentazione; in particolare su questa struttura dati è possibile fare tutta una serie di analisi al fine di identificare le dipendenze tra i vari task oppure trovare quelli tra loro in parallelo. Esistono una serie di tecniche, fondamentali nel campo dei compilatori, per effettuare queste analisi; tali tecniche non verranno discusse in questo lavoro, ma qualora si volesse approfondire l'argomento si può far riferimento a [22]. In Figura 2.2 è riportato un esempio di DAG che rappresenta il task graph relativo al Listato 2.1.

Come si può vedere dall'esempio è facile identificare a partire dal DAG quali sono le parti dell'applicazione che possono essere eseguire in parallelo; per essere più precisi i task che

Listato 2.1 Codice di esempio per DAG

```
1 int a=5;
2 int b=17;
3 foo1(&a);
4 foo2(&b);
5 printf("%d",a+b);
```

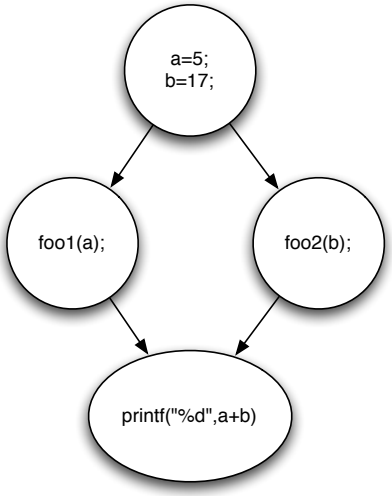

Figura 2.2: DAG corrispondente al codice riportato nel Listato 2.1.

posso essere eseguiti in parallelo sono tutti e soli i task tra loro non raggiungibili. Tuttavia al fine di rappresentare nella sua completezza un'applicazione la natura del DAG pone un vincolo molto stringente in quanto non permette di rappresentare cicli. Per superare questo problema viene quindi introdotto un nuovo tipo di rappresentazione, ovvero il task graph gerarchico, Hierarchical Task Graph (HTG) [17, 18].

Definizione 2.2.7 (Task Graph gerarchico) *Un task graph gerarchico, o HTG, è un task graph in forma di DAG dove ad ogni nodo è possibile associare un sottografo in forma di task graph gerarchico.*

Sfruttando questa struttura dati è possibile rappresentare anche applicazioni con cicli semplicemente rappresentando il corpo del ciclo come sottografo del nodo dove risiede la condizione; un esempio di questa rappresentazione è riportato in Figura 2.3 relativamente al codice fornito nel Listato 2.2.

Listato 2.2 Codice di esempio per HTG.

```
1 int a=5;
2 int i=0;
3 for(i=0;i<10;i++){
4         a++;
5 }
6 printf("%d",a);
```

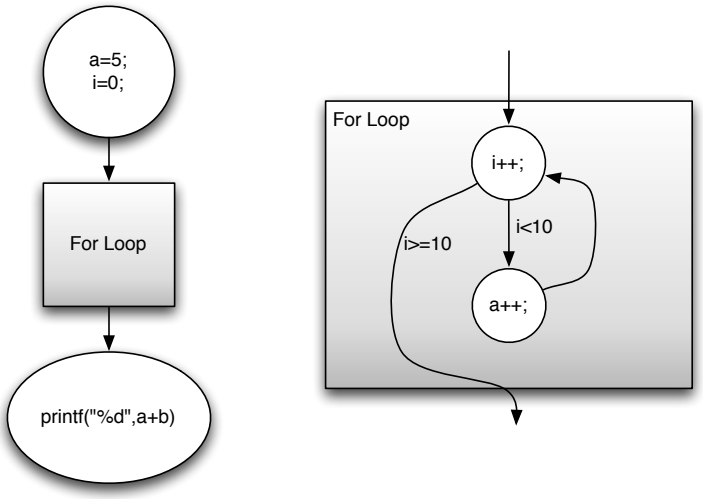

Figura 2.3: HTG corrispondente al codice riportato nel Listato 2.2. A sinistra il primo livello della gerarchia; a destra il dettaglio del sottografo del task *For Loop*.

2.3 Partizionamento Hw/Sw ed assegnamento dei task alle risorse

Come anticipato nella Sezione 1.1 in questa fase del flusso si vuole identificare quali tra i vari task dell'applicazione quali convenga eseguire in software e quali in hardware; sempre in questa fase i vari task vengono mappati alle varie risorse disponibili nel sistema e viene poi calcolato lo schedule per l'esecuzione dell'applicazione. Al fine di identificare questo assegnamento è necessario avere a disposizione i dati relativi ai tempi di esecuzione dei vari task, il tempo di trasferimento delle comunicazioni ed eventualmente il tempo necessario alla riconfigurazione del task qualora esso necessiti di essere riconfigurato; in assenza di questi dati è necessario fornirne una stima attraverso metodi e tecniche di stima appositi come ad esempio proposto in [23]. Alcune definizioni formali di questo problema si possono trovare ad esempio in [24, 25, 26]; verrà ora qui riportata quella fornita da [26].

Definizione 2.3.1 *Partizionamento Hardware/Software ed assegnamento dei task alle risorse.*

Siano $G = (V, E)$ *un grafo diretto diretto aciclico associato ad un'applicazione ed* $A = P \cup C$ *un modello dell'architettura target nella quale sono disponibili, per ogni componente,* Q *risorse suddivise tra rinnovabili* R *e non rinnovabili* N *con* $Q = R \cup N \wedge R \cap N = \varnothing$ *(*P *è l'insieme delle unità funzionali, mentre* C *è l'insieme degli elementi di comunicazione). Un job* j *come un'attività che deve essere eseguita su di un componente di un'architettura. Ogni task* t *sarà quindi rappresentato da un singolo job. Un'implementazione* i *è definita come una particolare combinazione di risorse e tempo richiesto per l'esecuzione di un job* j *su di un componente* a_k *dell'architettura. Ogni job può avere diverse implementazioni e non necessariamente su tutti i componenti dell'architettura.*

Si definiscono inoltre le tre seguenti funzioni:

$$\gamma : I \to A$$

$$\delta : I \to \mathbb{N}$$

$$\sigma : I \times Q \to N$$

dove $\gamma(i)$ *associa ogni implementazione* i *il componente* $a \in A$ *a cui l'implementazione è riferita;* $\delta(i)$ *denota invece il tempo di esecuzione necessario per eseguire l'implementazione* i *sul componente ad essa associato; mentre* $\sigma(i, q)$ *associa ad ogni implementazione* i *e risorsa* q *la quantità di questa risorsa richiesta per portare a termine l'implementazione stessa.*

Definiamo ora la funzione M *di assegnamento dei job alle implementazioni e la funzione* S *dello scheduling dei job rispettivamente come:*

$$M : J \to I$$

$$S : J \to \mathbb{N}$$

dove $M(j)$ *associa ogni job* j *con l'implementazione* i *scelta e* $S(j)$ *identifica il suo tempo di inizio. Il problema di trovare l'assegnamento ottimo consiste nel trovare quell'assegnamento che minimizza o massimizza una certa metrica di importanza per il problema in questione. In questo lavoro si fa riferimento al caso di minimizzazione del tempo di esecuzione di una funzione che quindi coincide con la lunghezza dello schedule generato. Se definiamo* H_j *come l'istante di tempo in cui un job* j *termina l'esecuzione. ovvero* $H_j = S(j) + \delta(j)$*; la lunghezza totale dello schedule sarà quindi in questo caso definibile come:*

$$Z = max(H_j) \;\; \forall j \in J$$

Per minimizzare Z *nel caso ideale ogni job j deve essere assegnato all'implementazione più rapida tra quelle possibili; tuttavia l'assegnamento deve soddisfare alcuni vincoli. In particolare affinché*

l'assegnamento trovato è valido se e solo se:

$$\sum_{j \in J : a_k = \gamma(M(j))} \sigma(M(j), q) <= A_k^q \;\; \forall a_k \in A, q \in N$$

questo significa che la richiesta delle risorse non rinnovabili $N \in Q$ *non deve superare la disponibilità delle stesse per ogni componente. Inoltre lo schedule trovato si dice valido se e solo se ogni job inizia la propria esecuzione dopo che i suoi predecessori sono terminati e se il componente che lo deve eseguire è libero. In particolare quindi deve sempre valere questo vincolo:*

$$max[S(j\prime) + \delta(M(j\prime)), avail(a_k)] \leqslant S(j) \;\; \forall j\prime \in prec(j) : a_k = \gamma(M(j))$$

dove $prec(j)$ *indica l'insieme dei predecessori del job* j *e* $avail(a_k)$ *indica il momento in cui il componente* a_k *è disponibile.*

Un esempio del problema appena descritto è fornito in Figura 2.4, in alto a sinistra è rappresentato il task graph iniziale, in basso il task graph che denota gli assegnamenti scelti, mentre a destra è riportato lo schedule finale. Il partizionamento raffigurato è il migliore possibile al fine di minimizzare la lunghezza dello schedule. Nella Tabella 2.1 sono riportati i dati relativi ad alcuni possibili assegnamenti e le relative lunghezze dello schedule. Si noti come il task T2 benché abbia un'implementazione più rapida sul componente hardware non venga assegnato a questo in quanto tale soluzione non porterebbe al minimo schedule possibile.

Tabella 2.1: Possibili partizionamenti per il task graph di Figura 2.4. L'ultimo è quello rappresentato nella figura.

Tasks Hw	Tasks Sw	Tempo d'esecuzione
T1,T2,T3,T4		22
	T1,T2,T3,T4	27
T1,T2	T3,T4	26
T3,T4	T1,T2	**17**

Il problema di trovare una partizione dati i tempi di esecuzione dei vari task è però un problema NP-completo [27] e quindi una sua soluzione tramite modelli matematici risulta infattibile se non per grafi molto semplici; un esempio di formulazione come problma di Integer Linear Programming (ILP) è fornita sempre in [27]. Per risolvere il problema del partizionamento si è quindi fatto ricorso a diversi euristiche; lo stato dell'arte presenta lavori che fanno ricorso a strategie quali varianti dell'algoritmo A^* [25], l'ant colony optimization [26, 28, 29, 30], gli algoritmi genetici [31, 32], soluzioni di tipo greedy [33], l'euristica

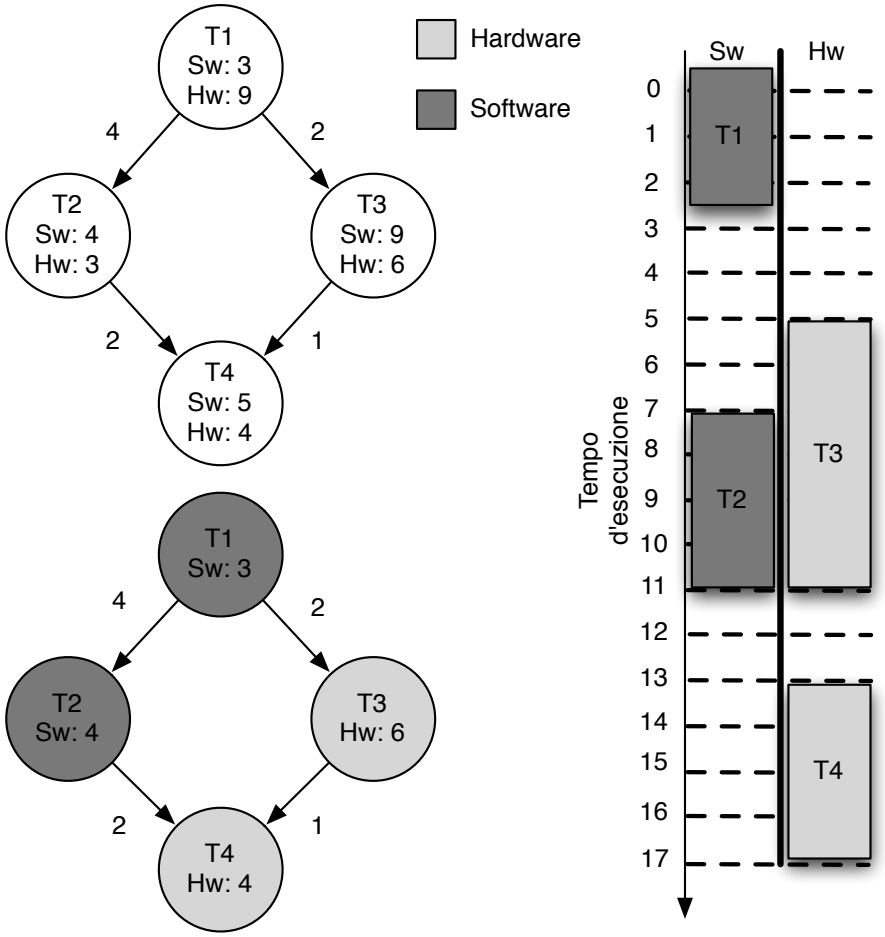

Figura 2.4: Esempio di partizionamento Hw/Sw. A sinistra il task graph originale; a destra quello partizionato

di Kernighan/Lin (KL) [34, 35, 36] adattata al caso del partizionamento, oppure euristiche proposte per questo problema specifico [27, 24].

Un'altra soluzione possibile è infine quella di lasciare identificare allo sviluppatore il partizionamento che secondo lui risulta essere esatto, permettendogli così di poter effettuare una fase di analisi dello spazio di ricerca manualmente. Questa soluzione è quella seguita in questo lavoro, e la giustificazione di questo risiede nel fatto che tale soluzione permette un controllo molto più diretto sulla strategia di analisi dello spazio delle soluzioni; inoltre gli strumenti commerciali attualmente disponibili (ad esempio quelli di Xilinx), si avvalgono anch'essi di questa soluzione. Infine, dato che questo lavoro ha come obiettivo quello di supportare l'esecuzione di più applicazioni in modo concorrente ed in funzione delle condizioni di lavoro del dispositivo, ogni analisi in fase di design del dispositivo per trovare lo schedule migliore identificherebbe uno schedule statico che quindi non garantirebbe la migliore esecuzione in caso di sistema sottoposto a condizioni di lavoro variabili.

2.4 Sintesi di alto livello

Il problema del HLS consiste nel trasformare un'applicazione, o una qualche sua parte, descritta in un linguaggio di alto livello nel suo equivalente in HDL in modo automatizzato. Questo problema è di fondamentale importanza per lo sviluppo dei sistemi embedded in quanto la realizzazione di uno strumento che riesca a risolvere il problema ottenendo una soluzione di buona qualità permetterebbe di velocizzare enormemente il processo di sviluppo di questi dispositivi. Data la sua importanza questo problema è oggetto di molti lavori di ricerca e lo stato attuale degli strumenti sviluppati consente di realizzare una traduzione da linguaggi come *C* a *VHLD* o *Verilog* di buona qualità e con performance finali più che buone; tuttavia nel caso di design che necessitino di alte performance o comunque con vincoli stringenti l'intervento umano è ad oggi necessario.

Gli strumenti per effettuare HLS partono generalmente da due input: il codice dell'applicazione da tradurre ed una libreria di componenti hardware; la libreria contiene dei componenti associati ognuno a delle operazioni del linguaggio sorgente o a delle intere funzioni.

La strategia di soluzione del problema consiste generalmente nei seguenti passi:

- analisi del codice di alto livello al fine di estrarne un grafo delle operazioni eseguite;
- algoritmi di copertura del grafo al fine di mappare ogni nodo ad un componente hardware della libreria;
- definizione ed allocazione dei registri e della memoria del core hardware da generare;
- creazione della macchina a stati, o Finite State Machine (FSM), per la gestione del core stesso

Ad oggi sono presenti in letteratura alcuni strumenti in grado di affrontare questo problema; alcuni di questi sono integrati in flussi di sviluppo per sistemi embedded e le principali soluzioni verranno illustrate nel Capitolo 3. La differenza tra gli strumenti che effettuano HLS sta in primo luogo nella qualità del codice HDL prodotto, ma soprattutto nei limiti che lo strumento di traduzione impone allo sviluppatore nello scrivere il codice di alto livello da tradurre.

2.5 Gestione del sistema a runtime

Le tecniche e le problematiche mostrate finora fanno riferimento al caso in cui il sistema da generare sia completamente analizzabile a design time e quindi in cui questo non sia soggetto a variazioni dovute alle condizioni di lavoro quali ad esempio diverse condizioni di carico. Questa problematica sta guadagnando sempre più rilevanza; un sistema di questo tipo ha la necessità di adattare la propria struttura alle condizioni di lavoro variabili a runtime. Per far questo il dispositivo ha l'obbligo di sfruttare la riconfigurazione nel modo più intelligente possibile; alcune soluzioni proposte in letteratura, come ad esempio [37], fanno uso di queste tecniche per ottimizzare la riconfigurazione:

- *mascheramento del tempo di riconfigurazione*: consiste nel cercare di schedulare la riconfigurazione di un componente mentre il sistema sta effettuando delle operazioni al fine di non introdurre ritardo
- *prefetch delle riconfigurazioni*: ovvero cercare di anticipare la riconfigurazione dei task che devono essere eseguiti anche se essi non sono ancora pronti
- *riuso dei componenti*: consiste nell'evitare riconfigurazioni inutili cercando di riutilizzare i componenti configurati il più possibile

Le soluzioni attualmente disponibili in questo ambito sono ancora poche; alcuni lavori propongono algoritmi e gestori a runtime, sia hardware che software, per gestire le riconfigurazioni di acceleratori [37], mentre altri mirano invece a realizzare architetture in grado di riconfigurare il numero di processori disponibili a runtime al fine di adattarli alle condizioni di lavoro [38, 14].

2.6 Sommario

In questo capitolo sono state introdotte tutte le definizioni necessarie al fine di riuscire a capire al meglio quanto sviluppato in questo lavoro. Dapprima sono state introdotte le FPGA che sono i dispositivi verso i quali è orientato il flusso di sviluppo per Multi Processors System on Chip (MPSoC) sviluppato in questa tesi; quindi è stato definito il concetto di task graph che è la struttura dati generalmente utilizzata per rappresentare un'applicazione all'interno dei framework per lo sviluppo di sistemi embedded ed, in particolare, è quella utilizzata in questo lavoro; sono stati infine presentati i problemi principali che devono essere affrontati

nella fase di sviluppo del sistema dandone una definizione formale, fornendo vari esempi al fine di illustrarne al meglio le proprietà ed illustrando come questi problemi siano affrontati nello stato dell'arte.

Capitolo 3

Stato dell'arte

In questo capitolo verranno analizzati i framework attualmente disponibili in letteratura. Per ognuno di essi verrà descritto il flusso di lavoro mettendo in evidenza le soluzioni adottate per risolvere i problemi evidenziati nei capitoli 1 e 2. Nella Sezione 3.1 verrà descritto il framework PandA che è quello utilizzato per sviluppare questo lavoro; nella Sezione 3.2 verrà spiegato il funzionamento di Daedalus; nella Sezione 3.3 viene data un'approfondita descrizione del framework RAMPSoC che ad oggi è l'unico in grado di supportare un sistema che a runtime si adatti alle condizioni di lavoro; infine nella Sezione 3.4 verrà presentato XPilot che tra i flussi presentati è l'unico a carattere commerciale, essendo stato acquisito da Xilinx, ed oggi alla base del prodotto AutoESL [39].

3.1 PandA

PandA è uno strumento sviluppato al Politecnico di Milano per lo sviluppo di sistemi embedded ed in particolare per fornire un supporto allo sviluppatore in tutte le fasi dello sviluppo del sistema. Il framework si divide in diversi componenti ognuno dei quali si occupa di assistere ed automatizzare il lavoro dello sviluppatore in particolare si farà qui riferimento ai seguenti strumenti:

- Zebu: è lo strumento che realizza la parte di Hw/Sw Co-design
- Bambu: è lo strumento che si occupa della parte di High Level Synthesis (HLS)

3.1.1 Zebu

Zebu è la parte del framework che, come detto, si occupa della fase di hardware software co-design ed in particolare si occupa di decidere, dato il modello di un'architettura, a quali unità funzionali assegnare i task dell'applicazione. Zebu ha come input tre elementi:

1. una specifica C dell'applicazione da implementare opportunamente annotata con pragma OpenMP [40] per evidenziarne i task ed il loro parallelismo
2. la specifica dell'architettura target descritta in XML che elenca quali sono:
    - i processori presenti
    - le memorie disponibili
    - i collegamenti tra i vari componenti
3. una serie di stime di esecuzione dei vari task sui processori

A partire da questi input il framework effettua i seguenti passi, rappresentati in Figura 3.1:

- all'inizio avviene la traduzione del codice dell'applicazione in una rappresentazione interna sotto forma di vari tipi di grafi (*task graph, Control Flow Graph (CFG), Data Flow Graph (DFG)*) al fine di poter effettuare un'analisi completa del flusso dell'applicazione e delle sue dipendenze dati; per effettuare questo passo ci si appoggia al compilatore GNU Compiler Collection (GCC)
- una volta ottenuta questa rappresentazione viene attuata una fase di assegnamento dei task ai processori al fine di ottimizzare il tempo di esecuzione dell'applicazione stessa;

questo passo può essere effettuato con diverse euristiche, una delle versioni più efficienti implementate sfrutta la tecnica dell'*ant colony optimization*[28, 26, 29]

- al termine di questo passo ogni task viene assegnato ad un processore specifico in modo che lo schedule risultante sia valido. Il risultato di questo passo viene fornito in output sotto forma di XML ed inoltre vengono generati i sorgenti C annotati anch'essi con OpenMP pronti per essere eseguiti sui processori dell'architettura
- per i core che vengono mappati per l'esecuzione su hardware dedicato è necessario tradurli in linguaggio Hardware Description Language (HDL) con *Bambu*

Uno dei limiti ad oggi presenti in Zebu è la sua mancata integrazione con Bambu al fine di generare automaticamente i core hardware necessari e la mancanza del supporto per la riconfigurazione dinamica dei core. Infine ad oggi manca la parte di gestione del sistema a runtime non creando il software uno scheduler in grado di eseguire il sistema su dispositivo finale lasciando questo compito allo sviluppatore.

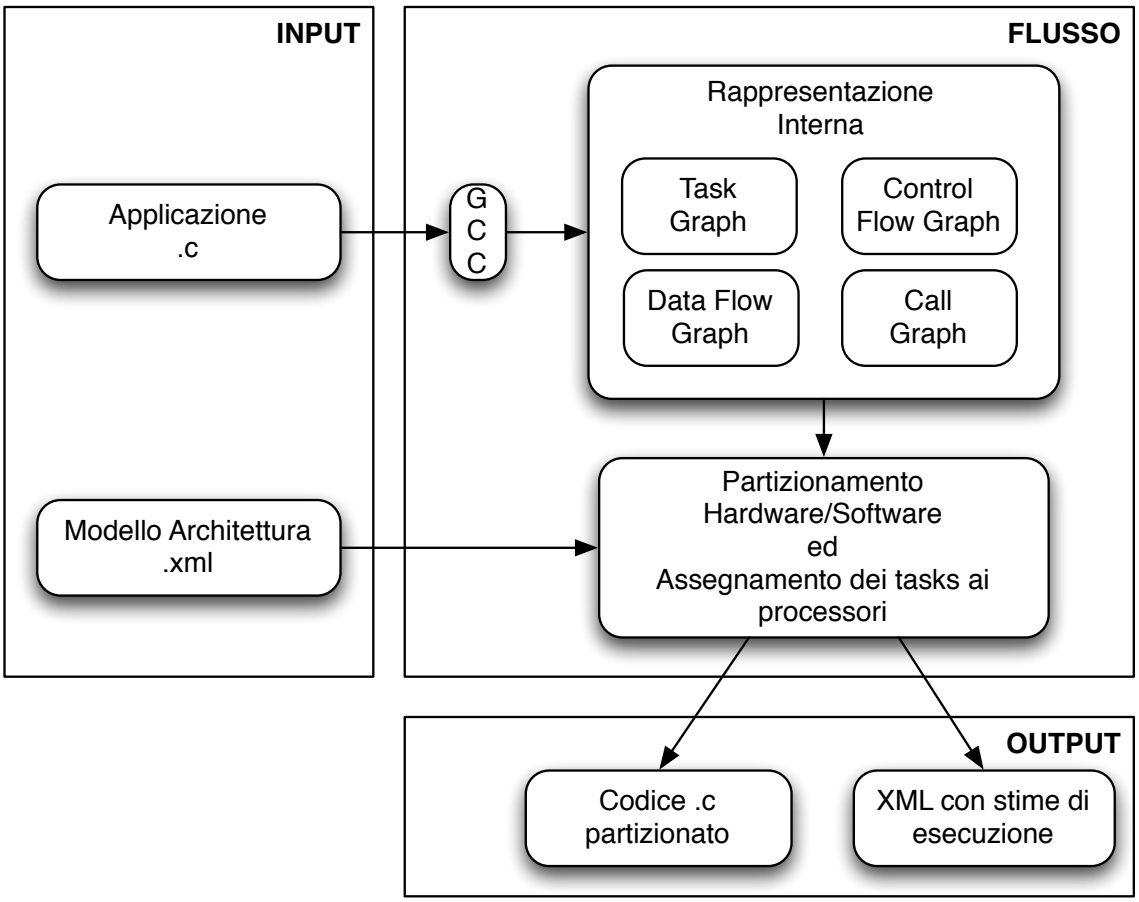

Figura 3.1: Flusso di lavoro di Zebu.

3.1.2 Bambu

Bambu come detto è la parte del progetto che si occupa di risolvere il problema della HLS. Esso necessita come input della descrizione dell'applicazione in C; il flusso segue la descrizione del generico flusso di lavoro per la risoluzione del problema di HLS descritto nel Capitolo 2 e si basa su di una libreria di componenti codificata in formato XML liberamente estendibile.

Le fasi principali del flusso sono le seguenti e sono rappresentate in Figura 3.2:

- traduzione del codice sorgente in forma di grafo (usa la stessa rappresentazione di Zebu)
- assegnamento ad ogni operazione e funzione del codice di un componente di libreria in grado di implementarla cercando di ottimizzare una certa metrica (ad esempio riduzione del tempo di esecuzione o dell'energia consumata)
- generazione del codice HDL relativo ai componenti scelti
- creazione del *datapath* per i core
- generazione della Finite State Machine (FSM) per gestire l'esecuzione del core

Una delle caratteristiche principali del framework è la capacità di generare core con un'interfaccia del tutto simile ad una funzione C, che quindi sono facili da integrare nel sistema finale; tuttavia questo passo di integrazione è ad oggi mancante principalmente in quanto lo strumento non genera le interfacce di cui il core necessita per integrarsi con il sistema intero e gli eventuali driver che sono necessari per comunicare con esso.

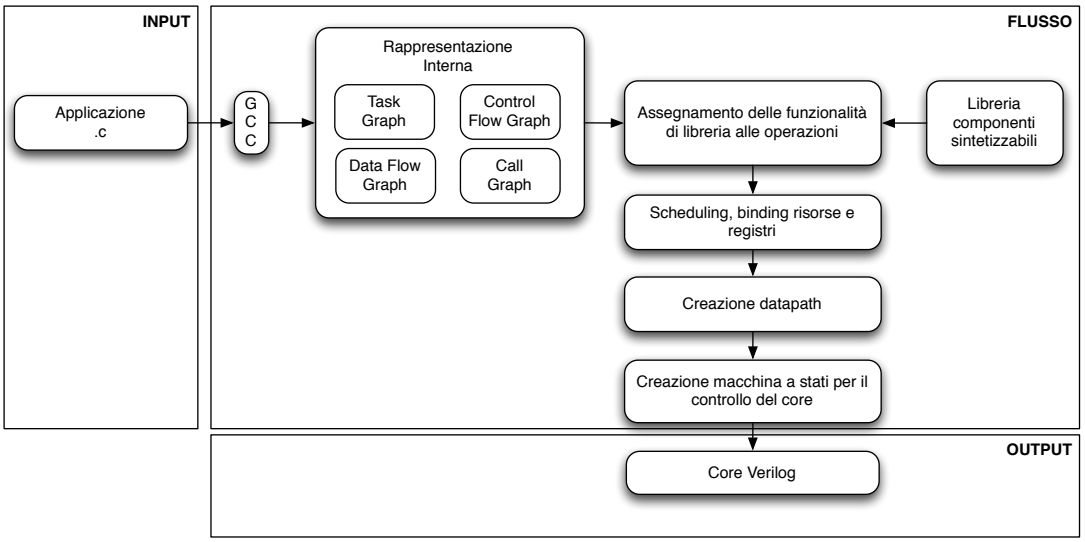

Figura 3.2: Flusso di lavoro di Bambu.

3.2 Daedalus

Daedalus [41] è un framework per lo sviluppo di MPSoC sviluppato presso il Leiden Embedded Research Center dell'università di Leiden in Belgio. Lo scopo del framework è quello di automatizzare lo sviluppo di architetture MPSoC effettuando in automatico i passi di partizionamento Hw/Sw ed assegnamento dei task ai processori al fine di fornire uno strumento in grado di fornire un supporto fondamentale allo sviluppatore nella fase di analisi dello spazio delle soluzioni. Il flusso di lavoro è composto dall'interazione di tre strumenti principali: KPNgen [42], Sesame [43, 44] e ESPAM [45, 46]. Il flusso, rappresentato in Figura 3.3, parte dalla specifica in C dell'applicazione da implementare nel sistema embedded ed affronta in sequenza questi passi:

- KPNgen: effettua la trasformazione del programma sequenziale in ingresso nella forma di Kahn Process Network (KPN) [47];
- l'applicazione espressa secondo questo formalismo viene poi analizzata da Sesame che si occupa di effettuare la fase di ricerca dello spazio delle soluzioni al fine di identificare un insieme di architetture candidate all'esecuzione dell'algoritmo in analisi; questo strumento si occupa di supportare il designer nella fase di partizionamento Hw/Sw e di assegnamento delle varie parti dell'applicazione ai vari processori;
- infine ESPAM si occupa della generazione dei core in HDL (VHDL) e del codice C per le parti che vanno eseguite sui General Purpose Processors (GPPs) inclusi nell'architettura; i core generati possono quindi essere integrati e i relativi bitstream generati con strumenti proprietari quali ad esempio Xilinx Platform Studio (XPS) prodotto da Xilinx.

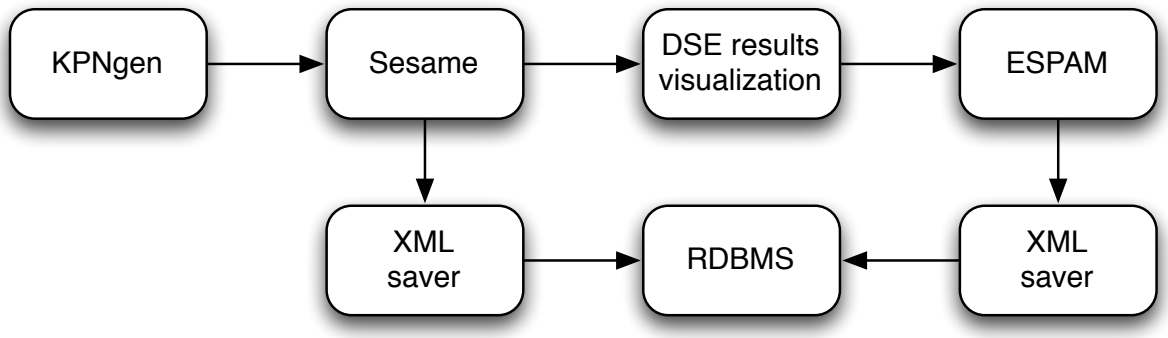

Figura 3.3: Daedalus: flusso di lavoro[41].

La Figura 3.4 mette in risalto gli input e gli output delle singole fasi mostrando come l'intero flusso si appoggi sul formato XML al fine di aumentare la compatibilità e l'interoperabilità tra le varie parti che lo compongono. Il framework gestisce inoltre per mezzo di un database una libreria di IP-core usata durante la fase di generazione del codice VHDL. Altri lavori pro-

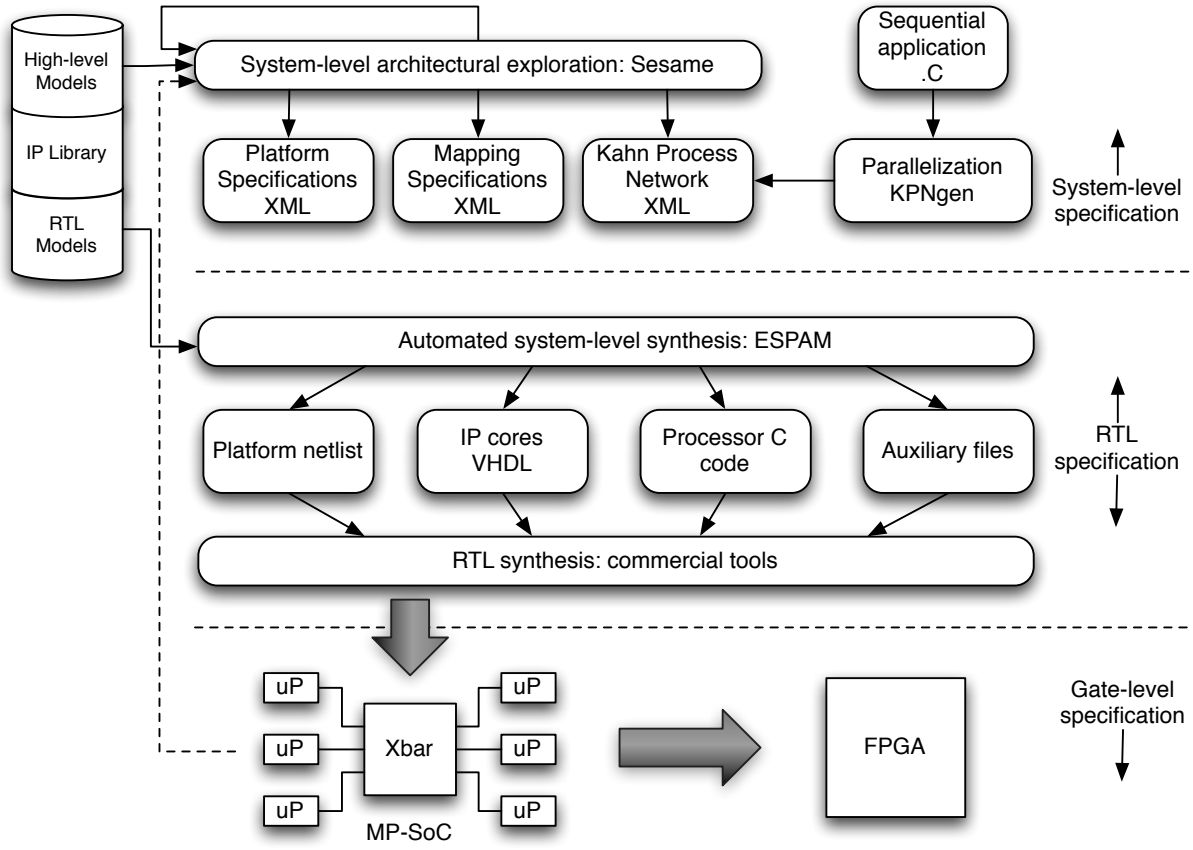

Figura 3.4: Daedalus: dettagli implementativi[41].

posti dallo stesso gruppo di lavoro si concentrano sulle singole parti del flusso presentato. In particolare in [48] gli autori definiscono un algoritmo in grado di ottimizzare il tempo di esecuzione di un task graph migliorandone la fase di partizionamento Hw/Sw effettuando un *merge* o *split* dei task iniziali al fine di ottimizzarne il bilanciamento rispetto all'architettura target. In [49, 50, 51, 52] gli autori forniscono analisi e metodi volti a ricavare un modello in forma di *Polyhedric Process Networks*[42] a partire da un'applicazione descritta in un linguaggio di alto livello fornendo anche un algoritmo per stimare le performance di eventuali trasformazioni di unione sui nodi di una *Polyhedric Process Networks*. Tale algoritmo risulta di fondamentale importanza per la parte di partizionamento ed assegnamento in quanto permette di poter generare e valutare un maggior numero di soluzioni rispetto a quelle analizzabili nel caso in cui

questo passo fosse lasciato al progettista. Altri lavori infine si concentrano sulla ottimizzazione dei loop per la loro sintesi su hardware riconfigurabili [53] e sulla definizione di uno scheduler hard-realtime per il sistema da generare [54].

É importante notare che tutte le analisi effettuate da questo framework sono possibili in quanto il sistema finale ha il compito di eseguire solamente l'applicazione attualmente considerata; sotto questa ipotesi il sistema è in grado di generare un *MPSoC* effettivamente configurabile su dispositivo Field Programmable Gate Array (FPGA); tuttavia queste considerazioni non sono valide nell'ambito discusso in questo lavoro in quanto l'obiettivo è sì quello di supportare il flusso di sviluppo per *MPSoC* classici, ma il punto principale consiste nel generare uno strumento in grado di assistere il progettista nel creare un sistema *MPSoC* riconfigurabile in grado di eseguire autonomamente un numero prefissato di applicazioni.

3.3 RAMPSoC

RAMPSoC [55], acronimo di *Reconfigurable Architecture Multi Processor System on Chip*, è un framework per lo sviluppo di sistemi basati su architetture riconfigurabili e nello specifico FPGA sviluppato in Germania presso il Karlsruher Instisute of Technologie. Ad oggi è l'unico framework completo, che parte quindi da una decrizione ad alto livello dell'applicazione (C/C++) ed arriva fino al bitstream pronto ad essere configurato su scheda, in grado di supportare sistemi in grado di adattarsi a runtime al carico di lavoro. Verranno ora analizzati in dettaglio da prima il framework a livello generale nella Sezione 3.3.1; nella Sezione 3.3.2 verranno forniti dettagli sull'architettura di riferimento utilizzata da questo framework ed infine nella Sezione 3.3.3 verrà descritto il layer di gestione software in grado di fornire al sistema la possibilità di adattarsi alle condizioni di lavoro a runtime.

3.3.1 Il framework

Il flusso di lavoro del framework è rappresentato in Figura 3.5 ed è suddiviso in tre fasi:

- Fase 1:
  - riceve in ingresso un'applicazione sequenziale descritta in C o C++ (1);
  - le varie funzioni vengono profilate e viene analizzato il loro *overhead* di comunicazione (2-3);

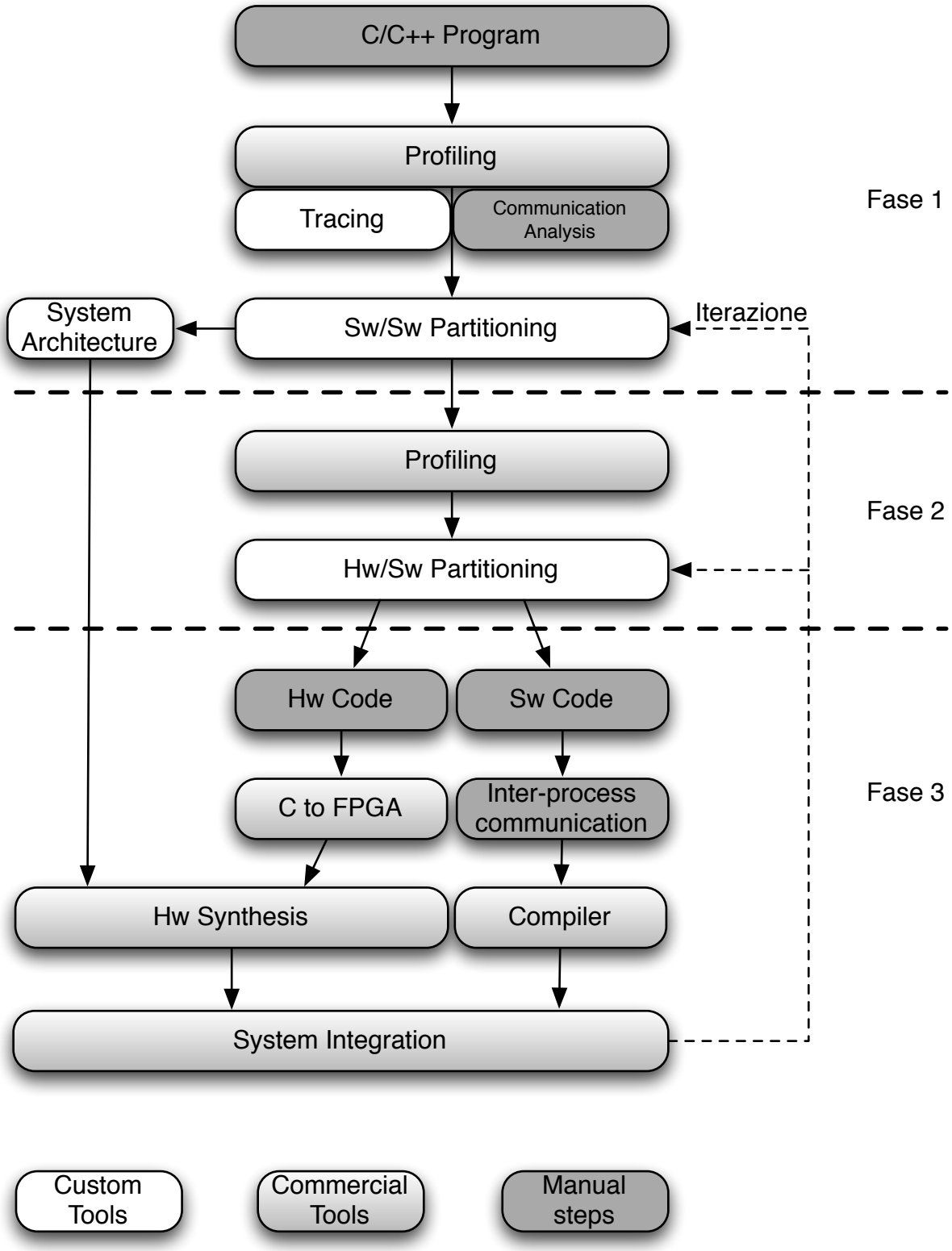

Figura 3.5: RAMPSoC: flusso di lavoro[55].

  - viene effettuata una fase di *tracing* con un set di input standard e viene estratto il call graph [56] dell'applicazione (4);
  - il risultato di queste fasi è una serie di parametri che vengono usati per eseguire un algoritmo di clustering volto ad effettuare un partizionamento Software/Software (5);
  - in uscita a questa fase vengono definiti l'architettura e un insieme di moduli applicativi, uno per ogni processore (6);

- Fase 2:
  - ogni modulo applicativo viene profilato al fine di identificare cicli o blocchi di codice computazionalmente intensivi; per questi viene suggerito di effettuare un'implementazione hardware del task, ma la sua generazione viene lasciata al progettista (7-8);

- Fase 3:
  - in questa fase vengono generati gli eseguibili software dei moduli applicativi identificati nelle fasi 1 e 2 (9-10-11);
  - per i moduli implementati in hardware viene usato uno strumento di HLS al fine di ottenere una traduzione in HDL (12-13-14);
  - vengono generati infine i bitstream completi e quelli parziali, quest'ultimi usando uno strumento appositamente sviluppato al fine di semplificare il lavoro del progettista [57] (15).

3.3.2 Architettura di riferimento

L'architettura di riferimento considerata in questo lavoro si basa su una combinazione di processori 32-bit (PowerPC [58] e Xilinx MicroBlaze [59]) e 8-bit (Xilinx PicoBlaze [60]), acceleratori hardware e diverse infrastrutture di comunicazione tra cui sistemi point-to-point (*Xilinx Fast Simplex Link* (FSL)), basati su bus (PLB,OPB,XPS,AMBA) oppure Network on Chip (NoC). In particolare per quanto riguarda la componente di comunicazione, nell'ambito di questo lavoro, è stata sviluppata un'infrastruttura, basata su NoC chiamata *Star-Wheel Network-on-Chip* [61, 62], particolarmente adatta per soddisfare i requisiti delle applicazioni di analisi video.

Questa infrastruttura ha una topologia eterogenea in quanto usa una topologia mista ad anello e stella nelle sottoreti (identificata con il termine *wheel* per la sua classica forma a ruota), mentre le varie sottoreti sono collegate mediante una topologia a stella; un esempio di questa topologia è raffigurato in Figura 3.6. Nella figura sono mostrate quattro sottoreti coordinate dai *super switch* 1,2,3,4 al centro delle reti periferiche; ognuna delle sottoreti è collegata al *root switch* 0 che coordina il trasferimento dati tra le sottoreti; infine ogni sottorete include 7 *sub switch* ai quali sono collegati i processori che implementano i task dell'applicazione (PE1,PE2,PE3,PE4 in figura).

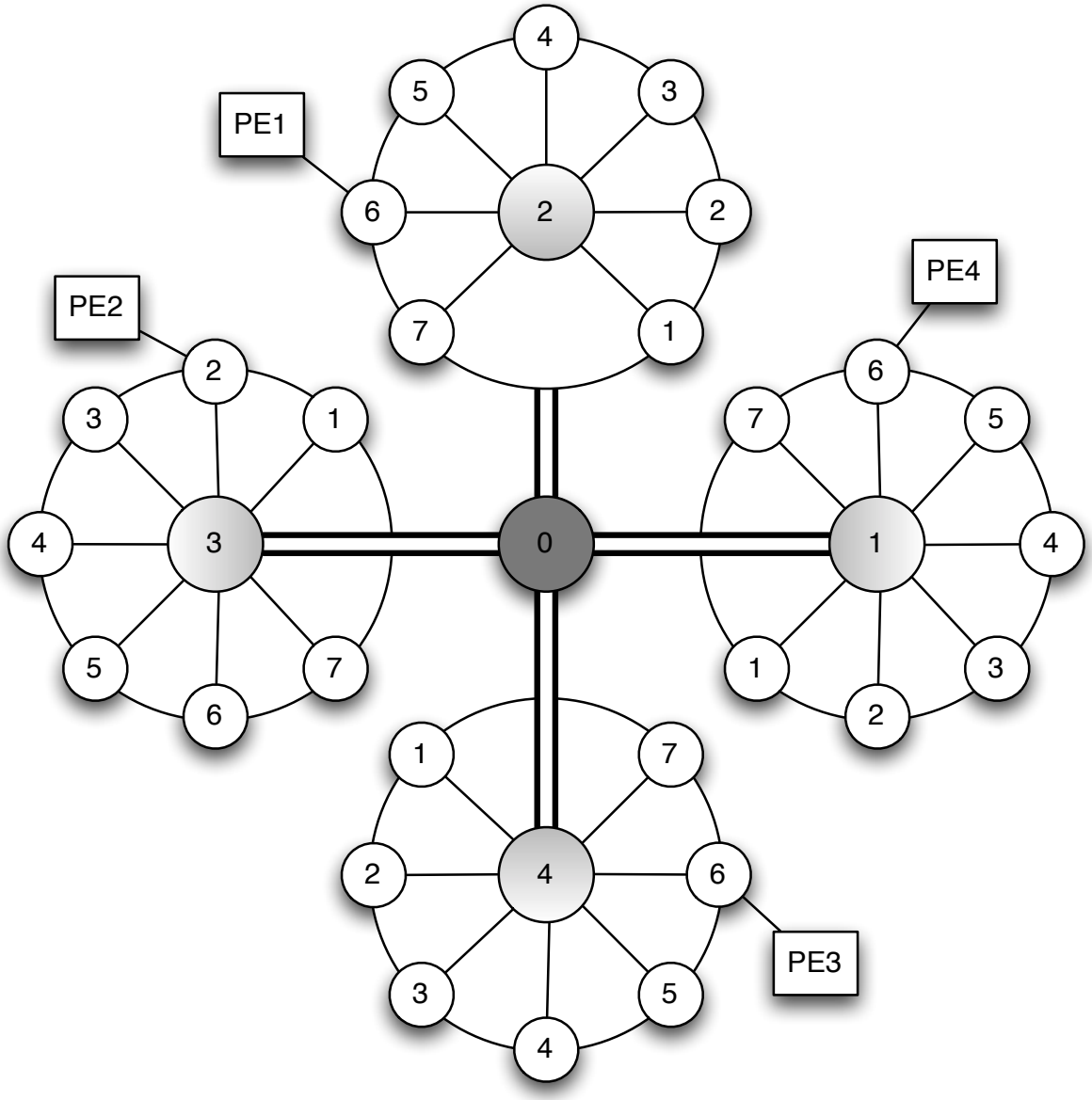

Figura 3.6: Wheel-Start Network On Chip[55].

3.3.3 CAP-OS

RAMPSoC è l'unico tra i framework in analisi che supporta sistemi che si adattano al carico di lavoro a runtime. Ricordando quanto anticipato nel Capitolo 2 per quanto concerne la gestione runtime del dispositivo, il layer software di questi sistemi necessita di impiegare strategie svolte ad ottimizzare i tempi di riconfigurazione del dispositivo al fine di minimizzare l'*overhead* dovuto alla riconfigurazione stessa.

CAP-OS [38] è il layer software utilizzato per gestire l'architettura di RAMPSoC; esso ha come input la descrizione, in forma di taskgraph, delle applicazioni che possono essere eseguite e per ognuna di esse le diverse implementazioni disponibili con i relativi tempi di esecuzione e riconfigurazione. CAP-OS sfrutta il riuso dei componenti ed il *prefetch* delle riconfigurazioni; fornisce inoltre la possibilità di interrompere una riconfigurazione in atto se un task con maggior priorità è pronto per essere riconfigurato; affinché un task possa essere riconfigurato è necessario che tutti i suoi predecessori siano già stati configurati. L'algoritmo che viene sfruttato da CAP-OS per gestire le riconfigurazioni è rappresentato in Figura 3.7. Analizzando la figura risulta evidente come l'approccio proposto cerchi di minimizzare il numero delle riconfigurazioni necessarie riutilizzando, ove possibile, processori già presenti e liberi oppure che saranno liberi a breve.

3.4 XPilot

Come anticipato nell'introduzione di questo capitolo XPilot [63] è l'unico prodotto tra quelli presentati ad avere avuto un seguito in ambito commerciale. Inizialmente sviluppato presso il dipartimento di Computer Science dell'università della California è stato poi acquistato e sviluppato da Xilinx arrivando al prodotto AutoESL [39] in commercio dallo scorso anno.

Il framework segue la struttura generica descritta nel Capitolo 1, partendo da una descrizione dell'applicazione da realizzare in System C o C, arriva ad avere una sua descrizione in codice HDL sintetizzabile pronto per l'implementazione nel caso di un suo uso su dispositivi hardware, oppure una traduzione ed ottimizzazione per dispositivi multiprocessore.

Il flusso di lavoro seguito dal framework è rappresentato in Figura 3.8; gli input sono l'applicazione descritta in C o System C e la descrizione della piattaforma target. Il flusso è diviso nei seguenti passi:

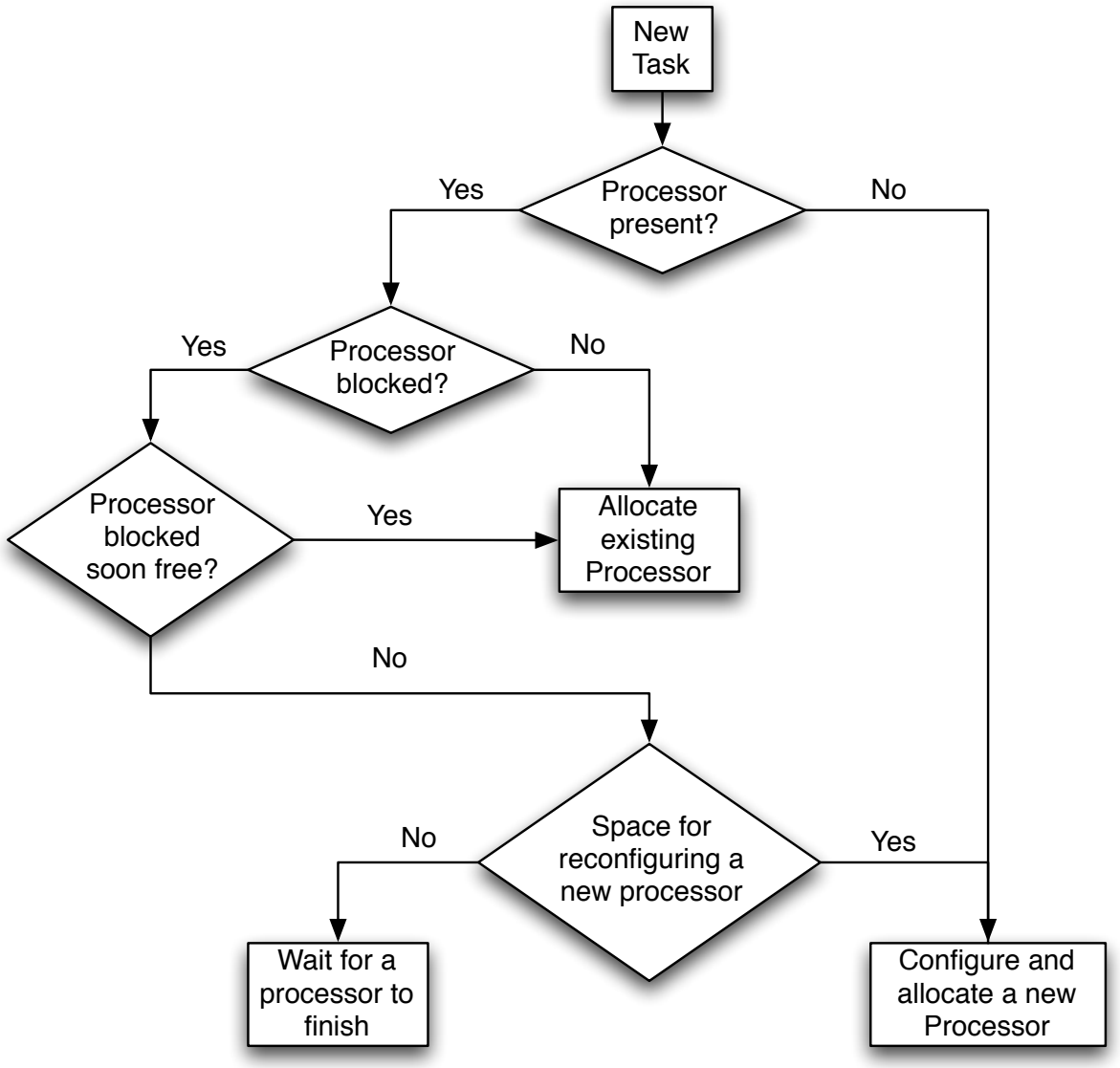

Figura 3.7: CAP-OS[38].

- Front-End: Analizza il codice dell'applicazione e ne genera una descrizione in forma di grafo (CFG[64], DFG[65])
- Analisi: in questa fase chiamata viene effettuata un'analisi ed una profilazione dell'applicazione e queste informazioni vengono usate per effettuare l'assegnamento delle varie parti dell'applicazione ai processori; le informazioni generate in questa fase vengono salvate in un modello chiamato *System-Level Synthesis and Data Model*;
- a questo punto vengono generati gli output ed il flusso di lavoro si divide in tre parti indipendenti volte a realizzare una la traduzione del codice per i GPPs nel sistema,

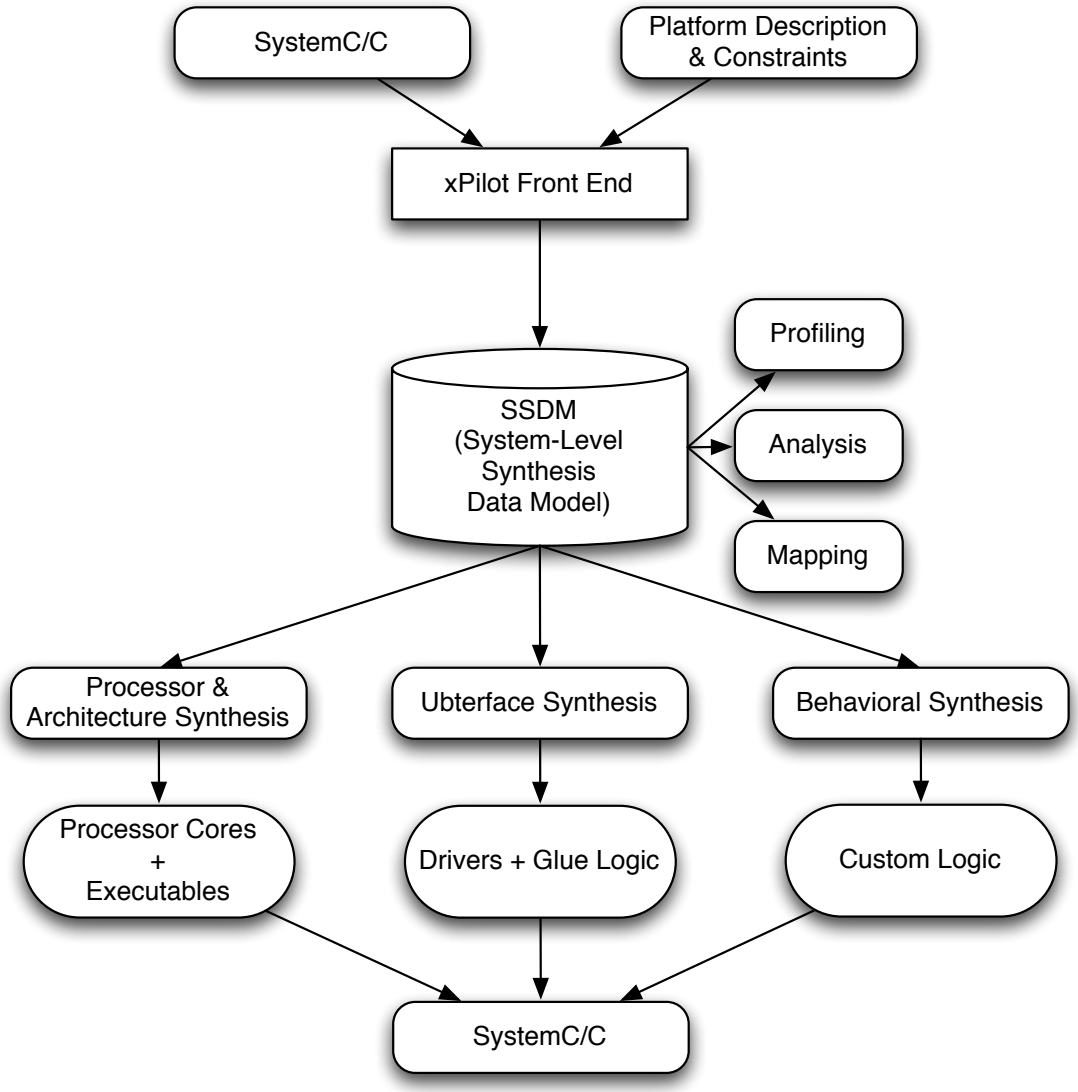

Figura 3.8: XPilot[63].

una la traduzione in HDL delle parti che andranno implementate su hardware dedicato e l'ultima volta a generare le interfacce ed i drivers per la comunicazione tra i vari componenti.

La parte divenuta commerciale è quella relativa allo sviluppo di componenti hardware dedicati con un'attenzione particolare al problema del HLS. Questa parte è stata affrontata in molti lavori dal gruppo di XPilot [66, 67, 68, 69, 70, 71] ed è stata ulteriormente migliorata da Xilinx per arrivare ad avere non solo una descrizione in Register Transfer Level (RTL), e quindi poco leggibile, dell'applicazione, ma una descrizione *behavioral* che quindi ne permette un uso molto più flessibile da parte dello sviluppatore.

Per quanto riguarda il prodotto sviluppato da Xilinx, AutoESL, esso si basa sulla parte di HLS sviluppata in questo lavoro avendo come ulteriore input al flusso un insieme di direttive, che esprimono le decisioni dello sviluppatore in relazione all'implementazione dei vari componenti; tali direttive possono riguardare l'assegnamento dei vari task ai diversi processori oppure direttive di traduzione quali ad esempio il livello di *loop unrolling* desiderato per un particolare ciclo del codice.

3.5 Sommario

In questo capitolo sono stati presentati i principali framework per lo sviluppo di *MPSoC* presenti nello stato dell'arte. In particolare sono stati presentati strumenti che supportano lo sviluppo di sistemi classici come PandA, Daedalus e XPilot e strumenti in grado di supportare lo sviluppo di *MPSoC* riconfigurabili come quello che si vuole sviluppare in questo lavoro, ovvero RAMPSoC. Questo strumento pur affrontando e risolvendo molti dei problemi che si vogliono affrontare in questo lavoro pecca nella mancanza di una completa automatizzazione del flusso in quanto lascia a carico del progettista la riscrittura del codice dell'applicazione al fine di supportare la comunicazione tra i core ed i processori che verranno generati. Sebbene sarebbe molto interessante utilizzare RAMPSoC come base dalla quale partire per sviluppare questo lavoro, questo non è possibile in quanto il codice sorgente di questo progetto non è stato rilasciato. Infine per quanto riguarda XPilot e Daedalus essendo essi fortemente orientati allo sviluppo di sistemi per supportare una singola applicazione operando una fase di analisi molto spinta in questa direzione risulterebbe sconveniente integrarsi in questi flussi. Fatte queste considerazioni la scelta dello strumento da usare come base per il lavoro proposto è ricaduta su PandA in quanto la struttura disaccoppiata tra la parte di Hw/Sw co-design e la fase di sintesi di alto livello permette una facile integrazione al fine di orientare il flusso in oggetto alla realizzazione di sistemi anche molto complessi.

Capitolo 4

Metodologia

Questo capitolo fornisce una descrizione del problema trattato nel lavoro mostrando nel dettaglio le problematiche affrontate e le relative soluzioni. La Sezione 4.1 fornisce una descrizione delle caratteristiche e delle proprietà che il framework sviluppato deve avere per rispondere alle necessità dello sviluppatore elencate nei capitoli precedenti; la Sezione 4.2 analizza in dettaglio le singole parti sviluppate in questo lavoro ponendo particolare attenzione alla descrizione del flusso di lavoro proposto ed infine la Sezione 4.3 fornisce i dettagli della parte relativa alla gestione dell'intero sistema a runtime usata al fine di adattare il funzionamento del dispositivo alle condizioni di lavoro variabili.

4.1 Definizione del problema

Come anticipato nei capitoli precedenti, l'incessante richiesta di performance sempre migliori ha spinto diversi lavori di ricerca a sviluppare architetture in grado di garantire tali performance contenendo al contempo il consumo di potenza e l'occupazione in termini di area. A questo scopo un settore importante di ricerca è quello riguardante le architetture riconfigurabili al fine di realizzare sistemi in grado di adattarsi a runtime alle diverse condizioni di lavoro supportando anche l'esecuzione di più processi contemporaneamente. Spesso la realizzazione di questi sistemi avviene in modo quasi completamente manuale in quanto i framework per lo sviluppo di sistemi MPSoC non supportano ancora appieno questa casistica. In particolare il flusso di lavoro richiede l'intervento del progettista nella fase di definizione delle interfacce verso i diversi core e per la loro integrazione all'interno del meccanismo hardware oppure software atto alla loro gestione.

Vi è quindi la necessità di realizzare un framework in grado di assistere lo sviluppatore in tutte le fase del progetto sollevandolo dai compiti quali la scrittura dei *driver* per la comunicazione tra le parti hardware e software e la creazione delle interfacce dei componenti hardware in particolare considerando la necessità di dover avere interfacce standard per i componenti hardware al fine di supportare la riconfigurazione parziale dinamica. Il framework deve inoltre avere come obiettivo quello di realizzare un livello di gestione software dell'intera architettura al fine di gestire in modo automatico la gestione dello scheduling e della riconfigurazione dei vari core in relazione alla diverse condizioni di lavoro del sistema. Questa parte se svolta in modo automatico è di fondamentale importanza in quanto, allo stato attuale, un sistema di questo tipo pone la necessità di doversi appoggiare a soluzioni la cui integrazione con il flusso di lavoro non è automatizzata. Una di queste soluzioni è ad esempio l'uso di distribuzioni linux ad hoc [72] al fine di riuscire a gestire lo scheduling di più processi, tuttavia in questo caso sorge il problema di dover integrare all'interno dello scheduler di sistema la parte relativa alla comunicazione ed alla gestione della parte hardware; un'altra soluzione consiste invece nella creazione di un livello software ad hoc per l'applicazione in fase di sviluppo con le problematiche di dover sviluppare ogni volta algoritmi che possono presentare problemi quali la presenza di *deadlock* e possibilità di *starvation* dei processi in esecuzione e che quindi necessitano di capacità e conoscenze di programmazione che possono non essere alla base del percorso formativo di un progettista hardware. Il framework in oggetto deve quindi permettere di automatizzare la creazione di questo livello software personalizzato per il sistema in

fase di sviluppo creando in automatico le interfacce necessarie ed i driver di comunicazione e fornendo diversi algoritmi e politiche per la gestione dello scheduling e della riconfigurazione che possono essere scelti in fase di creazione dell'architettura. Infine un framework con queste caratteristiche deve poter essere utilizzato per diversi scopi quali:

- verificare i vantaggi che un'applicazione ottiene da una sua implementazione parallela su MPSoC;
- verificare nuove politiche di scheduling e riconfigurazione direttamente sul dispositivo in modo semplice e rapido con la possibilità di generare diversi casi di studio e carichi di lavoro provenienti da applicazioni reali;
- creare nuovi modelli architetturali di MPSoC e validarli sempre alla luce di carichi di lavoro reali;

4.2 Il framework proposto

Il framework sviluppato in questo lavoro si pone come obiettivo quello di affrontare tutte le problematiche evidenziate alla sezione precedente; verranno ora illustrate nel dettaglio le soluzioni adottate per affrontare tali problemi ed in particolare in questa sezione viene presentato il flusso generale seguito dal framework; per quanto riguarda i dettagli implementativi del lavoro si rimanda ai capitoli 6 e 7.

Lo scopo del framework è quello di creare un sistema MPSoC con supporto alla riconfigurazione, con la possibilità di adattarsi a runtime al carico di lavoro e con la possibilità di eseguire più applicazione in parallelo. Il flusso ha come obiettivo quello di assistere lo sviluppatore in ogni fase del progetto e si propone quindi come flusso semi automatico che, una volta definiti gli input da esso richiesti, genera come output l'insieme di file di configurazione pronti ad essere portati sul dispositivo finale. Il framework proposto si occupa dell'automazione dell'integrazione dell'intero sistema, ma si appoggia al flusso Xilinx per quanto riguarda la creazione del sistema, tramite *Xilinx Platform Studio*[73], e la sintesi finale del progetto sfruttando il flusso di lavoro di *PlanAhead*[74]. Diamo ora alcune definizioni che meglio specificano i concetti utili a comprendere quanto descritto nel resto del lavoro.

Definizione 4.2.1 (Applicazione) *Il termine applicazione identifica l'insieme delle linee di codice volte ad eseguire un particolare compito; l'applicazione può essere divisa in task. Nel sistema finale possono*

essere in esecuzione fino ad N *applicazioni diverse tra loro in modo concorrente,* N *identifica il numero totale delle applicazioni prese in considerazione in fase di design del sistema. Ogni applicazione è univoca nel sistema.*

Definizione 4.2.2 (Task) *Ogni task identifica una particolare parte dell'applicazione che unitamente agli altri task ad essa associati partecipano alla realizzazione dell'applicazione stessa; i task possono avere diverse implementazioni. Ogni task è univoco all'interno dell'intero sistema. Un task identifica una funzione nel codice dell'applicazione a cui si riferisce.*

Definizione 4.2.3 (Implementazione) *Un'implementazione di un singolo task fa riferimento ad una sua particolare realizzazione; con questo termine si identifica quindi un particolare algoritmo software volto ad essere eseguito su di un processore, oppure il suo equivalente hadware che deve essere istanziato come un core hadware statico o riconfigurabile a runtime. Le implementazioni possono essere condivise tra i diversi task nel sistema, indipendentemente che essi appartengano o meno alla stessa applicazione. Ogni implementazione è caratterizzata da un insieme di proprietà quali ad esempio il tempo di esecuzione, di riconfigurazione, oppure la sua occupazione in termini di area.*

A partire da queste definizioni è importante contestualizzare le problematiche che il framework deve affrontare; in particolare per riuscire a generare il sistema richiesto, il framework ha la necessità di identificare i vari task presenti nell'applicazione ed individuare quale sia l'implementazione migliore da eseguire assegnandola alle unità processanti presenti sull'architettura finale. Lo stato dell'arte relativo a queste problematiche, sebbene molto vasto, non è ancora condiviso appieno soprattutto per quanto riguarda l'ambito industriale. La scelta fatta in questo lavoro è quindi quella di lasciare al progettista queste scelte specificandole tramite un insieme di direttive in input al flusso; tale scelta è giustificata soprattutto considerando il fatto che il problema dell'assegnamento è ad oggi risolto solamente nel caso di sviluppo di sistemi per i quali si conosce in modo deterministico il loro comportamento a runtime. Tuttavia questa ipotesi non è valida in questo lavoro considerato che il sistema è inerentemente non deterministico trovandosi a dover eseguire più applicazioni in parallelo in risposta a comandi ricevuti dall'esterno. Alla luce di questo si è deciso di non cercare di automatizzare questo passo in quanto permettere allo sviluppatore di scegliere a quali unità processanti assegnare i task consente ad esso di fornire ai task più critici una maggior flessibilità implementativa che ha come risultato una maggior possibilità che questi vengano eseguiti senza essere ritardati.

Il flusso di lavoro proposto è evidenziato in Figura 4.1 e può essere diviso in quattro fasi principali di seguito analizzate.

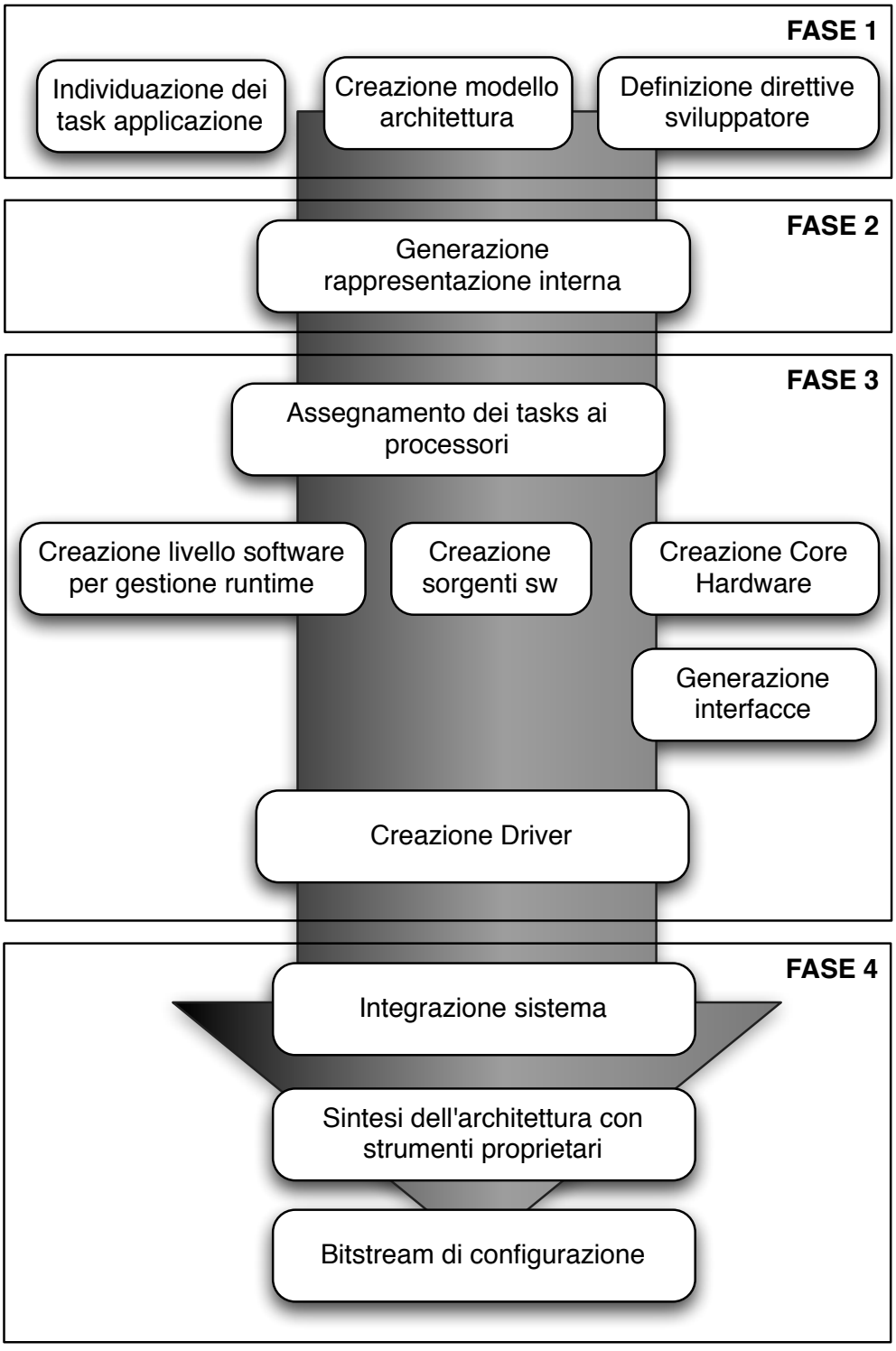

Figura 4.1: Flusso di lavoro proposto.

4.2.1 Fase 1: Preparazione degli input

La fase di preparazione degli input è l'unica fase non automatica del flusso ed in quanto tale completamente a carico del progettista. Tale scelta è stata effettuata tenendo in considerazione che le decisioni progettuali prese in questa fase hanno un grande impatto sulle performance finali del sistema e le capacità del progettista sono al giorno d'oggi ancora superiori alle soluzioni disponibili in letteratura in questo ambito. In particolare il progettista deve fornire i seguenti input:

- *Individuazione dei task dell'applicazione*: In questa fase il progettista identifica il vari task dell'applicazione e ne esplicita anche il loro parallelismo, ulteriori dettagli su questo passo sono dati nel Capitolo 6;
- *Definizione del modello dell'architettura*: specifica i componenti dell'architettura, le loro proprietà e le loro interconnessioni; vengono qui specificati ad esempio il numero di processori presenti nell'architettura, le memorie e i componenti di comunicazione;
- *Definizione delle direttive dello sviluppatore*: In questa fase lo sviluppatore specifica informazioni aggiuntive necessarie al flusso di lavoro per generare correttamente il sistema. In particolare lo sviluppatore deve specificare:
  - Assegnamento dei task alle unità processanti: questo serve per mappare i vari task individuati sulle unità processanti presenti nel modello dell'architettura. Ogni task può essere assegnato anche a più unità processanti; in questo caso il flusso si occuperà di creare un'implementazione per ogni unità lasciando al sistema runtime la possibilità di scegliere la migliore da utilizzare in base al carico di lavoro al quale il sistema è sottoposto;
  - Definizione dei loro tempi di esecuzione: il tempo di esecuzione è quello poi usato dal sistema runtime al fine di calcolare le metriche necessarie per la scelta della migliore implementazione da eseguire per il singolo task;
  - Definizione degli eventuali tempi di riconfigurazione: analogamente al tempo di esecuzione concorrono alla definizione della metrica per la scelta della dell'implementazione; ovviamente questa informazione è rilevante solo nel caso si tratti di un core riconfigurabile;

4.2.2 Fase 2: Analisi indipendente dall'architettura

- Creazione della rappresentazione interna in forma di *task graph* gerarchico: In questa fase viene analizzata l'applicazione, viene effettuata la sua suddivisione in task e viene generata la sua rappresentazione in termini di task graph. Le trasformazioni e i passi di analisi dell'applicazione applicati in questa fase sono totalmente indipendenti dall'architettura target e consistono nell'analisi della struttura dell'applicazione con i metodi e le tecniche provenienti dalla teoria dei compilatori [22].

4.2.3 Fase 3: Analisi dipendente dall'architettura

In questa fase vengono effettuati tutti i passi volti alla creazione dei componenti hardware e software atti ad implementare l'applicazione specificata. In particolare vengono svolti i seguenti passi:

- Assegnamento dei task: Le direttive relative all'assegnamento dei task alle unità processanti vengono analizzate e il task graph creato nella Fase 3 viene annotato con tali informazioni;
- Creazione dei componenti software: per i task che hanno un'implementazione software vengono creati i sorgenti relativi;
- Creazione dei componenti hardware: i task annotati per avere un'implementazione hardware attraversano le due fasi seguenti:
  - Fase di sintesi di alto livello: il codice relativo al task in analisi viene tradotto in modo automatico nel corrispondente codice Hardware Description Language (HDL);
  - Creazione delle interfacce: il codice HDL viene arricchito con le informazioni relative all'interfaccia necessaria per la successiva integrazione nel sistema completo;
- Creazione dei driver: la parte software del sistema viene arricchita con i driver atti a garantire la corretta comunicazione tra i componenti hardware e software;
- Creazione del livello software per la gestione runtime: In questo passo vengono analizzate le direttive fornite dallo sviluppatore ed i task da esso individuati al fine di creare il layer software per la gestione del sistema a runtime; in particolare vengono affrontati i seguenti passi:

- Allocazione variabili condivise: i vari task dell'applicazione vengono analizzati e qualora vi sia una condivisione di parametri attraverso la memoria condivisa, questi i parametri relativi vengono allocati fin dalla loro inizializzazione nella memoria condivisa e lo spazio di memoria presente sull'architettura viene gestito automaticamente dal framework;
- Creazione strutture dati relative all'applicazione: vengono anche create in questo passo le varie strutture dati per la gestione del sistema e vengono inizializzate secondo le decisioni prese dallo sviluppatore e specificate con le direttive sopra descritte;
- Creazione sorgenti: infine si procede alla creazione vera e propria dei sorgenti del sistema di gestione runtime.

4.2.4 Fase 4: Creazione MPSoC

In questa fase vengono attuati i passi per la generazione del sistema finale a partire dagli output della fase precedente; come anticipato il framework si basa sui flussi di lavoro proposti da Xilinx[19] e quindi sugli strumenti da questa proposti. In particolare in questa fase si compiono i seguenti passi:

- Integrazione del sistema: I core hardware dotati delle loro interfacce ed i sorgenti software vengono integrati nel sistema finale;
- Creazione automatica Xilinx: Il sistema nel suo complesso viene gestito come progetto di *Xilinx Platform Studio*[73] e la fase di creazione del sistema avviene automatizzando i passi effettuati da questo strumento;
- Gestione automatizzata della riconfigurazione tramite PlanAhead[74]: In caso di sistema riconfigurabile è necessario integrare il framework in via di sviluppo con gli strumenti messi a disposizione da Xilinx; in particolare in questa fase il framework interagisce con il flusso *PR* automatizzando i passi che devono essere effettuati dall'utente tramite l'uso di PlanAhead;
- Sintesi dell'architettura e creazione dei bitstreams di configurazione: Infine vengono creati i bitstream sempre integrando il flusso con gli strumenti proposti da Xilinx ed automatizzandone i passi necessari; alla fine di questo passo i bitstream sono disponibili per essere configurati sul dispositivo finale.

4.3 Gestione del sistema a runtime

Il flusso di lavoro appena descritto rispecchia nei suoi passi, a meno della fase 4, il flusso di lavoro generale rappresentato in Figura 1.2 ed al pari di questo elabora una singola applicazione alla volta affrontando per essa tutti i passi delle fasi 1,2,3 ottenendo come risultato però non il sistema finale, ma una serie di prodotti intermedi che saranno poi integrati nella Fase 4 al fine di produrre un sistema in grado di eseguire in modo concorrente le applicazioni analizzate. Andiamo ora ad analizzare in dettaglio le soluzioni proposte per la realizzazione del sistema di gestione a runtime al fine di supportare la riconfigurazione e l'esecuzione concorrente di più applicazioni; per i dettagli implementativi si faccia riferimento al Capitolo 7.

L'algoritmo per la gestione a runtime dell'applicazione deve avere le seguenti caratteristiche:

- Consentire l'esecuzione concorrente di diverse applicazioni;
- Gestire la riconfigurazione dei task in modo intelligente e quindi:
  - Mascherare il più possibile il tempo di riconfigurazione anticipando la riconfigurazione di un task che verrà usato in futuro;
  - Consentire il riuso dei task quando possibile al fine di ridurre il numero di riconfigurazioni;
  - Scegliere in modo ottimale quale task riconfigurare in base a politiche che garantiscano la riduzione del tempo di esecuzione
- Gestire lo scheduling dei vari task
- Gestire il trasferimento dati tra i task e con la memoria condivisa
- Scegliere a runtime in base al carico di lavoro attuale la migliore implementazione da usare per eseguire un singolo task

Al fine poter garantire il raggiungimento di questi obiettivi sono necessari da un lato gli algoritmi opportuni e dall'altro una serie di input che devono essere forniti dal flusso descritto al passo precedente e che devono essere organizzati in apposite strutture dati. Verrà ora data una descrizione delle caratteristiche che gli input e gli algoritmi devono possedere al fine di soddisfare i requisiti specificati.

4.3.1 Input e strutture dati

In primo luogo il sistema di gestione a runtime ha bisogno di ricevere in ingresso una serie di input:

- la descrizione delle applicazioni in forma di task graph
- la descrizione dei singoli task
- le possibili implementazioni disponibili ed i relativi sorgenti (funzioni da invocare) e bitstream di configurazione
- le stime dei tempi di esecuzione e riconfigurazione
- la descrizione delle unità processanti

la descrizione dell'implementazione e del formato usato per questi input verrà discussa, come anticipato, nel Capitolo 7; tuttavia è utile specificare ora quelle proprietà ed attributi che devono essere presenti al fine di permettere al sistema di gestione a runtime di assolvere ai compiti richiesti. In particolare è utile fornire una descrizione relativa agli input inerenti alle applicazioni, ai task, alle implementazioni ed alle unità processanti.

Applicazioni

La struttura dati che codifica la singola applicazione necessita di conoscere in primo luogo i task che la compongono, ma un altro dato fondamentale che deve essere a disposizione è la conoscenza del fatto che l'applicazione in oggetto sia al momento in esecuzione o meno. Questo risulta fondamentale in quanto, per una gestione ottimale della riconfigurazione, è necessario, come spiegato in precedenza, cercare di riconfigurare i task il prima possibile; tuttavia questo processo può risultare pesante in quanto vi è la necessità di analizzare ogni task al momento attivo, quindi un attributo di questo tipo a livello dell'oggetto applicazione permette di ridurre il tempo necessario ad effettuare questa operazione; durante la fase di gestione a runtime il livello software può, controllando questo attributo, analizzare esclusivamente i task delle applicazioni attualmente in esecuzione ed ignorare quelli che appartengono ad applicazioni non attive o che sono già stati eseguiti.

Task

La struttura dati che rappresenta il task di un'applicazione necessita della seguente serie di attributi:

- le stime dei tempi di esecuzione: si noti che sebbene più task possano condividere una stessa implementazione, il tempo di esecuzione è proprio del singolo task e non dell'implementazione stessa; basti ad esempio pensare ad un ciclo con uno stesso corpo ma invocato due volte con un numero di iterazioni differenti, i due task in oggetto condivideranno la stessa implementazione, tuttavia i tempi di esecuzione saranno diversi e direttamente proporzionali al numero delle iterazioni eseguite.
- la codifica dello stato del task: questa risulta fondamentale per la gestione dello scheduling e della riconfigurazione; in particolare gli stati in cui il singolo task si può trovare sono:
  - in attesa di riconfigurazione: Il task necessita di essere riconfigurato od assegnato su di un core hardware oppure deve essere assegnato ad un processore software;
  - da eseguire: il task è tra quelli attivi nell'applicazione;
  - in attesa della terminazione di qualche predecessore: il task sta aspettando che qualche predecessore termini; a seconda dell'algoritmo di gestione a runtime utilizzato questo task può essere preso in considerazione o meno al fine della riconfigurazione;
  - pronto per essere eseguito: il task è pronto per essere eseguito in quanto i suoi predecessori sono terminati; prima di essere eseguito può essere tuttavia necessario assegnarlo ad un'unità funzionale;
  - in esecuzione: il task è al momento in esecuzione;
  - eseguito: il task è già stato eseguito;

  ovviamente questi stati non sono mutualmente esclusivi tra loro, ma alcuni di essi possono essere veri allo stesso istante, ad esempio un task può essere al contempo da eseguire ed in attesa di riconfigurazione.
- le informazioni relative al suo utilizzo (numero di esecuzioni, numero di riconfigurazioni) utili in quanto forniscono statistiche che un eventuale sviluppatore di politiche di scheduling e riconfigurazione può usare per personalizzare gli algoritmi da lui sviluppati.

Implementazioni

Le implementazioni come detto possono essere condivise tra più task. Ad ogni implementazione hardware è associato il relativo bitstream che deve di essere configurato ed il relativo tempo di configurazione. Oltre a queste informazioni è necessario, come avviene per i task, che l'oggetto che le rappresenta contenga le informazioni relative al loro utilizzo come ad esempio il numero di volte che sono state utilizzate e quante volte è stato necessario riconfigurarle.

Il task graph

Il task graph di un'applicazione è usato nella fase della gestione del sistema a runtime al fine di individuare quando un task è pronto ad essere eseguito. In particolare un task può essere eseguito solamente quando tutti i suoi predecessori sono terminati. Il task graph necessita di un'implementazione molto leggera in quanto deve essere navigato per prendere le decisioni di scheduling a runtime.

Unità processanti

Le unità processanti infine devono contenere attributi che ne codificano lo stato:

- unità disponibile
- unità occupata
- unità in fase di configurazione
- untià alla quale è stato assegnato un task che deve essere eseguito in futuro

Al contrario degli stati dei task questi sono in mutua esclusione tra loro. Oltre a questi attributi l'ultima informazione necessaria è quella relativa all'algoritmo correntemente configurato su di un'unità riconfigurabile; questo attributo è quello che permette di sfruttare il riuso dei componenti hardware attualmente configurati sul dispositivo.

4.3.2 Algoritmi

Per quanto riguarda gli algoritmi da eseguire questi devono soddisfare due proprietà contrastanti: da un lato devono essere veloci e dare garanzia sulla loro correttezza essendo il compito dello scheduling critico; dall'altro devono riuscire a fornire una soluzione il quanto più vicino all'ottimo possibile. Il primo problema relativo alla correttezza dello schedule implica

che un task non deve mai essere eseguito prima che tutti i suoi predecessori siano terminati; questo problema è gestito dall'algoritmo analizzando a runtime le dipendenze attraverso la struttura dati che rappresenta il task graph. Questa soluzione che, come si vedrà nel Capitolo 7, ha un'implementazione molto semplice che garantisce la correttezza dello scheduling sia nel caso base dove il sistema non presenta elementi riconfigurabili o parallelismo a livello di applicazione, sia nel caso più complesso che è l'obiettivo di questo lavoro. Per quanto riguarda la gestione intelligente della riconfigurabilità, non essendo un problema che invalida la correttezza della soluzione finale, ma che al più introduce del ritardo nell'esecuzione si è deciso di mettere a disposizione del progettista una serie di algoritmi preconfezionati ognuno dei quali comporta un differente costo computazionale che si traduce però in una differente qualità della soluzione del problema della riconfigurazione; è compito dello sviluppatore scegliere quale usare. Per il secondo algoritmo proposto esistono due diverse versioni, una che cerca di anticipare il più possibile la riconfigurazione, mentre l'altra che aspetta che il task sia pronto prima di riconfigurarlo; questa scelta è dovuta al fatto che nel primo caso è possibile ridurre il numero di volte in cui controllare se sia opportuno o meno riconfigurare qualche area in modo da ridurre l'overhead dovuto al gestore runtime.

First Fit

L'algoritmo *first fit* prende per ogni task che deve essere riconfigurato la prima implementazione che può essere usata. La sua esecuzione è molto veloce, tuttavia non è garantito che si ottenga il massimo riuso possibile dei componenti. Se t sono i task e p le unità processanti presenti nel sistema, questo algoritmo ha una complessità pessima dell'ordine di $O(tp)$.

Best Fit

La prima versione dell'algoritmo *best fit* sceglie per ogni task che deve essere riconfigurato l'area riconfigurabile tra quelle disponibili che riduce il suo tempo di esecuzione. In questo modo viene favorito il riuso dei core hardware già configurati, tuttavia non si garantisce che le decisioni prese a livello di singolo task non obblighino a prendere una soluzione complessivamente non ottima. Se t sono i task e p le unità processanti l'algoritmo ha una complessità pari a $O(tp)$.

Best Fit con prefetch (v2)

La seconda versione dell'algoritmo *best fit* cerca di ottimizzare la scelta delle riconfigurazioni da effettuare. La scelta di assegnare o di riconfigurare un task, a differenza del caso precedente, avviene anche quando il task in questione non è ancora pronto per essere eseguito. Questa caratteristica permette all'algoritmo di anticipare le riconfigurazioni prima che queste siano effettivamente necessarie; tuttavia in questo caso l'algoritmo deve essere in grado di rivedere le proprie decisioni in quanto mappare un task non pronto ad eseguire vuol dire creare una potenziale situazione di deadlock, che quindi deve essere gestita. In sostanza l'algoritmo deve essere in di eliminare un assegnamento effettuato in precedenza a favore di un task che risulta al momento pronto ad eseguire.

4.4 Sommario

In questo capitolo si è introdotto il problema che si vuole affrontare in questo lavoro ponendo in particolare l'attenzione su come i framework per lo sviluppo di Multi Processors System on Chip (MPSoC) ad oggi esistenti non supportino appieno la riconfigurabilità e di come sia in genere difficile riuscire ad integrare un sistema che utilizzi processori, coprocessori hardware e che li gestisca in modo intelligente a runtime a fronte delle diverse condizioni di lavoro nelle quali si può trovare il dispositivo. É quindi stato illustrato il framework proposto illustrando come le varie fasi di questo siano funzionale alla soluzione del problema e di come una volta creato il dispositivo questo sia in grado, attraverso un livello software creato dal framework stesso in modo automatico, di gestire autonomamente il sistema a fronte delle diversi condizioni di lavoro nelle quali esso si può trovare.

Capitolo 5

Architettura di riferimento

In questo capitolo viene fornita la descrizione dell'architettura usata come riferimento per questo lavoro. Nella Sezione 5.1 viene descritta l'architettura nel suo complesso; le sezioni successive analizzano invece i requisiti che l'architettura mette a disposizione al fine di supportare da un lato l'integrazione in un flusso automatico e dall'altro la parte di gestione a runtime e di gestione della riconfigurazione introdotta nella Sezione 4.3. In particolare la Sezione 5.2 descrive la struttura dei core che vengono generati del flusso ed il modo in cui essi vengono integrati nel sistema, siano essi riconfigurabili o meno; la Sezione 5.3 fornisce i dettagli di come avvenga la sincronizzazione tra i vari core a runtime e quali siano gli accorgimenti adottati per garantire una corretta gestione del sistema; infine la Sezione 5.4 illustra come l'architettura così definita supporti la riconfigurazione e come questa venga gestita.

É importante notare che l'architettura qui proposta non è l'unica che può essere usata, tuttavia i dettagli e gli accorgimenti implementativi qui esposti devono essere condivisi anche dalle altre architetture eventualmente utilizzabili con il flusso presentato al fine di garantirne una corretta e rapida integrazione.

5.1 Descrizione dell'architettura

L'architettura di riferimento utilizzata per testare il framework consiste in un'architettura basata su bus con due processori ed una memoria condivisa. La motivazione che ha spinto all'uso dei due processori è che questo tipo di architettura risulta flessibile nel senso che ben si adatta ai due casi in cui si debba sviluppare un sistema statico oppure riconfigurabile. In entrambi i casi a uno dei due processori è delegato il compito di gestire lo scheduling, la sincronizzazione ed il trasferimento dei dati tra i task. Nel caso di sistema statico il secondo processore è destinato al compito di coprocessore software per eseguire i task presenti nel sistema; nel caso di sistema riconfigurabile il secondo processore è invece destinato a implementare la funzione di riconfigurazione. L'architettura così definita è presentata in Figura 5.1.

Come si vede in figura, ognuno dei due processori è connesso mediante un bus differente alla memoria condivisa; gli eventuali core hardware sono connessi al processore che svolge il ruolo di scheduler mediante il suo bus. I core hardware inoltre sono direttamente connessi alla memoria condivisa senza passare per un bus o altri meccanismi di comunicazione condivisi che sarebbero altrimenti soggetti ad un elevato livello di congestione; questa connessione avviene attraverso l'interfaccia NPI. I due processori comunicano mediante un meccanismo di *message passing* , mentre la sincronizzazione dell'intero sistema è gestita mediante un sistema di *interrupt* con una linea dedicata condivisa tra tutti gli elementi, processori o core hardware.

L'accesso alla memoria condivisa è gestito mediante il core *Multi-Port Memory Controler* (MPMC) messo a disposizione da Xilinx [75]; questo controllore gestisce gli accessi alla memoria ed è in grado di supportare fino ad 8 core contemporaneamente. Questo fornisce una limitazione relativa alla struttura generale dell'architettura finale; considerando che ognuno dei due processori necessita di avere accesso alla memoria condivisa questo implica che rimangono solamente 6 porte libere alle quali poter collegare i core hardware. Questa limitazione consente di poter gestire applicazioni che usano al massimo 6 core hardware che accedono alla memoria esterna, oppure applicazioni implementate su di un'architettura riconfigurabile che fanno uso al massimo di 6 aree riconfigurabili.

Infine questo tipo di architettura con memoria condivisa permette di semplificare la fase di gestione a runtime del dispositivo in quanto lo scheduler che sarà implementato dovrà tenere conto delle precedenze tra i task, ma non dovrà tenere conto di come questi si scambino informazioni in quanto, nel caso, saranno scambiate attraverso la memoria condivisa.

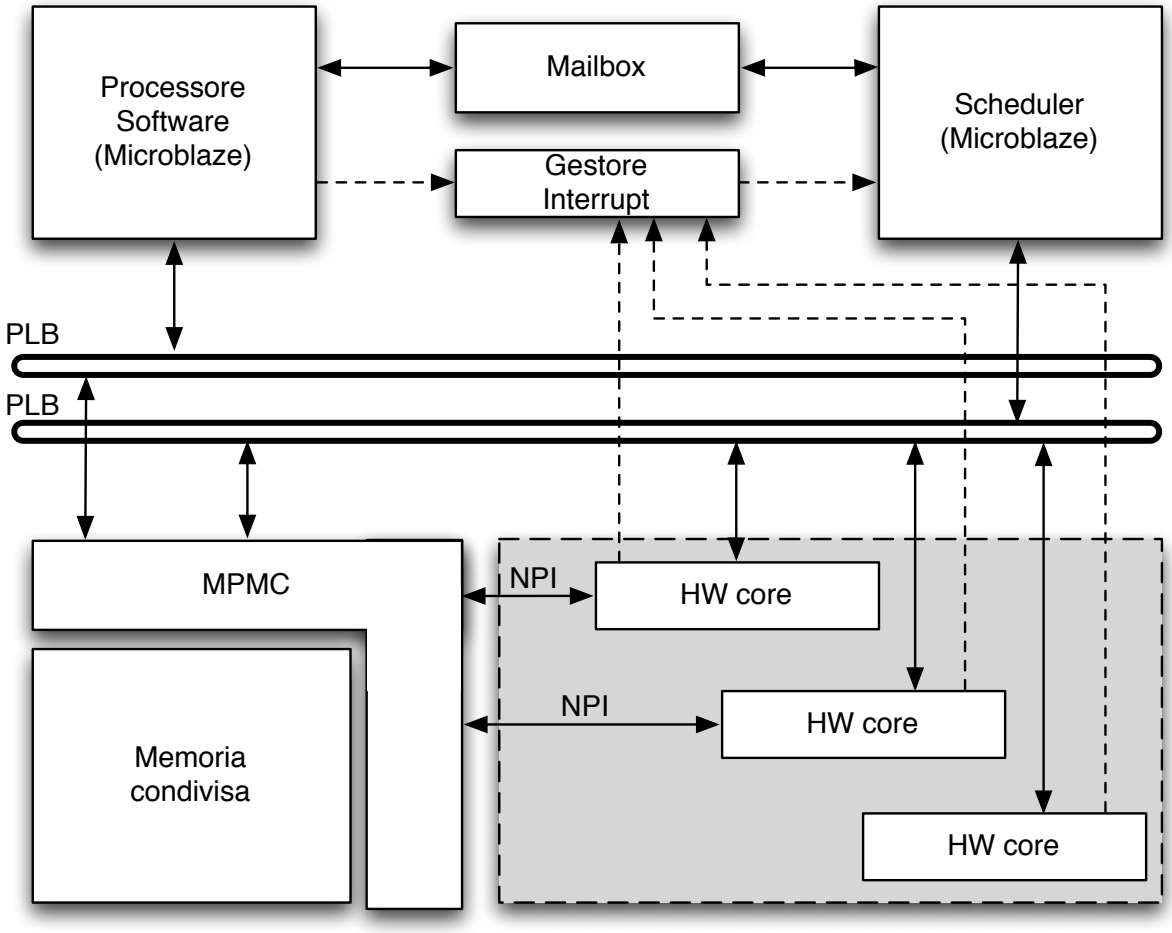

Figura 5.1: Architettura target.

5.2 Core hardware

La struttura dei core utilizzati nell'architettura hanno una struttura standardizzata in quanto sono pensati per poter essere generati automaticamente e soprattutto per poter essere generati a partire da funzioni in codice ad alto livello in quanto riflettono la struttura tipica della definizione di una funzione. Questa caratteristica permette non solo di avere un core facilmente generabile, ma anche di facilitare la generazione dei driver per l'interfacciamento rendendo trasparente la comunicazione con il core da parte dello scheduler che non si deve preoccupare di capire se il task a cui deve passare i parametri sarà eseguito in hardware oppure in software.

In Figura 5.2 è rappresentato un esempio di core che verrà integrato nell'architettura. Come si vede il core contiene una serie di porte di ingresso, uno per ogni parametro della funzione da cui è generato, e una porta di uscita facoltativa generata nel caso la funzione abbia un valore di ritorno. Il trasferimento dati tra il bus e la logica del core è supportato da una serie

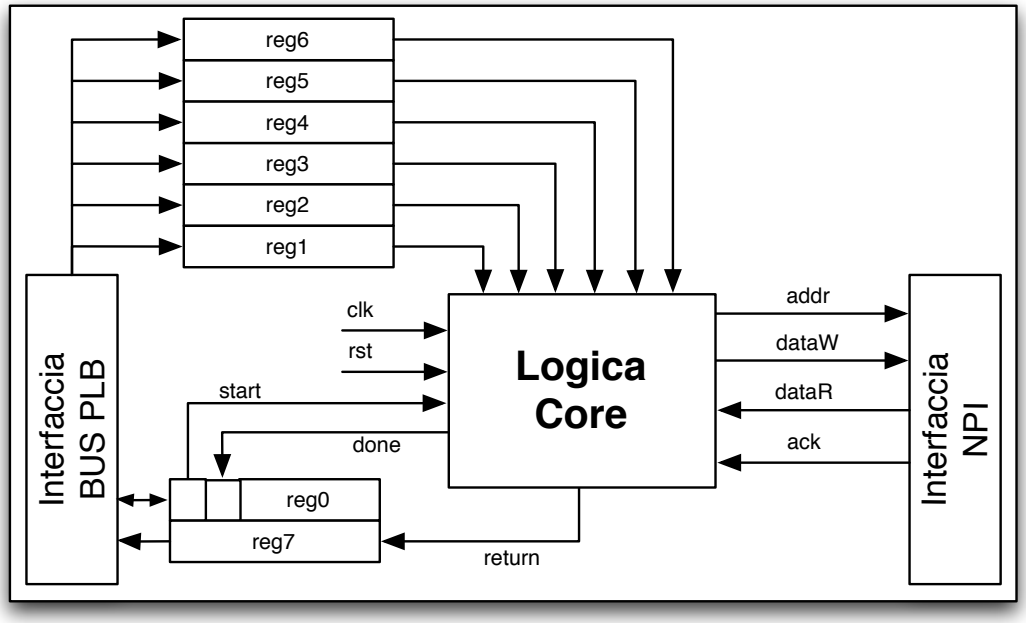

Figura 5.2: Struttura dei core hardware.

di registri, al massimo 8 per ogni core: 6 dedicati ai parametri di ingresso (da *reg1* a *reg5* in figura), uno per il valore di ritorno (*reg7* in figura) ed infine un ultimo registro dedicato a contenere il valore dei segnali di controllo del core (*reg0* in figura). Il core inoltre necessita di un'interfaccia vero il bus che lo collega al Microblaze che funge da scheduler ed eventualmente un'interfaccia di memoria NPI nel caso il core debba accedere alla memoria condivisa. Infine è importante notare come una struttura così standard faciliti l'utilizzo di questa tipologia di core nell'ambito di un sistema ricofigurabile che, in quanto tale, necessita di un'interfaccia con una struttura fissa; la descrizione di come il core appena descritto possa essere esteso per un'implementazione riconfigurabile è presentata nella Sezione 5.4.

In Figura 5.3 è infine mostrato un esempio di come il codice riportato in Listato 5.1 venga tradotto in core con la struttura appena descritta.

Listato 5.1 Esempio per la generazione del core hardware

```
1 int foo(int a, int b){
2         int c=a+b;
3         return c;
4 }
```

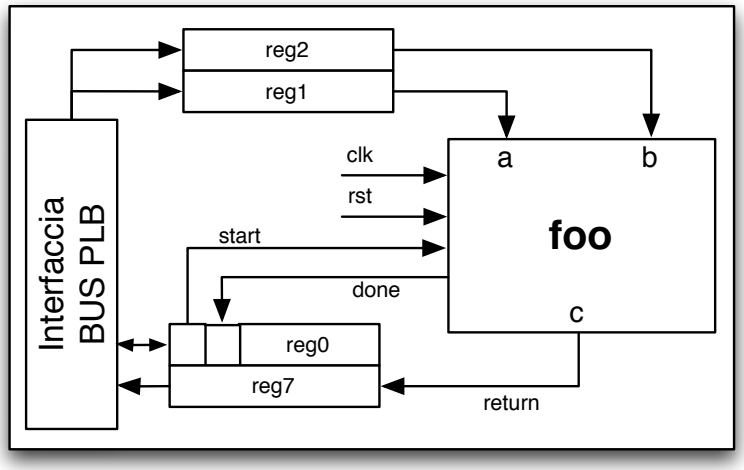

Figura 5.3: Esempio di core generato da una funzione C.

5.3 Supporto per la gestione a runtime

Per quanto riguarda la gestione del sistema a runtime sono necessari meccanismi che garantiscano la possibilità di trasferimento dati e la gestione di eventi quali terminazione dell'esecuzione dei task e delle riconfigurazioni. I sistemi di comunicazioni supportati sono il *message passing* tra i due processori ed il sistema di interrupt usato per comunicare quando un'unità processante ha terminato il compito ad esso assegnato. Il sistema di gestione a runtime per garantire la correttezza dello schedule generato necessita di meccanismi per il trasferimento dei dati a vari core. In questo ambito un ruolo fondamentale è dato dalla struttura dei core hardware che molto simile al corrispettivo della funzione del linguaggio ad alto livello da cui è generata. In questo modo è possibile definire dei driver per la comunicazione tra i vari elementi che sono a loro volta standard, ovvero un task da eseguire su di un software sarà invocato allo stesso modo di un task da eseguire su di un core hardware dedicato e la struttura dei driver sarà abbastanza simile nei due casi. Per la gestione degli eventi di terminazione invece è presente un sistema di gestione degli *interrupt* tale per cui ad ogni core hardware ed ai due processori software è associata una linea di interrupt condivisa, a questa linea è collegato anche l'ICAP per la parte di riconfigurazione. Quando un'unità processante termina il compito ad essa assegnato, comunica tramite un *interrupt* al processore che funge da scheduler la terminazione; lo scheduler quindi gestisce l'*interrupt* andando per ad aggiornare le strutture dati relative ai task e alle unità processanti con le informazioni sul loro nuovo stato.

5.4 Supporto alla riconfigurazione

Al fine di supportare la riconfigurazione è necessario definire delle aree riconfigurabili, le quali devono mantenere la stessa interfaccia verso il resto del sistema e cambiare solo nella funzionalità implementata. Questa caratteristica è stata implementata astraendo ulteriormente il core descritto nella Sezione 5.2 per fare in modo di disaccoppiare l'interfaccia sul bus con l'interfaccia vera e propria del core ovvero inserendo un ulteriore livello tra l'interfaccia verso il bus e la parte contenente i registri che poi vengono mappati sulle porte di ingresso ed uscita della logica del core.

La struttura del core così definita è rappresentata in Figura 5.4; come si può notare la parte descritta nella sezione relativa alla descrizione dei core è ora posta ad un livello inferiore della gerarchia. Questo spostamento non ne cambia la struttura e quindi quanto detto nella

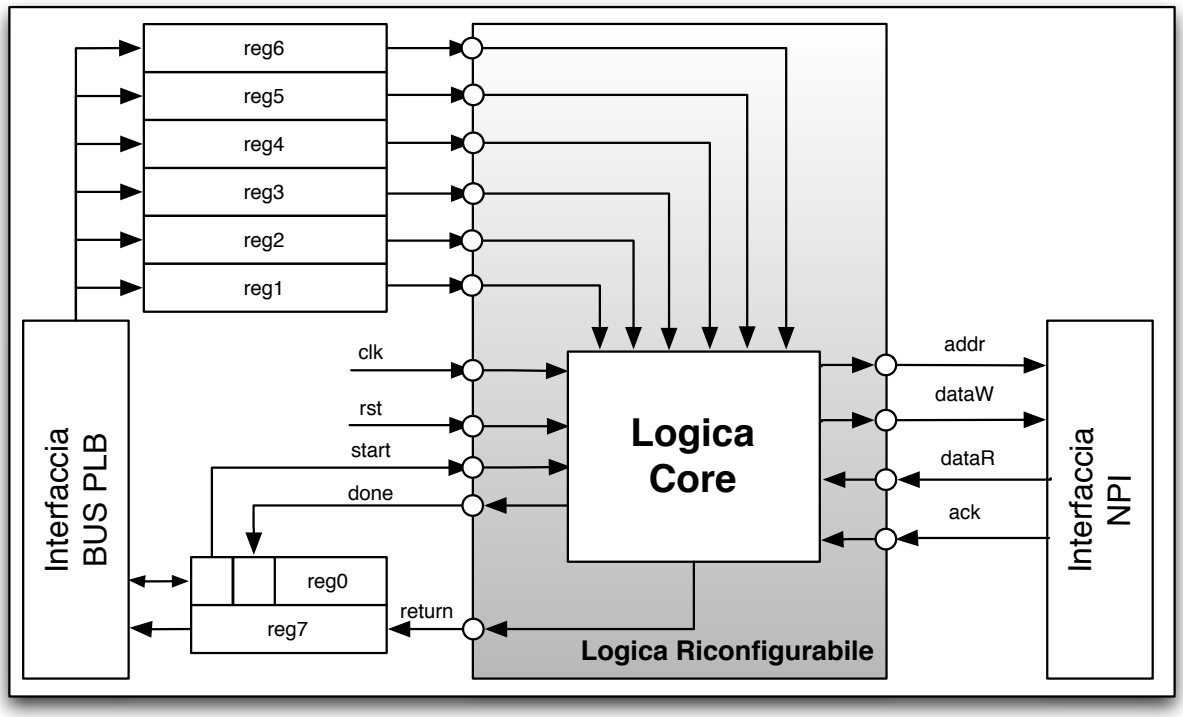

Figura 5.4: Struttura dei core hardware riconfigurabili.

Sezione 5.2 continua a rimanere valido. Il livello aggiuntivo che è stato introdotto per gestire la riconfigurazione inoltre è identico per tutte le aree riconfigurabili ed è indipendente dal core al suo interno; questo fa sì che in fase di creazione del core questo non debba essere creato e personalizzato in relazione al core stesso ma possa semplicemente essere riusato per tutte le aree riconfigurabili. Questo fatto quindi garantisce all'architettura il supporto alla riconfigura-

zione, ma non complica la fase di generazione dei core sia che essi siano generati a mano da parte dello sviluppatore, sia che essi vengano creati con uno strumento automatico così come è in questo lavoro.

La Figura 5.5 infine mostra l'architettura con supporto alla riconfigurazione con il massimo numero di aree riconfigurabili dovuto al limite introdotto dall'uso del core MPMC; come si vede dalla figura infatti, in questo caso a differenza dell'architettura statica ogni core hardware è collegato tramite l'interfaccia NPI alla memoria condivisa. Infine si può notare come il Microblaze che prima era destinato ad eseguire task dell'applicazione sia stato ora destinato alla gestione della riconfigurazione; questa è solo una scelta progettuale nello sviluppo dell'architettura e non un vincolo imposto ad esempio dal sistema di gestione a runtime per il quale non fa alcuna differenza se la riconfigurazione è gestita da un processore o da un core hardware dedicato.

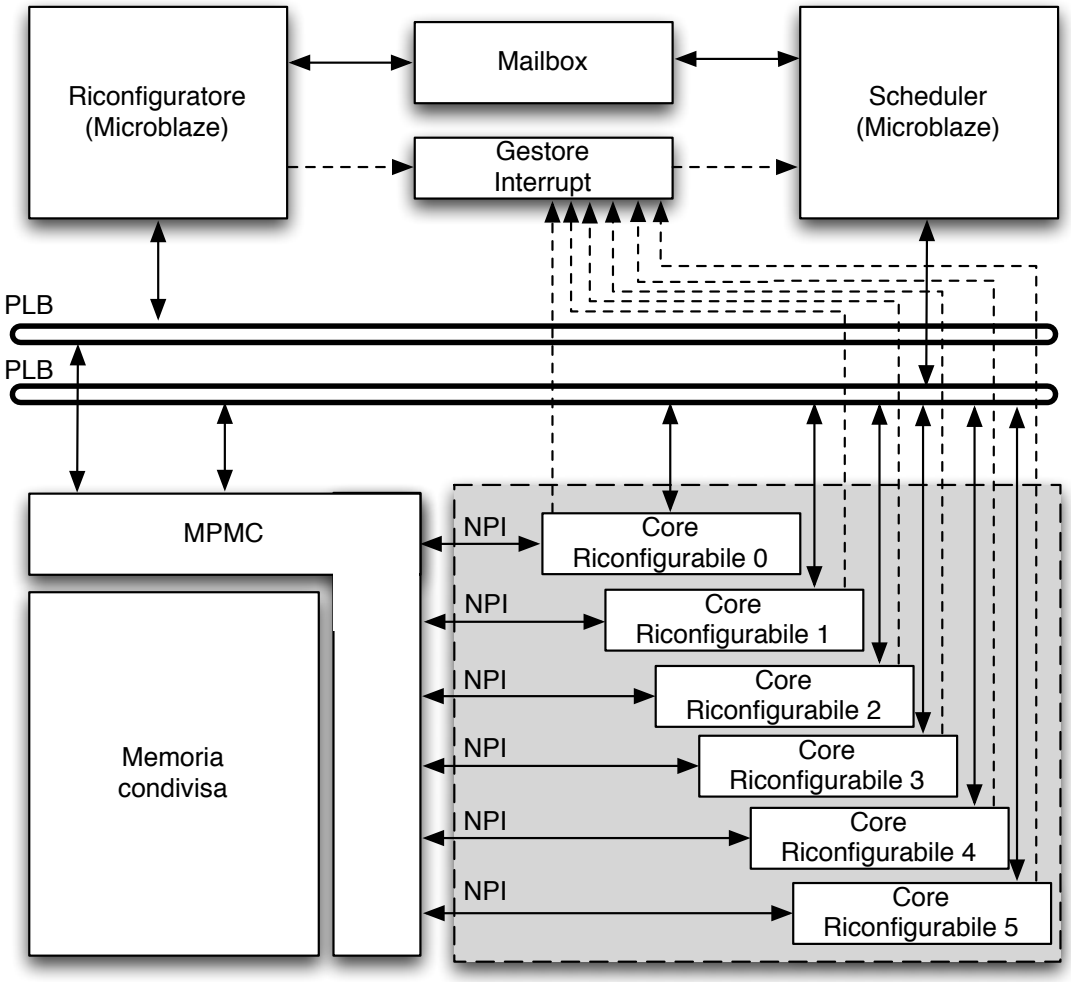

Figura 5.5: Schema architettura riconfigurabile.

5.5 Sommario

In questo capitolo si è illustrata l'architettura target utilizzata in questo lavoro. Dapprima è stata presentata l'architettura senza supporto per la riconfigurazione illustrando la struttura dei core presenti nell'architettura in questo caso e di come questi presentino un'interfaccia del tutto analoga a quella della funzione C che essi implementano. É stato poi illustrato come sia possibile estendere la struttura di questi core al fine di avere un'interfaccia standard indipendente dalla funzionalità che essi implementano al fine di supportare la riconfigurazione dinamica parziale; infine si è illustrato come questi core riconfigurabili sono integrati nel resto del sistema al fine di definire l'architettura target per questo quando il sistema finale necessita di supporto alla riconfigurazione.

Capitolo 6

Realizzazione del Framework

In questo capitolo vengono presentati i dettagli implementativi del flusso illustrando come i vari passi introdotti nel Capitolo 4 sono stato affrontati dal punto di vista implementativo al fine di fornire allo sviluppatore uno strumento conforme agli obiettivi dichiarati per questo lavoro. In particolare, si illustrano i diversi file di supporto usati dal flusso e le diverse strutture dati usate per la gestione delle informazioni. Nella Sezione 6.1 è descritto il formato dei file di input che rappresentano da un lato l'architettura di riferimento e dall'altro il formato di specifica delle direttive dello sviluppatore. Viene inoltre introdotto in questa sezione il formalismo usato per annotare il codice dell'applicazione al fine di esplicitare il parallelismo in essa presente e per definirne i task, il loro assegnamento e le loro specifiche in termini di tempo di esecuzione o riconfigurazione. La Sezione 6.2 fornisce i dettagli riguardanti la rappresentazione interna usata dal framework e su come le direttive fornite dallo sviluppatore, relativamente all'assegnamento dei core, vengono integrate in questa rappresentazione. La Sezione 6.3 fornisce dettagli sulla parte relativa alla creazione dei file sorgenti per i processori e la creazione del codice Hardware Description Language (HDL) per i core hardware dell'architettura. Sempre in questa sezione verranno forniti i dettagli implementativi della parte del flusso che si occupa della creazione del livello software per la gestione del sistema a runtime; ai dettagli implementativi relativi a questa è dedicato l'intero Capitolo 7 e quindi non verranno qui descritti. Infine la Sezione 6.4 fornisce i dettagli sull'ultima fase del flusso che si occupa dell'integrazione dei vari prodotti delle fasi precedenti in un unico sistema e della sua seguente sua trasformazione in uno o più bitstream pronti ad essere configurati su Field Programmable Gate Array (FPGA).

6.1 La preparazione degli input

Come descritto nel Capitolo 4 il flusso sviluppato ha bisogno di 3 input: l'applicazione da implementare su dispositivo riconfigurabile, la descrizione del modello dell'architettura target ed infine l'insieme delle direttive dello sviluppatore. Per la definizione del modello architetturale e delle direttive dello sviluppatore è stato utilizzato un file XML al fine di garantire la portabilità e la semplicità nella descrizione. Per la specifica del livello di parallelismo tra i task si è invece deciso di utilizzare un formalismo già largamente in uso per risolvere questa problematica, ovvero OpenMP [40]. Infine sebbene le specifiche relative all'assegnamento ed alle stime dei tempi di esecuzione dei task siano rappresentate con un file XML, si è deciso di introdurre la possibilità di inserire queste informazioni direttamente nel file sorgente dell'applicazione come annotazioni che saranno poi analizzate e trasformate, da uno strumento appositamente sviluppato, nella specifica XML richiesta in input.

6.1.1 La specifica XML

Il file XML in input al flusso contiene, come anticipato, due parti fondamentali; la prima si occupa di fornire un modello dell'architettura target verso la quale orientare il flusso automatico; mentre al seconda specifica tutte le proprietà delle applicazioni da integrare nel sistema riconfigurabile come ad esempio le scelte progettuali fatte dallo sviluppatore in termini di partizionamento dell'applicazione le statistiche (tempi di esecuzione e riconfigurazione) relative ai singoli task.

Il modello dell'architettura

La parte del file XML di input atto modellare l'architettura target del flusso contiene al suo interno tutte le caratteristiche architetturali necessarie per la creazione del sistema finale. In dettaglio l'XML contiene *tag* specifici per queste parti dell'architettura:

- Gli Ip-Core: rappresentano i componenti hardware configurati in fase di creazione del progetto EDK, rientrano in questo *tag* ad esempio il core MPMC, il core per il debug ed il core per la gestione della mailbox. Le informazioni incluse in questo *tag* sono:
  - il tipo di core;
  - lo spazio di indirizzamento;
  - l'interfaccia;

- Le memorie: rappresentano le memorie sia condivise che private presenti nell'architettura; contengono informazioni relative a:
  - il tipo di memoria;
  - frequenza di funzionamento;
  - lo spazio di indirzzamento;
  - l'interfaccia;
- Elementi di comunicazione: descrivono tutti gli elementi di comunicazione basati su bus presenti nell'architettura e contengono le seguenti informazioni:
  - il tipo di bus;
  - l'interfaccia
- Le unità processanti: contengono le informazioni relative a tutti gli elementi processanti in particolare ne vengono definiti tre tipi:
  - i processori software
    Vengono forniti in questo caso i dettagli relativi a:
    * il tipo di processore;
    * la frequenza di funzionamento;
    * l'interfaccia;
  - i coprocessori hardware statici
  - i coprocessori hardware riconfigurabili
    In questo caso vi è la possibilità di fornire le informazioni relative alle aree riconfigurabili quali la dimensione e la posizione.
- Le connessioni: definiscono i collegamenti tra i vari componenti, sia fisici, ovvero relativi alla singola porta dell'interfaccia, sia virtuali ovvero relativi alle connessioni tra due componenti dell'architettura. Nel primo caso vengono specificati le porte sorgente e destinazione e i dettagli relativi alla dimensione del collegamento, nel secondo i componenti sorgente e destinazione e una stima della latenza per il trasferimento.
- Il sistema: Il sistema è un contenitore che al suo interno può contenere tutti gli elementi specificati in precedenza e può a sua volta includere dei sistemi. In questo modo è possibile creare architetture complesse.

É infine importante notare che nel caso di un'architettura generata con il framework XPS di Xilinx è stato sviluppato un traduttore che genera il codice XML necessario al flusso proposto a partire dal codice XML generato dal framework Xilinx.

Le applicazioni

Come anticipato la seconda parte del file di specifica XML si occupa di definire le direttive dello sviluppatore relativamente alla specifica dei task presenti nell'architettura, il loro assegnamento alle unità processanti ed infine le stime riguardanti i tempi di esecuzione e riconfigurazione. Il formato di questa parte di XML è molto più semplice della precedente ed in particolare include i seguenti tag:

- Applicazione: definisce un'applicazione specificata dal suo nome identificativo. Contiene una lista di task;
- Task: specifica il singolo task della funzione identificato dal nome della funzione ad esso associata. Questo task contiene una lista di assegnamenti;
- Assegnamento: identifica che il task in questione deve essere associato all'unità processante specificata. Al suo interno contiene le informazioni sui tempi di esecuzione ed eventualmente il tempo di riconfigurazione necessari per l'esecuzione dell'implementazione associata al task;

6.1.2 Annotazione dei file sorgenti

Il secondo tipo di input che deve essere fornito al flusso sono i sorgenti delle applicazioni da implementare sull'architettura. Come introdotto nel Capitolo 4, lo sviluppatore deve annotare opportunamente il codice dell'applicazione al fine di esplicitare il parallelismo tra i vari task ed identificare i task. In particolare ogni funzione del codice identifica un task dell'applicazione. Il parallelismo tra i vari task è esplicitato mediante l'uso delle pragma OpenMP.

Infine come anticipato, per semplificare il lavoro allo sviluppatore, si è deciso di fornire ad esso la possibilità di definire tutte le direttive necessarie tramite opportune pragma all'interno del codice dell'applicazione. La scelta è ricaduta sulle pragma in quanto queste vengono normalmente ignorate in fase di compilazione dell'applicazione e quindi non alterano il comportamento del programma.

La descrizione del parallelismo

Per descrivere il parallelismo a livello dell'applicazione è stato usato OpenMP. Si è deciso di utilizzare questo formalismo per due motivi principali; innanzitutto tale formalismo è largamente accettato ed usato per esplicitare il parallelismo presente nelle applicazioni ed inoltre, essendo basato su direttive *pragma* ha il doppio vantaggio di essere inserito direttamente nel codice dell'applicazione e di venire ignorato in fase di compilazione se non diversamente specificato. Questa scelta riesce quindi a catturare tutte le necessità del framework in termini di esplicitazione del parallelismo senza gravare pesantemente sui compiti che lo sviluppatore deve svolgere. In particolare le direttive OpenMP utili per specificare il parallelismo a livello di applicazione sono riportate nel Listato 6.1. dove N nella *pragma omp parallel* rappresenta il

Listato 6.1 Pragma OpenMP per l'esplicitazione del parallelismo.

```
1 #pragma omp parallel sections num_thread(N)
2 #pragma omp section
```

numero di sezioni parallele nel frammento di codice annotato, mentre la *pragma omp section* specifica ognuna di queste sezioni.

Le direttive dello sviluppatore

Le direttive dello sviluppatore incluse nel file XML risultano scomode da utilizzare in questo modo in quanto questo formato è generalmente più verboso del necessario ed inoltre, essendo le direttive legate all'applicazione, sarebbe molto più produttivo riuscire ad includerle direttamente nel codice sorgente dell'applicazione stessa. Per semplificare questo passo si è deciso di fare uso ancora una volta delle direttive *pragma* che possono essere inserite dallo sviluppatore all'interno del codice dall'applicazione; sarà poi il framework a generare l'XML associato alle direttive specificate; come detto precedentemente inoltre tali annotazioni vengono ignorate dal compilatore non causando quindi problemi e comportamenti non voluti. Entrando nello specifico sono state introdotte *pragma* presenti nel Listato 6.2.

Le pragma così definite possono essere associate: al file sorgente che definisce all'applicazione (la prima), alla funzione del codice che rappresenta il task in oggetto (tutte le altre). Si fa presente che nel caso di creazione di un sistema senza il supporto alla riconfigurazione non c'è bisogno di creare N aree a cui assegnare gli N task presenti nell'applicazione, ma basta assegnarli tutti all'unità funzionale relativa alla logica riconfigurabile; avendo specificato che il

sistema da creare è un sistema senza il supporto alla riconfigurazione sarà il framework stesso a generare gli N core tra loro indipendenti.

Listato 6.2 Pragma per la definizione delle direttive dello sviluppatore

```
1 #pragma APP name=applicationName
2 #pragma TASK name=taskName
3 #pragma TASK mapping=P0,P1,...,Pn
4 #pragma TASK exTime=ET0,ET1,...,ETn
5 #pragma TASK recTime=RT0,RT1,...,RTn
```

6.2 L'analisi indipendente dall'architettura

In questa fase del flusso viene generata la rappresentazione interna dell'applicazione da analizzare partendo dal suo codice sorgente annotato con le *pragma* OpenMP come descritto nella sezione precedente. Oltre che la divisione in task in questo passo viene effettuata tutta una serie di analisi atte ad identificare quali sono i *basic block* del codice, quali le dipendenze dati e il flusso dell'applicazione; vengono infine identificati i loop viene generato il grafo in forma di *HTG* corrispondente. Tutte queste informazioni, che sono indipendenti dall'architettura di riferimento, verranno poi utilizzate nella fase di generazione dell'architettura ad esempio per identificare quali sono le variabili condivise tra diversi task, e che quindi devono essere allocate nella memoria condivisa, oppure quali sono le dipendenze tra i vari task al fine di gestire correttamente la fase di scheduling. Tutte le informazioni qui riportate sono codificate all'interno del framework facendo uso delle librerie *boost:graph*[76] e i differenti grafi generati sono visualizzabili dall'utente in formato *dot*[77].

Essendo questa parte già presente nel framework PandA precedentemente a questo lavoro, non se ne riportano qui i dettagli implementativi in quanto questi non rientrano in quanto svolto per questa tesi; qualora si volessero approfondire gli algoritmi e le tecniche usate per effettuare queste analisi si può far riferimento ad un qualsiasi testo relativo alla teoria dei compilatori, come ad esempio [22]. Risulta comunque utile mostrare a questo punto quali sono le informazioni che questa fase del flusso mette a disposizione alle fasi successive soprattutto in relazione al task graph dell'applicazione. Supponiamo ad esempio dover generare il sistema Multi Processors System on Chip (MPSoC), senza supporta alla riconfigurazione, che realizza l'applicazione riportata nel Listato 6.3; ogni funzione invocata corrisponderà ad un task del-

l'applicazione, avremo quindi un totale di 2 task nell'applicazione (*coreA* e *coreB*) più quelli dovuti al *main*.

Listato 6.3 Esempio di riferimento per il flusso di lavoro.

```
1  int main(int argc, char** argv)
2  {
3     int array[ARRAY_SIZE] = {4, 3, 6, 5, 4};
4     int a, b, c;
5
6  #pragma omp parallel sections num_threads(2)
7     {
8  #pragma omp section
9        {
10          a = coreA(array, ARRAY_SIZE, array[0]);
11       }
12 #pragma omp section
13       {
14          b = coreA(array, ARRAY_SIZE, array[1]);
15       }
16    }
17
18    c=coreB(a, b);
19
20    return c;
21 }
```

L'analisi del codice effettuata in questa fase genera come detto una serie di strutture dati dovute all'analisi del codice dell'applicazione; per le fasi successive del flusso, ed in particolare per la fase 3, è fondamentale il task graph generato durante questa analisi sulla base delle pragma OpenMP, che è rappresentato nella Figura 6.1. Come si vede da questa figura sono stati generati 4 task dovuti solamente alle pragma e non alle funzioni invocate nel codice, il primo task (Task0) è associato alla parte di codice precedente le pragma, i due task paralleli sono associati al codice annotato con le pragma ed infine l'ultimo task (Task1) è relativo al codice dopo di esse; in particolare questo task contiene la chiamata a *coreB*, la *return* e prima di esse vi è del codice che effettua l'assegnamento delle variabili *a*, *b* dopo i task precedenti; questi assegnamenti sono dovuti al fatto che il codice, prima di essere analizzato, è tradotto nella rappresentazione *Single Statement Assignement* dove quindi le variabili passate in ingresso a *coreB* non corrispondono più con quelle dichiarate all'inizio del programma.

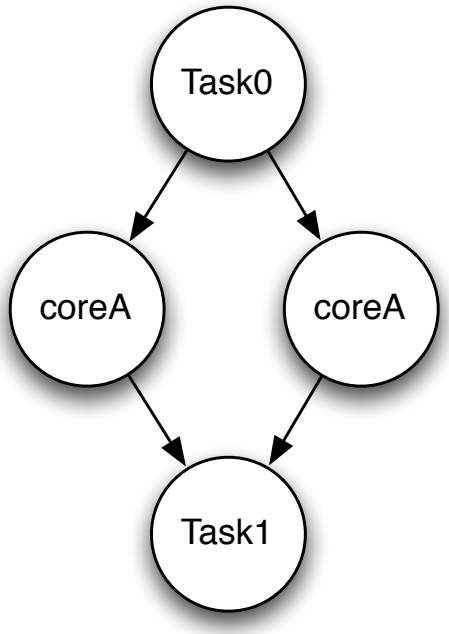

Figura 6.1: Task graph generato in base all'annotazione tramite pragma (in riferimento al Listato 6.3).

6.3 L'analisi dipendente dall'architettura

Nella terza fase del flusso vengono effettuate tutte le trasformazioni necessarie ad ottenere i sorgenti in linguaggio di alto livello ed in HDL per poi generare il sistema finale. In questa fase viene quindi analizzata l'applicazione in ingresso e, sfruttando le direttive specificate dallo sviluppatore, vengono generati le funzioni software, i core hardware, le loro interfacce ed i driver necessari alla comunicazione. Sempre in questa fase del flusso vengono create le strutture dati che servono da input al livello software per la gestione a runtime del sistema e viene decisa l'allocazione delle variabili in memoria in caso alcune di esse debba essere mappata su una memoria esterna condivisa.

In particolare in questa fase il flusso attraversa i seguenti passi: l'assegnamento dei task alle unità processanti secondo le direttive dello sviluppatore, la creazione dei sorgenti software, la creazione dei sorgenti hardware e delle relative interfacce, la creazione dei driver di comunicazione, l'analisi delle dipendenze dati ed allocazione delle variabili in memoria, ed infine la creazione delle strutture dati che permettono la gestione a runtime del sistema.

Assegnamento dei task alle unità processanti

A partire dal task graph generato nella fase precedente e dalle specifiche riguardanti l'assegnamento specificate nel file XML in ingresso, questo passo del flusso effettua una trasformazione del task graph originale sfruttando ed integrando al contempo le direttive dello sviluppatore. In particolare le trasformazioni necessarie sono volte in parte a risolvere dei limiti imposti dal risultato della fase precedente del flusso, in parte a garantire una corretta gestione del flusso di informazioni a runtime dato il modello di esecuzione del sistema specificato nel Capitolo 4.

Il limite della fase precedente sta nel fatto che ogni task è specificato da una funzione del codice ed la specifica del parallelismo non vieta di esplicitare come parallele tra loro due invocazioni della stessa funzione che operano su dati differenti. Seguendo il modello utilizzato precedentemente a questo lavoro nel framework tali funzioni verrebbero assegnate sulla stessa unità funzionale con l'evidente problema di doverne serializzare l'esecuzione, cosa evidentemente non desiderata dallo sviluppatore considerato l'uso fatto delle pragma. Il primo modo di ovviare a questo problema consiste nel lasciare questo compito a carico dello sviluppatore costringendolo a specificare due funzioni diverse per svolgere le stesse operazioni. Lo sviluppatore dovrebbe quindi creare all'interno del file sorgente due funzioni con diversa intestazione e stesso corpo. Tale soluzione sebbene efficace obbliga lo sviluppatore, in fase di scrittura dell'applicazione, ad effettuare operazioni poco intuitive e concettualmente inutili. La soluzione adottata consiste nell'effettuare una trasformazione del task graph creato alla fase precedente facendo in modo che, in caso di invocazioni della stessa funzione in parallelo, queste vengano mappate su due task differenti e quindi che vengano mappate su due unità funzionali diverse. Si fa presente che questa operazione viene effettuata solo in fase di creazione di un sistema statico; per un sistema riconfigurabile allo sviluppatore basta indicare che lo stesso task è mappato su più aree riconfigurabili, sarà poi il sistema di gestione a runtime a parallelizzarle quando possibile. Riprendendo l'esempio del Listato 6.3 quindi questa trasformazione del task graph influenza i nodi *coreA* in quanto questi contengono l'invocazione della medesima funzione; la trasformazione in oggetto è rappresentata nella Figura 6.2.

Il secondo limite nel task graph generato nella fase precedente è dovuto al fatto che la funzione *main* del codice, che corrisponde alla successione di operazioni che vengono eseguite in prima persona dallo scheduler, è generato in base alle sole direttive OpenMP ed è pensato per essere eseguito su un processore dedicato e che non deve gestire altro se non quella

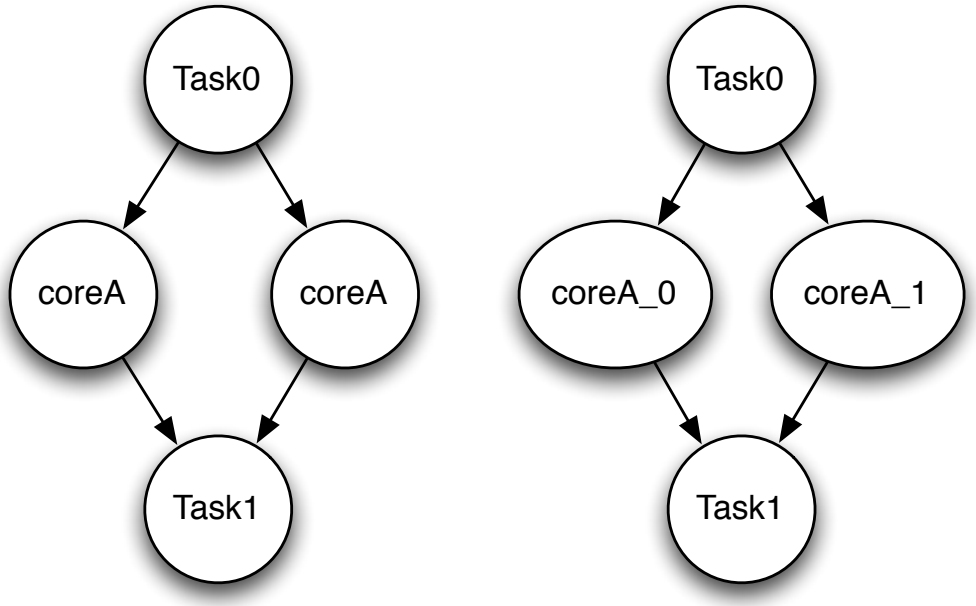

Figura 6.2: Prima trasformazione sul task graph. A sinistra il task graph iniziale, a destra il risultato (in riferimento al Listato 6.3).

applicazione. Questo pone un limite nel caso in cui più task assegnati a processori differenti vengano invocati in serie tra loro; in questo caso il task graph avrebbe un solo task per tutte le funzioni chiamate in serie e questo task sarebbe mappato su una sola unità funzionale cosa evidentemente sbagliata. Anche in questo caso la soluzione banale consiste nel lasciare che lo sviluppatore specifichi questa cosa tramite un uso contro intuitivo delle annotazioni OpenMP indicando come sezioni parallele con un solo task le chiamate alle singole funzioni. Considerato che il framework ha a disposizione le direttive dello sviluppatore relativamente all'assegnamento dei task alle varie unità funzionali questa trasformazione del task graph può essere fatta dal framework in modo automatico andando a spezzare in più task quei task, generati nella fase precedente, che contengono invocazioni di funzioni (e quindi task) che lo sviluppatore ha deciso di assegnare ad unità funzionali tra loro differenti. Riprendendo l'esempio del Listato 6.3 quindi questa trasformazione del task graph influenza il nodo *Task1* in quanto questo contiene alcune istruzioni della funzione *main* che verrà mappata sul Microblaze che esegue lo scheduler e l'invocazione di *coreB* che è invece eseguita da un'unità funzionale dedicata; la trasformazione in oggetto è rappresentata nella Figura 6.3.

L'ultima trasformazione necessaria è dovuta al modello dell'architettura che verrà generata. La gestione del flusso delle informazioni è effettuata dal processore che implementa lo scheduler e quindi questo è responsabile di inviare le informazioni necessarie all'esecuzio-

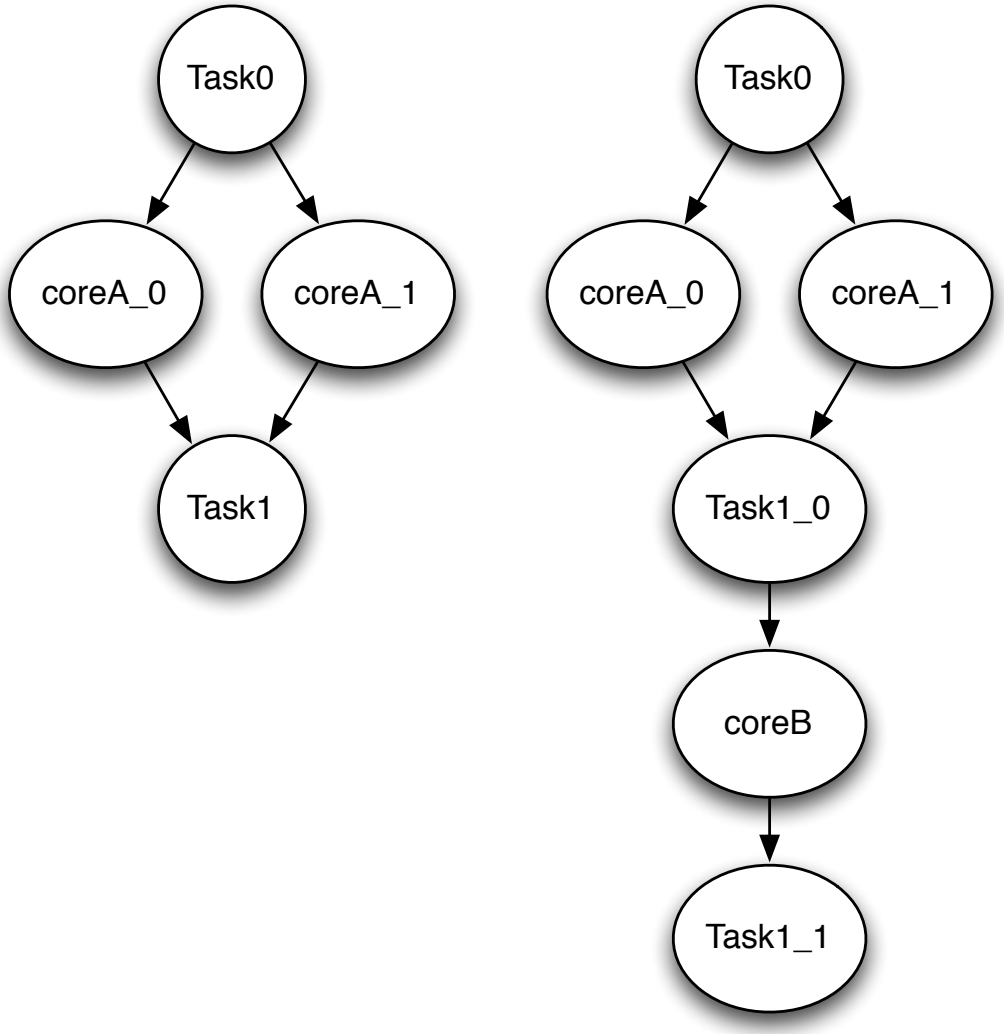

Figura 6.3: Seconda trasformazione sul task graph. A sinistra il task graph iniziale, a destra il risultato (in riferimento al Listato 6.3).

ne dei singoli task e ricevere le informazioni da esso loro generate, ovvero i valori di ritorno delle funzioni. Lo scheduler quindi è responsabile dell'esecuzione dei task, ma non li esegue in prima persona e quindi è necessario separare la fase di invocazione di un task da quella del recupero dei suoi risultati; per far questo ogni task che prevede un valore di ritorno viene spezzato in due task, uno corrispondente al passaggio dati iniziale ed all'invocazione della sua esecuzione, l'altro relativo al recupero del valore di ritorno. Sempre tenendo come riferimento il Listato 6.3 ed il risultato della trasformazione precedente, questa trasformazione si applica ai nodi *coreA_0*, *coreA_1* e *coreB* in quanto i task ad essi relativi hanno tutti un valore di ritorno da gestire. La trasformazione così effettuata ed il task graph associato sono rappresentati in Figura 6.4.

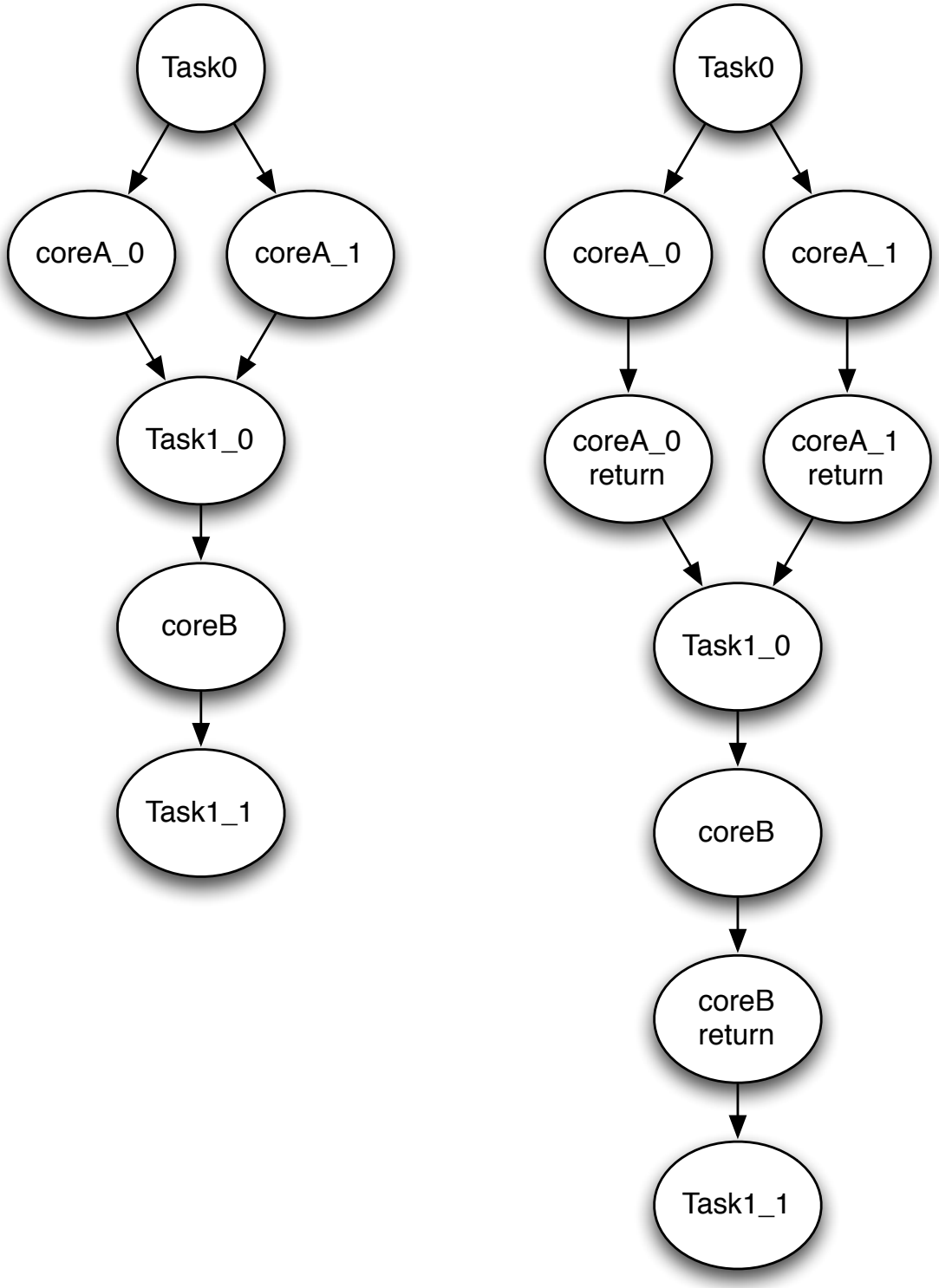

Figura 6.4: Terza trasformazione sul task graph. A sinistra il task graph iniziale, a destra il risultato (in riferimento al Listato 6.3).

Creazione dei sorgenti hardware e delle interfacce

In questo passo il codice dei task che necessitano di un'implementazione hardware vengono convertiti in codice HDL utilizzando un software per la sintesi di alto livello; in questo lavoro essendo in PandA presente uno strumento apposito, ovvero Bambu, si è utilizzata questa soluzione; si fa presente che è anche possibile effettuare la sintesi di alto livello con altri strumenti. Sempre a questo punto del flusso vengono poi generate le interfacce dei core per la comunicazione con il bus e con la memoria, quest'ultima solamente nel caso il core necessiti di scambiare dati con la memoria esterna. Nel caso della generazione di core statici, viene generato in questo step l'intero core che viene poi integrato nel sistema finale e quindi l'interfaccia di comunicazione con il bus dell'architettura. Al contrario nel caso di un sistema riconfigurabile viene generata solamente la parte che deve essere riconfigurata, così come descritto nella Sezione 5.4. Questo passo del flusso ed in particolare la creazione dell'interfaccia è dipendente dall'architettura anche nella sua implementazione, quindi in caso di estensione del flusso per il supporto ad un'altra architettura target parti di questo passo devono essere modificate.

Creazione dei sorgenti software

In questo passo vengono generati i sorgenti software da eseguire sui processore presenti nell'architettura finale. Per i task assegnati a questi processori viene generato il loro codice applicativo. In questa fase non sono necessarie particolari trasformazioni; il codice della funzione associata al task viene inserito nel file sorgente che deve essere compilato per un determinato processore dell'architettura.

Creazione dei driver

Questa fase genera i driver di comunicazione necessari per il passaggio di informazione tra lo scheduler ed i core hardware. I driver sono realizzati in modo automatico utilizzando l'interfaccia del core (che è generata in accordo con la funzione C a cui si riferisce) e le funzioni di libreria messe a disposizione per il Microblaze dalla Xilinx per il trasferimento dati su bus PLB. Inoltre si è deciso di inserire una struttura dati che astrae la struttura dati del core per fare in modo che lo scheduler quando necessita di eseguire un core, o una funzione implementata sull'altro microblaze effettui le medesime operazioni ovvero:

- passaggio dei parametri scrivendo nella struttura dati, che contiene gli attributi corrispondenti per numero e tipo ai parametri in ingresso al core e all'eventuale valore di ritorno;
- invocazione della funzione associata al core; questa funzione è responsabile del trasferimento dati utilizzando le funzioni di libreria;
- recupero del valore di ritorno leggendo dalla struttura dati

In questo modo ogni dettaglio implementativo dell'architettura è nascosto allo scheduler che può quindi rimanere invariato al variare dell'architettura sottostante. Si fa presente che la struttura dati è associata all'interfaccia del core collegato al bus PLB; questa risulta quindi personalizzata per ogni singolo core nel caso di architettura senza supporto alla riconfigurazione, mentre è standard per tutti i core nel caso di architettura riconfigurabile. Questo fatto ha costretto ad effettuare una scelta implementativa che grava sulla scrittura dell'applicazione da parte dello sviluppatore in quanto in questo secondo caso i parametri corrispondono a quelli della funzione solamente nel numero, mentre il tipo è per tutti *int*. Questo scelta fa in modo che si riesca a gestire i passaggio di parametri per i quali è possibile effettuare un *cast* a tipo di dato *int* senza perdere informazioni; sono quindi supportati i numeri interi ed i puntatori, ogni altro tipo di dato deve essere passato per riferimento nel codice dell'applicazione.

Allocazione delle variabili

In questo passo del flusso viene effettuata l'analisi delle variabili dell'applicazione al fine di identificare quelle condivise tra più task. Tutte le variabili che vengono passate per indirizzo e che sono accessibili da più task necessitano di essere allocate in una memoria che tutti i task condividono. Nel caso dell'architettura di riferimento per questo lavoro le variabili vengono allocate nell'unica memoria condivisa disponibile; tuttavia adattando il backend per una generica architettura si hanno a disposizione tutte le informazioni sulle memorie condivise dal modello architetturale in ingresso al flusso così da individuare in modo automatico la migliore memoria dove allocare le variabili. Questa fase viene realizzata analizzando il Data Flow Graph (DFG) dell'applicazione ed allocando per la variabile una zona di memoria pari alla sua dimensione. Tale procedura risulta attuabile considerando che una delle limitazioni classiche del codice sorgente per la sintesi di alto livello è la necessità di conoscere la dimensione di tutte le variabili in fase di compilazione e quindi l'attuazione di questo passo non impone alcun vincolo ulteriore al progettista.

6.4 Creazione MPSoC

In quest'ultima fase si procede ad effettuare l'integrazione dell'intero sistema e la sua successiva sintesi al fine di creare i bitstream necessari a configurare l'architettura. L'intera fase è gestita tramite script *bash* e *python* ed avviene esternamente al framework PandA. In particolare il funzionamento degli script, che non saranno riportati per non appesantire la trattazione, si differenzia in base al fatto che si stia creando un sistema statico oppure con supporto alla riconfigurazione.

Nel caso di un sistema statico la scrittura intera del codice, scheduler compreso, è creata dal *backend* implementato in PandA in quanto essendo PandA in grado di effettuare l'analisi di una singola applicazione esso è in grado di gestire al meglio questo caso. Gli script per l'integrazione si occupano solo di automatizzare le parti relative alla sintesi del sistema all'interno di *Xilixn Platform Studio* (XPS)[73] andando quindi a personalizzare il progetto di riferimento per l'architettura con le informazioni specifiche dell'applicazione quali i core hardware, le loro connessioni, i driver, i sorgenti per il Microblaze che esegue i task dell'applicazione ed infine i sorgenti dello scheduler. Una volta creato il progetto finale viene eseguita la sintesi in modo automatico ed il bitstream è disponibile per essere configurato sul dispositivo.

Nel caso di un sistema riconfigurabile invece PandA non è in grado di gestire l'intera scrittura del codice, e quindi il *backend* implementato si occupa di generare solamente i core hardware, mentre la restante parte del codice viene generata con delle annotazioni che guidano gli script nella creazione del codice finale; la parte non generata coincide esclusivamente con lo scheduler in quanto i driver e le interconnessioni sono fisse una volta definite il numero di aree riconfigurabili nell'architettura. Gli script a questo punto hanno a disposizione le informazioni relative a tutte le applicazioni che si vogliono implementare nel sistema e possono innanzitutto generare tutto il codice applicativo dello scheduler ed associarlo al Microblaze che è dedicato allo scheduling; tramite parametri in ingresso agli script è possibile decidere che algoritmo di gestione della riconfigurazione includere nello scheduler finale. Una volta creato lo scheduler si può passare all'automatizzazione della fase di sintesi del dispositivo sintetizzando da prima la parte statica tramite XPS e poi effettuando in modo automatico il flusso per la creazione dei bitstream riconfigurabili con PlanAhead [74]. Una volta terminato anche questo passo è possibile testare il tutto su dispositivo.

6.5 Sommario

In questo capitolo si sono mostrati i dettagli implementativi del framework realizzato mostrando come a partire da un'applicazione scritta in C sia possibile arrivare in modo automatico ai bitstream pronti per essere configurati sul dispositivo. In particolare si è mostrato come sia necessario effettuare trasformazioni sul task graph dell'applicazione per adattarlo ad una sua integrazione con uno scheduler per la gestione a runtime del MPSoC indipendentemente dal fatto che questo supporti o meno la riconfigurazione. Inoltre si è mostrato come nella parte software dell'architettura si sia disaccoppiato lo scheduler dalle funzioni che invocano i core, con la conseguenza quindi di disaccoppiare la fase di gestione a runtime dalla parte puramente architetturale; in questo modo in caso di modifica all'architettura sarà necessario solamente modificare le parti del flusso relative all'interfaccia dei core e ai relativi driver per il trasferimento dati, lasciando tutto il resto del *backend* immutato.

Capitolo 7

La gestione a runtime

In questo capitolo viene presentata l'implementazione del livello software atto a gestire il sistema a runtime. Il software che si occupa della gestione a runtime è stato modularizzato al fine di favorirne una generazione automatica; la parte di algoritimi di scheduling e partizionamento è indipendente dall'architettura e dall'applicazione, mentre vi è una parte di strutture di dati in ingresso che essendo dipendente da queste due necessita di essere generata ad ogni esecuzione del framework. Viene da prima presentata, nella Sezione 7.1, la fase del flusso che genera a partire dalle applicazioni in ingresso le strutture dati che vengono usate a runtime; si illustra inoltre come si è cercato di realizzare queste strutture dati al fine di ottimizzare e velocizzare la fase di scheduling essendo questo un task critico nel sistema. Nella Sezione 7.2 vengono poi presentati gli algoritmi per la gestione del sistema a runtime illustrando come viene gestita la fase di scheduling, la sincronizzazione dell'esecuzione dei vari core ed infine i tre algoritmi proposti per la gestione della riconfigurazione. Verrà quindi illustrato, come per altro già introdotto nel Capitolo 4, il *trade off* tra l'efficienza e la velocità degli algoritmi per la gestione della riconfigurazione proposti e di come siano stati implementati al fine di contenerne la complessità temporale.

7.1 Creazione delle strutture dati

Le strutture dati su cui si fonda lo scheduler sono quelle introdotte nel Capitolo 4; ovvero l'unità funzionale, l'applicazione, il task e l'implementazione. Si illustra ora come sono implementate le strutture dati ed in seguito come sia possibile generarle automaticamente a partire dalla rappresentazione dell'applicazione create nella fase 3 del flusso.

7.1.1 Le strutture dati

Tutte le strutture dati sono definite con delle *struct* del linguaggio C, e la seguente creazione del tipo di dato tramite la direttiva *typedef*; questo per semplificare e rendere più intuitivo il codice del sistema a fine di poterlo eventualmente manipolare nelle sue parti se necessario, ad esempio modificando l'algoritmo di scheduling per adattarlo ad una particolare architettura. Vengono ora illustrate nel dettaglio le implementazioni delle singole strutture dati mostrando le funzioni che vengono messe a disposizione per la loro manipolazione, il tutto sempre nell'ottica di un loro possibile utilizzo da parte ad esempio di uno sviluppatore che voglia testare eventuali sue politiche a runtime per il sistema.

Le unità funzionali

La definizione delle unità funzionali è riportata nel Listato 7.1, l'attributo *type* può assumere i valori *GPP,FPGASTATIC,FPGARECONF* a seconda della tipologia dell'unità funzionale, mentre lo stato assume i valori *BUSY,AVAILABLE,RECONFIGURING,MAPPED* che rappresentano rispettivamente i casi in cui l'unità è occupata, disponibile, in fase di riconfigurazione e quando gli è stato assegnato un task tra quelli ancora da eseguire; i valori della variabile stato non sono tra loro in mutua esclusione. Infine l'attributo *conf* identifica quale implementazione è attualmente configurata sul dispositivo nel caso questo sia un dispositivo riconfigurabile. I campi *startTime ed endTime* contengono i tempo a cui si è iniziato ad usare l'unità l'ultima volta e quando si prevede che questa termini l'esecuzione; questi campi sono usati in fase di assegnamento di un task ad un'unità funzionale per scegliere quella che secondo le stime permette di eseguire il task nel minor tempo possibile. Infine vi è la dichiarazione del tipo di dato e la funzione di inizializzazione.

Si fa presente che il core che gestisce la riconfigurazione è rappresentato anch'esso come un'istanza di questa struttura dati, ma è gestito separatamente rispetto alle altre unità funzionali. Le unità funzionali sono organizzate in una lista (array) di unità funzionali.

Listato 7.1 Struttura dati delle unità funzionali.

```
1  #define GPP 1
2  #define FPGASTATIC   2
3  #define FPGARECONF   3
4
5  #define BUSY   1
6  #define AVAILABLE 2
7  #define RECONFIGURING 4
8  #define MAPPED 8
9
10 struct processingElementStruct{
11    unsigned int id;
12    unsigned int type;
13    unsigned int status;
14    int conf;
15
16    unsigned int startTime;
17    unsigned int endTime;
18 };
19
20 typedef struct processingElementStruct processingElement;
21
22 void initProcessingElement(processingElement *PE, unsigned int id, unsigned int type,
       unsigned int status, int conf);
```

Le implementazioni

La definizione delle implementazioni è riportata nel Listato 7.2, l'attributo *type* identifica il tipo di implementazione in relazione al tipo di unità funzionale sulla quale deve essere eseguita e *bitstream* identifica il nome del bitstream da caricare in caso l'implementazione sia associata ad un'unità funzionale di tipo *FPGARECONF*. Infine l'ultimo attributo, *reconfigurations*, conta il numero di volte in cui l'implementazione prima di venire utilizzata ha necessitato di una riconfigurazione; questo dato come anticipato può essere usato per implementare politiche di riconfigurazione personalizzate al fine di ridurre ad esempio le riconfigurazioni per le implementazioni usate con più frequenza. Anche in questo caso si mette a disposizione il tipo di dato associato e la funzione di inizializzazione. Anche le implementazioni sono organizzate in una lista.

Listato 7.2 Struttura dati delle implementazioni.

```
1  struct implementationStruct{
2     unsigned int id;
3     unsigned int type;
4     char bitstream[50];
5
6     unsigned int reconfigurations;
7  };
8
9  typedef struct implementationStruct implementation;
10
11 void initImplementation(implementation *i,unsigned int PEid, unsigned int id,char *bitstream);
```

I task

I task, essendo l'oggetto principale da gestire, sono rappresentati da una struttura dati molto articolata e da un'altra struttura dati di supporto, nonché dalle consuete funzioni messe a disposizione per l'inizializzazione; tutto il codice relativo è riportato nel Listato 7.3.

Analizzando per prima la struttura dati *taskStruct* si vede come ogni task possa avere più implementazioni, al massimo una per ogni unità funzionale (riga 17); ad ognuna di queste implementazioni possono essere associate un massimo di due funzioni, la prima (riga 18) è quella che viene eseguita in fase di invocazione del task ed è responsabile del passaggio dei dati al task; la seconda (riga 19) è quella che si occupa di gestire il valore di ritorno della funzione introdotta nel Capitolo 6. Ad ogni task è infine associata una serie di stime così come

descritto nel Capitolo 4 rappresentate in un'apposita struttura dati *estimationStruct*. Ogni task, come introdotto nel Capitolo 4, può trovarsi in differenti stati, che sono codificati nella struttura dati attraverso l'uso delle *define* specificate nel Listato 7.3. Gli attributi *waitFor e mapping* sono utilizzati nella fase di scheduling; in particolare il primo memorizza il numero di predecessori che il task corrente deve attendere prima di passare dallo stato *WAITING* a *READY*, mentre il secondo rappresenta quale unità funzionale è stata assegnata all'esecuzione del task corrente.

Gli ultimi due attributi *recNum e uses* monitorano l'utilizzo del task in termini di numero di esecuzioni e numero di riconfigurazioni necessarie per utilizzarlo; come nel caso delle implementazioni questi parametri vengono messi a disposizione dello sviluppatore qualora questo voglia implementare politiche di scheduling e riconfigurazione personalizzate. Infine sono definite due funzioni una per l'inizializzazione della struttura dati *initTask* ed una per consentire di aggiungere un'implementazione ad un task *addImplementation*.

Il task graph

Il task graph è rappresentato come una matrice di incidenza $N \times N$ dove N è il numero totale dei task gestiti dal sistema. Per ogni elemento alla posizione (i, j) della matrice vi è un 1 se il task i deve attendere la terminazione del task j per essere eseguito. Questa struttura dati tuttavia non è usata direttamente per controllare quando un task è pronto ad essere eseguito, ma codifica solo la struttura statica del task graph senza registrare quando le dipendenze vengono risolte; l'informazione relativa a quando un task può essere eseguito viene memorizzata nello stato del task attraverso l'analisi dell'attributo *waitFor*. Ogni volta che un task termina se ne controllano le dipendenze nella matrice di incidenza e si aggiornano i valori dell'attributo *waitFor* per i task interessati e, qualora questo diventi pari a 0, il task passa nello stato *READY*. Questa scelta implementativa è stata dettata dal fatto che scorrere una matrice $N \times N$ per determinare se un task è pronto oppure no risulta poco efficiente computazionalmente e quindi inadatta per essere usata a runtime all'interno dello scheduler del sistema.

Le applicazioni

L'ultima struttura dati che rimane da presentare è quella relativa alle applicazioni che devono essere gestite dallo scheduler, tale struttura è riportata nel Listato 7.4. Come ci si può aspettare queste contengono l'elenco dei relativi task (*tasksId*), al massimo sono *MAXNUMTASKS*; il numero esatto per ogni applicazione è contenuto nella variabile *tasksNum*. Per ogni

Listato 7.3 Struttura dati dei task.

```
1  #define TOCONFIGURE 1
2  #define READY 2
3  #define RUN 4
4  #define RUNNING 8
5  #define WAITING 16
6  #define TOEXECUTE 32
7
8  struct estimationStruct{
9     unsigned int execTime;
10    unsigned int recTime;
11 };
12
13 typedef struct estimationStruct estimation;
14
15 struct taskStruct{
16    unsigned int id;
17    unsigned int implementations[PROCELEM];
18    void (*functions[PROCELEM])(void);
19    void (*returnFunctions[PROCELEM])(void);
20    estimation estimations[PROCELEM];
21    unsigned int status;
22    unsigned int waitFor;
23    unsigned int mapping;
24
25    unsigned int recNum;
26    unsigned int uses;
27 };
28
29 typedef struct taskStruct task;
30
31 void initTask(task *t, unsigned int id, unsigned int status);
32
33 void addImplementation(task *t, unsigned int PEid, unsigned int implId, void *function, void
       *returnFunction, estimation est);
```

task vengono specificate le condizioni sul suo stato e sul numero di task che deve attendere nel momento in cui l'applicazione viene eseguita rispettivamente in *initialWait* e *initialStatus*. Infine ogni applicazione può essere nello stato *WAITING* oppure *RUNNING* a seconda che sia attualmente in esecuzione nel sistema o meno.

Listato 7.4 Struttura dati delle applicazioni.

```
1  #define RUNNING 8
2  #define WAITING 16
3
4  struct applications{
5     char name[50];
6     unsigned int id;
7     unsigned int tasksNum;
8     unsigned int tasksId[MAXNUMTASKS];
9     unsigned int initialWait[MAXNUMTASKS];
10    unsigned int initialStatus[MAXNUMTASKS];
11    unsigned int status;
12 };
13
14 typedef struct applications application;
```

7.1.2 La generazione automatica

Generare automaticamente il livello software per la gestione a runtime consiste nel definire un insieme di *define* da includere in fase di compilazione del codice e nell'inizializzare le strutture dati qui sopra riportate in accordo con le specifiche delle applicazioni in esame; generando ed includendo nei file sorgenti le implementazioni delle funzioni che devono essere invocate dallo scheduler per eseguire i task e per recuperarne i valori di ritorno.

In particolare le definizioni da specificare sono le seguenti:

- TASKNUM: numero totale dei task gestiti dal sistema
- PROCELEM: numero totale delle unità funzionali del sistema
- IMPLNUM: numero totale delle implementazioni
- APPNUM: numero totale delle applicazioni
- MAXNUMTASKS: numero massimo di task in un'applicazione

Per quanto riguarda la generazione delle strutture dati e le funzioni per la loro invocazione il tutto è fatto traducendo il task graph che si ha alla fine della fase 3 nel flusso associando ad ogni task una funzione la quale utilizza i driver illustrati nel Capitolo 6. Si omettono qui i dettagli implementativi in quanto non significativi e si rimanda al Capitolo 8 dove verrà mostrato un esempio passo passo del funzionamento del flusso che illustra come questo passo venga effettuato. Quello che è importante sottolineare è che è possibile effettuare l'intera inizializzazione delle strutture dati tramite una singola funzione dello scheduler *initScheduler* che viene invocata in fase di avvio del sistema.

7.2 Gli algoritmi per la gestione a runtime

La fase di gestione a runtime del sistema si divide in 4 parti fondamentali; la prima è dedicata a ricevere le richieste per l'esecuzione di un'applicazione ed ad avviarne l'esecuzione; la seconda si occupa della fase di assegnamento dei task alle unità funzionali ed alla gestione della riconfigurazione, se necessario; la terza si occupa di effettuare lo scheduling delle applicazioni, ovvero esegue i task pronti; infine al quarta si occupa di aggiornare le strutture dati ed il task graph in relazione agli *interrupt* ricevuti dalle dall'architettura.

7.2.1 L'assegnamento dei task e la riconfigurazione

La parte che si occupa dell'assegnamento dei task alle unità funzionali ed alla gestione della riconfigurazione può essere realizzata con tre diversi algoritmi messi a disposizione dal framework e scelti dallo sviluppatore in base alle proprie necessita; tali algoritmi, come anticipato, differiscono per il costo computazionale ed la qualità della soluzione individuata.

First fit

Questo algoritmo è quello più efficiente in termini di costo computazionale, ma la soluzione individuata è nella maggioranza dei casi sub ottima in quanto la riduzione del numero di riconfigurazioni tramite il riuso dei core hardware già presenti nell'architettura avviene per puro caso e non è effetto di una ricerca da parte dell'algoritmo; tuttavia nel caso di un sistema con un elevato numero di task che hanno poche implementazioni in comune tra loro questo algoritmo potrebbe avere prestazioni pari ai due che verranno presentati di seguito. Ogni volta che viene invocato l'algoritmo controlla tra tutti i task presenti nel sistema quali sono quelli

pronti per essere eseguiti; per ognuno di questi cerca la prima unità funzionale libera per i quali essi hanno un'implementazione disponibile, nel caso sia configurata con l'implementazione richiesta avviene l'assegnamento; in caso contrario viene invocata la riconfigurazione qualora l'unità che gestisce la riconfigurazione sia libera. Il riuso dei componenti è dovuto esclusivamente al caso; l'algoritmo deve trovare come prima unità libera una che nella sua ultima esecuzione abbia eseguito un task con un'implementazione compatibile con il task attuale.

Best fit

L'algoritmo denominato Best fit ha un comportamento analogo al *first fit* illustrato in precedenza, ma invece che scegliere la prima unità disponibile itera su tutte le unità e sceglie quella che minimizza il tempo di esecuzione andando quindi a sfruttare a pieno il riuso dei componenti. Lo svantaggio è che se la complessità computazionale dell'algoritmo precedente, dati t task e p unità processanti, è solamente nel caso pessimo $O(tp)$, in questo caso invece $O(tp)$ è la complessità ad ogni iterazione. Il calcolo di quale unità funzionale minimizzi il tempo di esecuzione è effettuato usando la stima del tempo in cui l'unità funzionale in esame terminerà la computazione, contenuta nel parametro *endTime*, e conoscendo le statistiche per l'esecuzione del task su quella unità, quindi i parametri *exTime*, *recTime*.

Sebbene la soluzione finale ottenibile con questo metodo migliora quella del caso precedente, questo metodo rischia di essere soggetto a minimi locali in quanto la scelta della soluzione è dipendente dall'ordine in cui vengono analizzati i task.

Best fit con prefetch (v2)

Quest'ultima versione dell'algoritmo migliora la soluzione al problema dell'assegnamento rispetto al caso precedente cercando di effettuare le riconfigurazioni in anticipo rispetto a quando il task è pronto. Per fare questo sfrutta gli attributi degli oggetti *task* ed *applicazione* che indicano quali sono i task che devono ancora essere eseguiti. Questa soluzione porta generalmente dei vantaggi nella soluzione finale tuttavia vi sono due controindicazioni. La prima consiste nel fatto che ora l'algoritmo non deve controllare solo i task pronti, ma tutti quelli che devono ancora essere eseguiti nel sistema, il che ne aumenta la complessità computazionale rispetto al caso precedente. La seconda sta nel fatto che andando a predicare su task non ancora pronti l'algoritmo deve poter tornare sui suoi passi qualora la scelta fatta nei confronti di

un task non ancora pronto possa causare una situazione di *deadlock*; anche questa situazione grava sul costo computazionale dell'algoritmo.

Quest'ultima versione è soggetta tuttavia ancora ad un problema di minimi locali dovuti al fatto che potrebbe rivelarsi opportuno evitare di effettuare un assegnamento ad un dato punto dell'esecuzione per rimandarlo più in avanti; tuttavia questa scelta presuppone la conoscenza almeno parziale delle condizioni di lavoro del sistema, quale ad esempio la frequenza di arrivo dei task, che in questo lavoro si è supposto tuttavia di non conoscere.

7.2.2 Lo scheduling

L'algoritmo di scheduling proposto per l'implementazione è semplice e non prende decisioni ad esempio riguardo la convenienza a rinviare in futuro l'esecuzione di un task; queste decisioni quando possibile vengono prese in fase di assegnamento del task all'unità funzionale, come illustrato al passo precedente.

L'algoritmo di scheduling è rappresentato dallo pseudo-codice rappresentato nel Listato 7.5; ad ogni passo controlla per tutti i task pronti ad essere eseguiti se sono da configurare o meno; nel caso siano già configurati e gli sia già stata assegnata un'unità funzionale l'algoritmo controlla che questa sia libera e che la sua implementazione corrente sia valida per il task (in caso contrario si deve attendere che venga riconfigurata); se queste condizioni sono rispettate il task viene eseguito invocando la funzione relativa all'implementazione scelta e vengono aggiornati lo stato dell'unità funzionale (impostato al valore *BUSY*) e del task (che diventa ora *RUNNING*), il numero di esecuzioni del task incrementando il campo *uses* ed infine aggiornati i valori di inizio e di fine utilizzo utilizzando come valore il tempo corrente e poi incrementandolo con le stime associate all'implementazione usata del task.

7.2.3 La gestione degli interrupt

Il livello software per la gestione del sistema si integra con le routine di gestione degli interrupt del sistema mettendo a disposizione due variabili condivise *reconfInterrupt* e *interrupts*. La prima deve essere impostata dalla routine di gestione dell'interrupt generato dal modulo che gestisce la riconfigurazione. La seconda invece assume tutti i $2^{32} - 1$ valori possibili ed ogni bit rappresenta il fatto che l'unità funzionale all'indice corrispondente abbia o meno un interrupt da gestire. In questo modo il livello software risulta disaccoppiato dall'architettura sottostante e quindi nel caso di un cambio dell'architettura, purché la parte architetturale comunichi con il

software con le stesse modalità, questo non deve essere modificato; questo significa che anche nel caso non si vogliano usare gli interrupt, ma un qualche altro metodo di sincronizzazione, basta rispettare il modo in cui vengono aggiornate le due variabili appena descritte. Ogni volta che c'è un evento da gestire, nel caso discusso in questa tesi un interrupt, il livello di gestione software effettua i passi descritti nello pseudo-codice riportato nel Listato 7.6.

Nel caso l'interrupt sia relativo al gestore della riconfigurazione viene invocata la funzione per gestire questo evento che:

- imposta lo stato dell'unità di riconfigurazione ad *AVAILABLE*;
- imposta ad *AVAILABLE e MAPPED* lo stato dell'unità funzionale che era in fase di riconfigurazione;
- aggiorna lo stato del task riconfigurato rimuovendo *TOCONFIGURE* dal rispettivo attributo di stato e aggiornando il campo relativo al suo assegnamento associandolo all'unità

Listato 7.5 Pseudo codice dello scheduler

```
1:  procedure SCHEDULE
2:      for all t ∈ listOfTasks do
3:          if (t → status & READY) ∧ ¬(t → status & TOCONFIGURE) then
4:              ProcessingElement p = t → mapping
5:              bool pAvailable = p → status & AVAILABLE
6:              bool rightTask = p → configuredTask == t
7:              if (pAvailable) ∧ rightTask then
8:                  t → execute()
9:                  t → status | = RUNNING
10:                 t → status & = ¬READY
11:                 t → uses + +
12:                 p → status = BUSY | MAPPED
13:                 p → startTime = now
14:                 p → endTime = now + t → execTime
15:             end if
16:         end if
17:     end for
18: end procedure
```

sulla quale è stato configurato;

Nel caso di interrupt di una delle unità funzionali atte all'esecuzione dei task invece i passi seguiti dall'algoritmo sono i seguenti:

- identifica l'unità funzionale che ha generato l'interrupt;
- esegue la funzione necessaria per la gestione del valore di ritorno del task se necessario;
- imposta lo stato dell'unità funzionale a *AVAILABLE* e quello del task a *RUN*;
- scorre il task graph e diminuisce il valore dell'attributo *waitFor* per i task che dipendevano da quello appena terminato; se *waitFor* raggiunge il valore 0, il task relativo passa dallo stato *WAITING* a *RUNNING*;

7.3 Sommario

In questo capitolo si è presentata la struttura del livello software per la gestione del sistema a runtime. Si sono descritte innanzitutto le strutture dati che lo compongono illustrandone le proprietà in relazione gli obiettivi che sono stati posti nella descrizione del lavoro; si è quindi illustrato come sia possibile generare automaticamente il livello software dal codice dell'applicazione. Sono stati poi presentati i tre algoritmi che il flusso mette a disposizione allo sviluppatore per gestire la riconfigurazione e l'assegnamento dei task ai core in fase d'esecuzione e si è infine illustrato come avviene lo scheduling e l'esecuzione dei task e come il livello software interagisca con il resto del sistema attraverso la gestione degli interrupt.

Listato 7.6 Pseudo codice per la gestione degli interrupt

```
1: procedure INTERRUPTHANDLER
2:     if reconfigurationInterrupt ≠ 0 then
3:         ProcessingElement r = RECONFIGURATOR
4:         r → status &= ¬RECONFIGURING
5:         r → status &= ¬BUSY
6:         r → status ||= AVAILABLE
7:         ProcessingElement p = listOfProcessingElements[reconfigurationInterrupt−
   1]
8:         reconfigurationInterrupt = 0
9:         Task t = r → configuredTask
10:        p → status = AVAILABLE | MAPPED
11:        p → configuredTask = t
12:        t → status &= ¬TOCONFIGURE
13:        t → mapping = p
14:    end if
15:    if interrupts ≠ 0 then
16:        for i = 0 → numberOfProcessingElements do
17:            if interrupts & (1 << i) then
18:                interrupts &= ¬(1 << i)
19:                ProcessingElement p = listOfProcessingElements[i]
20:                Task t = p → configuredTask
21:                if t → hasReturnValue then
22:                    t → getRerutnValue()
23:                end if
24:                p → status = AVAILABLE
25:                t → status &= ¬RUNNING
26:                t → status ||= ¬RUNNING
27:                updateTaskGraph(t)
28:            end if
29:        end for
30:    end if
31: end procedure
```

Capitolo 8

Risultati

Questo capitolo illustra i risultati finali del lavoro descrivendo una serie di prove sperimentali volte a validare il framework descritto in questa tesi. Tutte gli esperimenti mostrati in questo capitolo sono stati effettuati utilizzando il flusso Xilinx 12.4 e la piattaforma di sviluppo Xilinx XUPV5-LX110T. La Sezione 8.1 fornisce una breve introduzione e descrizione alle applicazioni utilizzate nei casi di studio illustrati in questo capitolo. La Sezione 8.2 mostra un esempio passo passo di tutto il flusso relativamente ai due casi di studio e mostra in particolare i dettagli riguardanti le trasformazioni sul task graph e l'esplicitazione del parallelismo attraverso l'uso delle pragma; nell'illustrare queste trasformazioni la sezione mostra anche come il framework proposto possa servire da supporto per la fase di esplorazione dello spazio delle soluzioni nel caso di un sistema statico. La Sezione 8.3 mostra invece il flusso di lavoro per la creazione di un sistema riconfigurabile, mostrando come i diversi algoritmi di gestione a runtime si comportino e influenzino l'esecuzione del sistema.

8.1 Le applicazioni

Per testare le funzionalità mese a disposizione dal framework sono stati utilizzati quattro differenti algoritmi. Il primo che verrà presentato è un semplice algoritmo che effettua operazioni matematiche su di un vettore di interi ed è utilizzato solamente al fine di mostrare il flusso di lavoro del framework. Gli altri tre invece sono algoritmi di analisi d'immagini e sono quindi più rilevanti dal punto di vista applicativo.

8.1.1 Applicazione di test

L'algoritmo usato per la validazione del framework è un'applicazione che manipola i dati di un array, il codice è riportato nel Listato 8.1.

Listato 8.1 Algoritmo di test.

```
1  #define ARRAY_SIZE 5
2
3  int coreA(int *array, int size);
4  void coreB(int *array, int size, int* max);
5  int coreC(int a, int b);
6  void coreD(int a, int b, int *c);
7  int coreE(int *a);
8
9  int main(int argc, char** argv)
10 {
11    int array[ARRAY_SIZE] = {4, 3, 6, 5, 4};
12    int a, b, c, d, e;
13
14    a = coreA(array, ARRAY_SIZE);
15    coreB(array, ARRAY_SIZE, &b);
16    c=coreC(a, b);
17    coreD(a, b, &d);
18    e=coreE(&d);
19
20    return c;
21 }
```

Nel listato sono state omesse le implementazioni; per completezza si fa presente che:

- coreA: ritorna il minimo valore presente nell'array
- coreB: scrive nella zona di memoria referenziata dal parametro *max* il massimo dell'array

- coreC: ritorna la somma di a e b
- coreD: scrive nella zona di memoria referenziata da c la somma di a e b
- coreE: ritorna il valore referenziato da a incrementato di 1

Questa applicazione è stata predisposta in quanto permette di testare tutte le trasformazioni sul task graph illustrate nei capitoli precedenti e utilizzando variabili passate per riferimento consente anche di testare la correttezza dell'allocazione delle variabili in memoria. I dettagli relativi a come quest'applicazione è stata portata sull'architettura statica presentata nella Sezione 5.1 sono presentati nella Sezione 8.2

8.1.2 Filtro di Laplace

Il filtro di Laplace è un filtro spaziale di convoluzione per la rilevazione dei contorni in un'immagine [78]; l'effetto del filtro è quello di approssimare la derivata seconda dell'intensità di colore dei pixel di un'immagine nella direzione verticale ed orizzontale al fine di identificare i punti dove l'intensità cambia in modo più repentino, punti che coincidono appunto con il contorno degli oggetti presenti nell'immagine.

Come la maggior parte dei filtri spaziali il filtro di Laplace è un operatore puntuale; con questo termine si indica il fatto che il risultato del filtro dipende solo dal pixel preso in esame e non dalla storia passata; il risultato di un filtro di convoluzione puntuale viene calcolato effettuando la convoluzione tra la matrice del filtro, chiamata kernel, e una matrice di pixel presi dalla zona nella vicinanza del pixel in esame; il procedimento è rappresentato in Figura 8.1 ed la matrice K riportata è la matrice del filtro di Laplace. Data la natura dei filtri puntuali ne consegue che essi sono altamente parallelizzabili in quanto possono essere calcolati su una sottoparte dell'immagine per poi unire i risultati ottenuti nell'immagine finale. Questa proprietà è stata utilizzata per parallelizzare i filtri di elaborazione di immagini utilizzati in questo capitolo. Si fa presente inoltre che generalmente questi algoritmi vengono utilizzati su immagini in scala di grigi e che quindi è necessario una conversione in scala di grigi prima di iniziare l'elaborazione.

8.1.3 Rilevazione dei contorni (Canny)

Il filtro di Canny [78] è, come il precedente, un filtro di rilevazione dei contorni, ma l'ideatore del filtro ha pensato che operare direttamente sull'immagine originale poteva dare

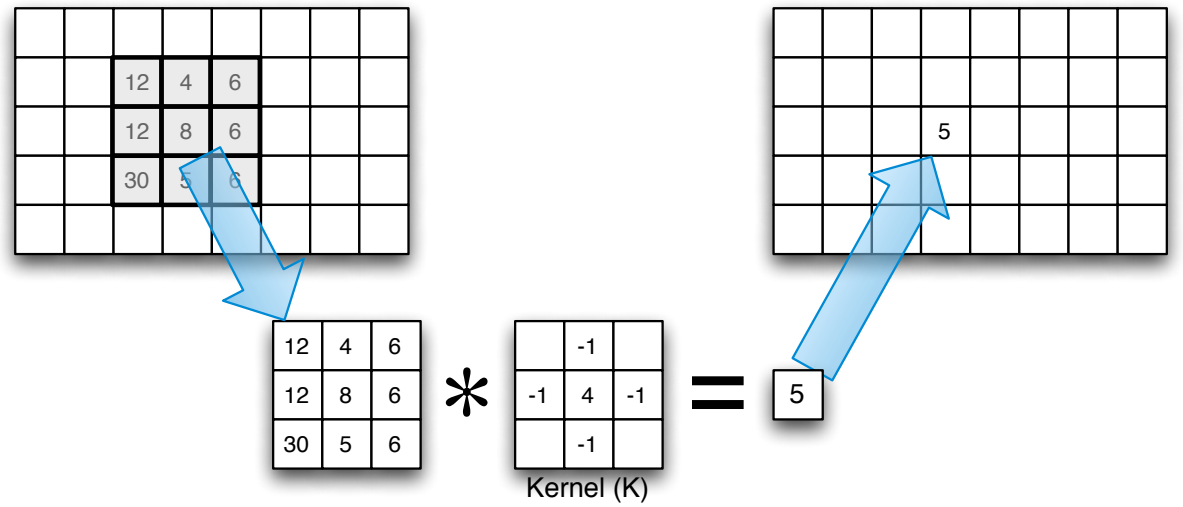

Figura 8.1: Filtro di convoluzione per immagini.

risultati non ottimali per via dell'eventuale rumore presente nell'immagine, in quanto questo sarebbe stato rilevato come contorno. La sua idea è stata quindi quella di introdurre un filtro per la rimozione del rumore (filtro di media, oppure gaussiano) a monte del filtro per l'individuazione dei contorni (nell'algoritmo originale era il filtro di Sobel [78], in questo lavoro si è utilizzato il filtro di Laplace); a valle di questo viene inoltre inserito un filtro di soglia che elimina dall'immagine in uscita quei pixel il cui valore non indica chiaramente se si tratti di un bordo dell'immagine originale oppure no.

La struttura a blocchi del filtro è rappresentata in Figura 8.2; una trattazione più dettagliata ed un'implementazione di questo filtro su FPGA è riportata in [79]. Come per il filtro precedente si fa presente che anche questo filtro nel complesso è un filtro puntuale, ovvero ogni singolo blocco di Figura 8.2 può essere effettuato su parti di immagine diverse per poi unire i risultati così ottenuti nel risultato finale.

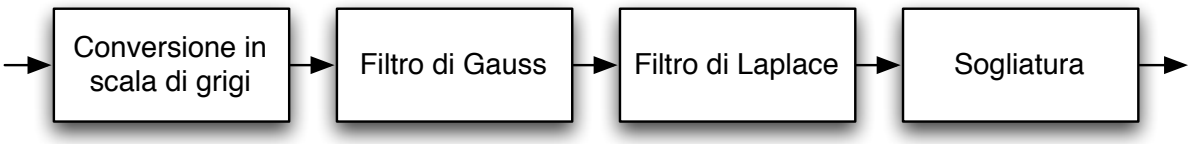

Figura 8.2: Diagramma a blocchi del filtro di Canny per il rilevamento dei contorni in un'immagine.

8.1.4 Rilevazione del movimento

L'algoritmo per la rilevazione del movimento implementato è una versione semplificata degli algoritmi di analisi del movimento nell'immagine e sfrutta la tecnica di rimozione dello sfondo [80]. Un'applicazione di questa tecnica si può avere ad esempio nelle telecamere di sorveglianza che mantengono fissa l'inquadratura; la telecamera, appena accesa, può acquisire l'immagine dell'ambiente, che si presuppone nello stato di quiete, e sottrarre questa immagine a tutte quelle successivamente catturate al fine di identificare movimenti nell'ambiente e segnalare opportunamente la cosa piuttosto che analizzare più a fondo le parti dell'immagine in questione. Data la bassa qualità che in genere si ha nelle immagini catturate da queste periferiche è opportuno far precedere al passo di rimozione dello sfondo un filtro di riduzione del rumore e farlo seguire da una fase di decimazione dei punti rilevati tramite confronto con un'opportuna soglia; l'applicazione appena definita è analoga a quella del caso di studio precedente dove il blocco corrispondente al filtro di Laplace è stato sostituito con il blocco che effettua la rimozione dello sfondo; il diagramma a blocchi dell'applicazione è raffigurato in Figura 8.3. É importante sottolineare che l'algoritmo di sottrazione dello sfondo è anch'esso un filtro puntuale e quindi parallelizzabile allo stesso modo degli algoritmi sopra definiti.

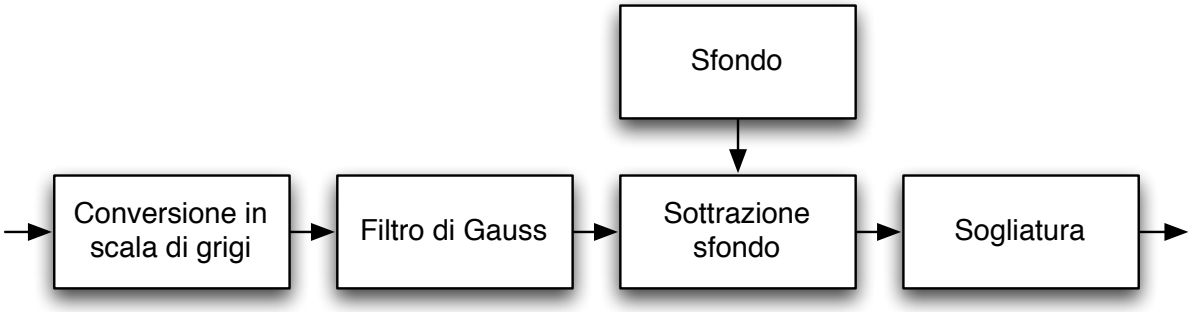

Figura 8.3: Diagramma a blocchi per il filtro di rilevamento del movimento.

8.2 Creazione di un sistema statico ed esplorazione dello spazio delle soluzioni

Questa sezione presenta due casi di test al fine di illustrare le diverse modalità d'uso dello strumento in relazione alla creazione di un sistema hardware non riconfigurabile. Il primo esempio si riferisce al caso in cui il framework venga usato solamente per implementare un'applicazione su dispositivo hardware; con questo caso di studio si vuole inoltre mostra-

re l'effettivo funzionamento di quanto proposto in questo lavoro illustrando nei particolari i passi affrontati dal framework. Il secondo caso di studio presentato in questa sezione è invece relativo al caso in cui lo strumento proposto venga utilizzato al fine di compiere una fase di analisi dello spazio delle soluzioni relativamente alle diverse implementazioni hardware dello stesso algoritmo. Per l'esecuzione di un'applicazione senza sfruttare la riconfigurazione, l'architettura di riferimento è quella riportata nella Sezione 5.1.

8.2.1 Verifica del framework

Per il caso di studio volto a verificare le funzionalità del framework relativamente alla capacità di generare un sistema statico effettivamente rispondente alle specifiche del progettista si è deciso di utilizzare l'applicazione di test illustrata nella Sezione 8.1.

Per prima cosa l'applicazione va arricchita esplicitando il parallelismo utilizzando le pragma OpenMP così come indicato nella Sezione 6.1. Analizzando il codice dell'applicazione riportato nel Listato 8.1 risulta evidente che le funzioni *coreA* e *coreB* possono essere eseguite tra loro in parallelo, mentre *coreC*,*coreD* e *coreE* devono essere eseguite in serie tra loro, avendo dipendenze dati, e devono essere eseguite solamente dopo la terminazione di *coreA* e *coreB*. Le informazioni sul mapping desiderato per i singoli task sono riportate invece prima della funzione alla quale si riferiscono, ed all'inizio del file per la pragma relativa all'applicazione; le informazioni relativamente ai tempi di esecuzione e riconfigurazione possono essere omessi in quanto non rilevanti per la gestione del sistema a runtime nel caso di un sistema che non fa uso di riconfigurazione.

Il Listato 8.2 riporta il codice del *main* annotato per esplicitare il livello di parallelismo voluto e per indicare l'assegnamento dei task; in particolare si noti l'uso delle pragma OpenMP (righe da 30 a 40). Una volta annotato il codice è possibile eseguire il flusso automatico che porta alla creazione del sistema sfruttando il modello XML dell'architettura proposto in questo lavoro; tale modello essendo indipendente dall'applicazione non necessita di modifiche. Il primo passo del flusso analizza le pragma OpenMP e crea il task graph in relazione ad esse; questo task graph iniziale, corrispondente con l'uscita della Fase 2 del flusso (l'analisi indipendente dall'architettura), è riportato nella Figura 8.4.

A questo punto, la rappresentazione in forma di task graph mostrata entra nella Fase 3 del flusso che si occupa di effettuare le trasformazioni dipendenti dall'architettura. In primo luogo si può notare come l'ultimo task esegua le funzioni *coreC*, *coreD* e *coreE*; questo caso fa

Listato 8.2 Algoritmo di test con annotazioni.

```
1  #pragma APP name=coreABCDE
2
3  #define ARRAY_SIZE 5
4
5  #pragma TASK name=coreA
6  #pragma TASK mapping=staticArea
7  int coreA(int *array, int size);
8
9  #pragma TASK name=coreB
10 #pragma TASK mapping=microblaze0
11 void coreB(int *array, int size, int* max);
12
13 #pragma TASK name=coreC
14 #pragma TASK mapping=staticArea
15 int coreC(int a, int b);
16
17 #pragma TASK name=coreD
18 #pragma TASK mapping=staticArea
19 void coreD(int a, int b, int *c);
20
21 #pragma TASK name=coreE
22 #pragma TASK mapping=staticArea
23 int coreE(int *a);
24
25 int main(int argc, char** argv)
26 {
27    int array[ARRAY_SIZE] = {4, 3, 6, 5, 4};
28    int a, b, c, d, e;
29
30 #pragma omp parallel sections num_threads(2)
31    {
32 #pragma omp section
33       {
34          a = coreA(array, ARRAY_SIZE);
35       }
36 #pragma omp section
37       {
38          coreB(array, ARRAY_SIZE, &b);
39       }
40    }
41
42    c=coreC(a, b);
43    coreD(a, b, &d);
44    e=coreE(&d);
45
46    return c;
47 }
```

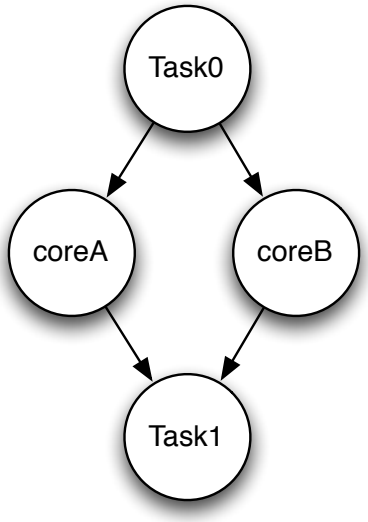

Figura 8.4: Task graph generato dalla fase 2 del flusso relativamente al Listato 8.2.

riferimento alla seconda delle trasformazioni necessarie illustrate nel Capitolo 6. L'invocazione delle tre funzioni e quindi l'esecuzione dei tre task deve essere separata avendo un task differente per ogni funzione; applicando questa trasformazione si ottiene quindi il task graph illustrato in Figura 8.5 al centro. Analizzando ora il task graph appena generato si può notare come le funzioni che hanno un valore di ritorno (*coreA*, *coreC* e *coreE*) non presentino un task apposito per il suo recupero dal core che le implementerà. Questo caso fa riferimento alla terza delle trasformazioni necessarie ed applicandola si ottiene come risultato il task graph mostrato nella Figura 8.5 sulla destra. Il task graph così rappresentato rispetta le specifiche richieste per procedere con il resto del flusso ed è quindi possibile procedere con la generazione del sistema finale utilizzando le informazione relative all'assegnamento dei task alle unità funzionali indicato dal progettista; queste informazioni vengono integrate nel task graph e questo passo della Fase 3 del flusso si conclude con un task graph della forma di quello rappresentato in Figura 8.5 sulla destra. Si noti come da un task graph iniziale da 4 task si sia giunti ad un task graph composto da 11 task.

A questo punto avviene la fase di generazione del codice sia per i task che andranno implementate in software, sia per quelle che si andranno ad implementare in hardware. Al termine di questa fase si avranno a disposizione i sorgenti software per il microblaze0, nel caso in oggetto solo a funzione *coreB*, ed i core hardware con la loro interfaccia da integrare nel sistema. In particolare si fa presente che solo le interfacce necessarie vengono generate per i vari core, e quindi i core che non necessitano di effettuare operazioni nella memoria condivisa non avran-

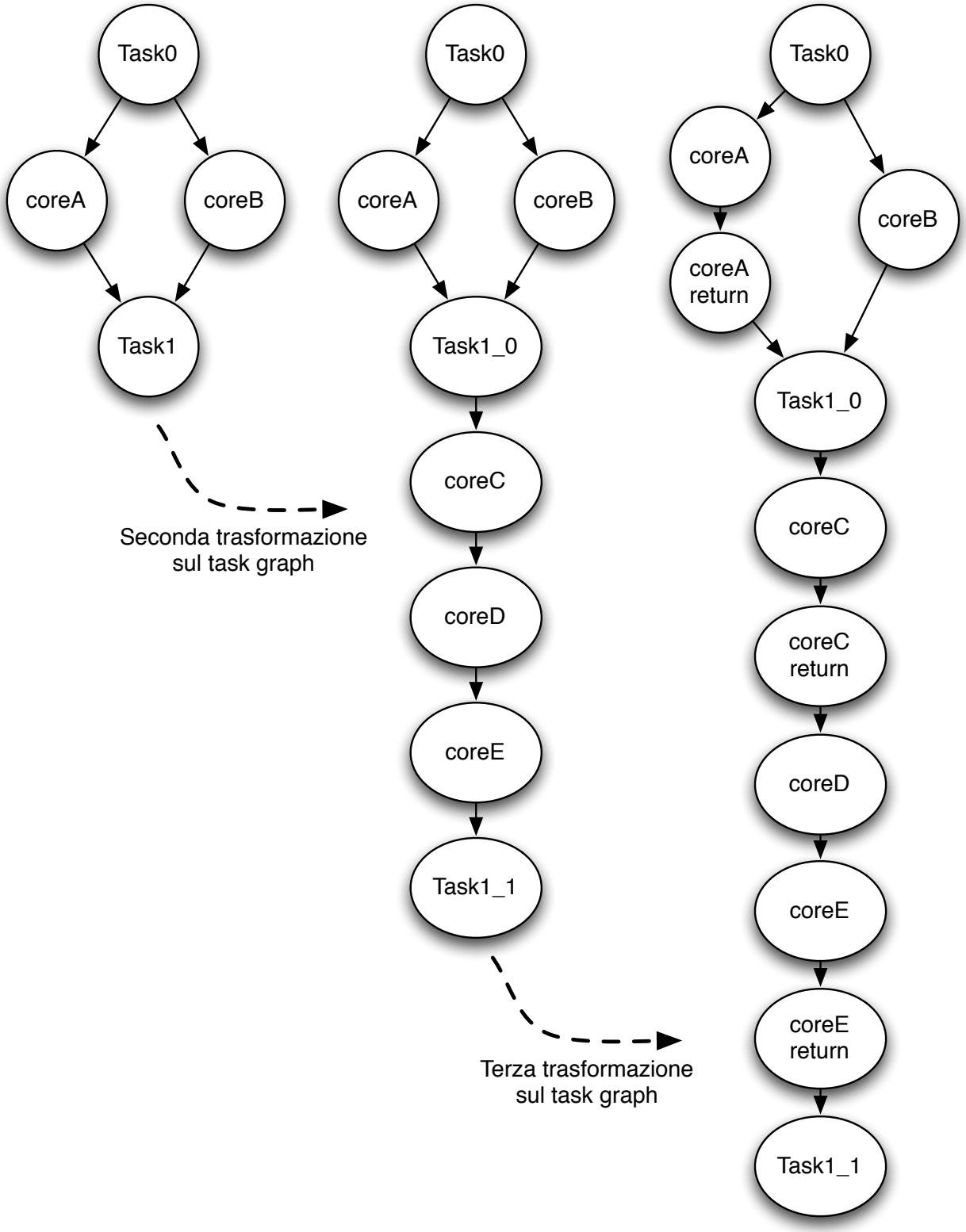

Figura 8.5: Trasformazioni sul task graph relativo al Listato 8.2.

no l'interfaccia NPI; in Figura 8.6 sono rappresentati i core associati all'applicazione in oggetto secondo le specifiche riportate nella Sezione 5.2. Si può quindi notare dalla figura come tutti i core presentino l'interfaccia verso il bus PLB, ognuno abbia un numero di registri differente in funzione del numero di ingressi della funzione che realizza e che *coreC* non abbia l'interfaccia NPI in quanto non necessita di accedere alla memoria condivisa.

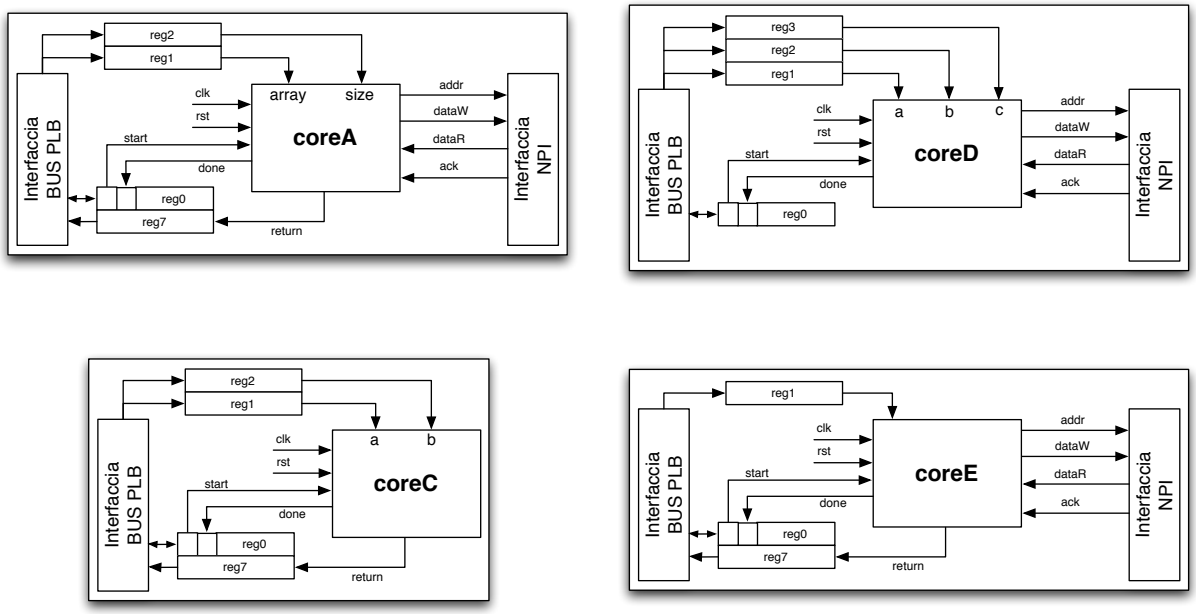

Figura 8.6: Core generati automaticamente dal framework con le loro interfacce relativamente al Listato 8.2.

A questo punto possono essere generati i driver per l'esecuzione dei vari task e le relative funzioni per la gestione degli interrupt da essi generati; il Listato 8.3 riporta il codice generato per il coreA, non si riporta nel dettaglio il codice per l'intera architettura in quanto non rilevante.

Analizzando nel dettaglio il listato proposto si vede come per ogni core venga create una struttura dati che funge da interfaccia con il driver di comunicazione, *data_coreA*, i cui campi sono generati in accordo con gli input e gli eventuali output della funzione a cui si riferiscono. La prima delle due funzioni è quella che si occupa di eseguire la funzionalità implementata da *coreA*; in particolare questa funzione scrive nei registri del core i valori contenuti nella struttura dati sopra specificata (righe 11 e 12) e invia il segnale di start (riga 13). Infine *Interrupt_coreA* è la funzione che si occupa di gestire l'interrupt ricevuto dal core una volta terminata la computazione; scrive nella struttura dati dell'interfaccia il valore di ritorno (righa 17), da comunicazione dell'avvenuta gestione dell'interrupt (riga 18) ed infine informa lo scheduler della terminazione del core (riga 19).

Listato 8.3 Driver di comunicazione.

```
1  typedef struct {
2          int *array;
3          int size;
4          int __return;
5  } data_coreA;
6
7  #define ACTIVE_coreA 1<<0
8  data_coreA Send_coreA;
9
10 void HW_coreA(void){
11         EM_BRIDGE_mWriteSlaveReg1(XPAR_COREA_BASEADDR, 0, Send_coreA.array);
12         EM_BRIDGE_mWriteSlaveReg2(XPAR_COREA_BASEADDR, 0, Send_coreA.size);
13         EM_BRIDGE_mWriteSlaveReg0(XPAR_COREA_BASEADDR, 0, START);
14 }
15
16 void Interrupt_coreA(void *CallbackRef){
17         Send_coreA.__return = EM_BRIDGE_mReadSlaveReg7(XPAR_COREA_BASEADDR, 0);
18         EM_BRIDGE_mWriteSlaveReg0(XPAR_COREA_BASEADDR, 0, 0);
19         ActiveTask_HW = ActiveTask_HW | ACTIVE_coreA;
20 }
```

A questo punto i prodotti delle prime tre fasi del flusso vengono integrati nel sistema finale e sintetizzati attraverso lo strumento della Xilinx, non si riportano i dettagli relativi a questa parte in quanto non particolarmente interessanti; in Figura 8.7 viene mostrata l'architettura generata in questo caso di studio in modo automatico dal framework e pronta per essere configurata su di una FPGA.

8.2.2 Analisi dello spazio delle soluzioni

In questo secondo caso di studio relativamente alla creazione di un design hardware che non sfrutta la riconfigurazione si fa riferimento all'algoritmo che implementa il filtro di Laplace. Come accennato nella relativa sezione questo algoritmo può essere parallelizzato a piacere; tuttavia limiti architetturali quali ad esempio il numero di core che possono essere collegati al *MPMC*, nel nostro caso 6, e limiti nell'area disponibile sul dispositivo influenzano la possibilità di parallelizzare l'algoritmo. Inoltre nel caso di un alto livello di parallelismo si potrebbero verificare problemi di congestione a livello dei bus di comunicazione; tali problemi sono spesso non prevedibili a priori in quanto si necessita di un modello del bus molto preciso al fine di individuarli e quantificarli correttamente.

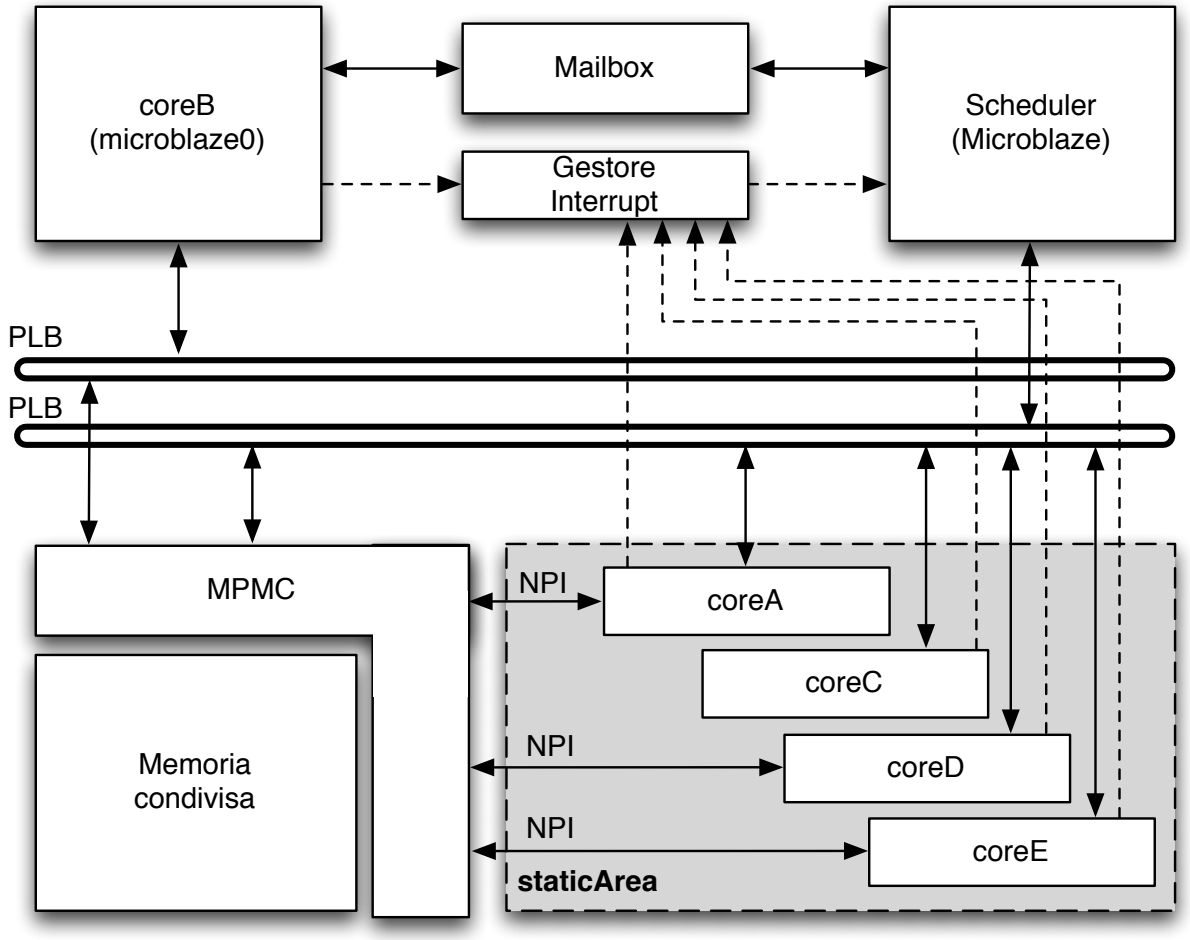

Figura 8.7: Architettura finale generata con riferimento al Listato 8.2.

Nel caso di studio in questione si è deciso di generare diverse soluzioni dell'algoritmo in esame utilizzando in parallelo 1,2,4 e 6 task che elaborano parti differenti dell'immagine in input; 6 è il limite massimo in quanto l'immagine in input è salvata nella memoria condivisa e quindi ogni core necessita di accedere a questa attraverso l'interfaccia NPI. Per ognuna di queste quattro soluzioni si sono implementati o tutti i task come core hardware, oppure si è assegnato un task al Microblaze e i rimanenti sono stati implementati come core hardware.

Le architetture relative alle diverse implementazioni proposte sono state generate automaticamente con il supporto del framework descritto in questo lavoro e successivamente si è provveduto al loro test sul dispositivo ed alla misura del loro tempo di esecuzione. Nella Tabella 8.1 sono riportati i risultati di questa fase di test; per ogni implementazione è riportato il partizionamento effettuato, l'occupazione d'area del dispositivo, il tempo d'esecuzione e lo speed-up relativo al caso di implementazione totalmente software dell'algoritmo. In Figura 8.8

sono rappresentati i valori di speed-up ottenuti con le diverse soluzioni.

Contrariamente a quanto ci si sarebbe aspettato i test mostrano come la soluzione computazionalmente più efficiente non sia quella che utilizza il massimo parallelismo, ma come la soluzione con 4 core implementati in hardware, e non 6, sia la più performante. Una spiegazione di questo può essere dovuta al fatto che dovendo ogni core accedere alla memoria condivisa sia in lettura che in scrittura, all'aumentare del numero di core si aumenta la congestione a livello del core *MPMC* e della memoria esterna.

Tabella 8.1: Occupazione d'area e tempo di esecuzione delle diverse implementazioni del filtro di Laplace. S_p rappresenta lo speed-up ottenuto nei confronti dell'implementazione interamente sequenziale in software.

Numero task			**Area**		**Tempo**	
	SW	HW	Numero slice	(%)	Cicli di clock	S_p
1	1	0	4 102	23.74	6 593 760	1.00
	0	1	5 088	29.44	2 663 764	2.48
2	1	1	4 452	25.76	3 348 185	1.97
	0	2	5 709	33.04	1 362 660	4.84
4	1	3	6 977	40,38	1 777 853	3.71
	0	**4**	**7 908**	**45.76**	**1 244 516**	**5.30**
6	1	5	8 408	48.66	1 396 466	4.72
	0	6	9 432	54.58	1 331 082	4.95

8.3 Creazione di un sistema riconfigurabile

Questa sezione presenta un caso di studio in cui l'obiettivo è quello di portare su scheda un sistema riconfigurabile che supporti l'esecuzione parallela di più applicazioni gestendo in modo automatico la riconfigurazione e l'assegnamento dei task alle risorse disponibili. In questo esempio inoltre verranno analizzati i comportamenti a runtime dei diversi algoritmi proposti andando ad analizzarne l'efficienza e l'overhead introdotto in fase di gestione del sistema. L'architettura di riferimento per questo esempio è quella riportata nella Sezione 5.4.

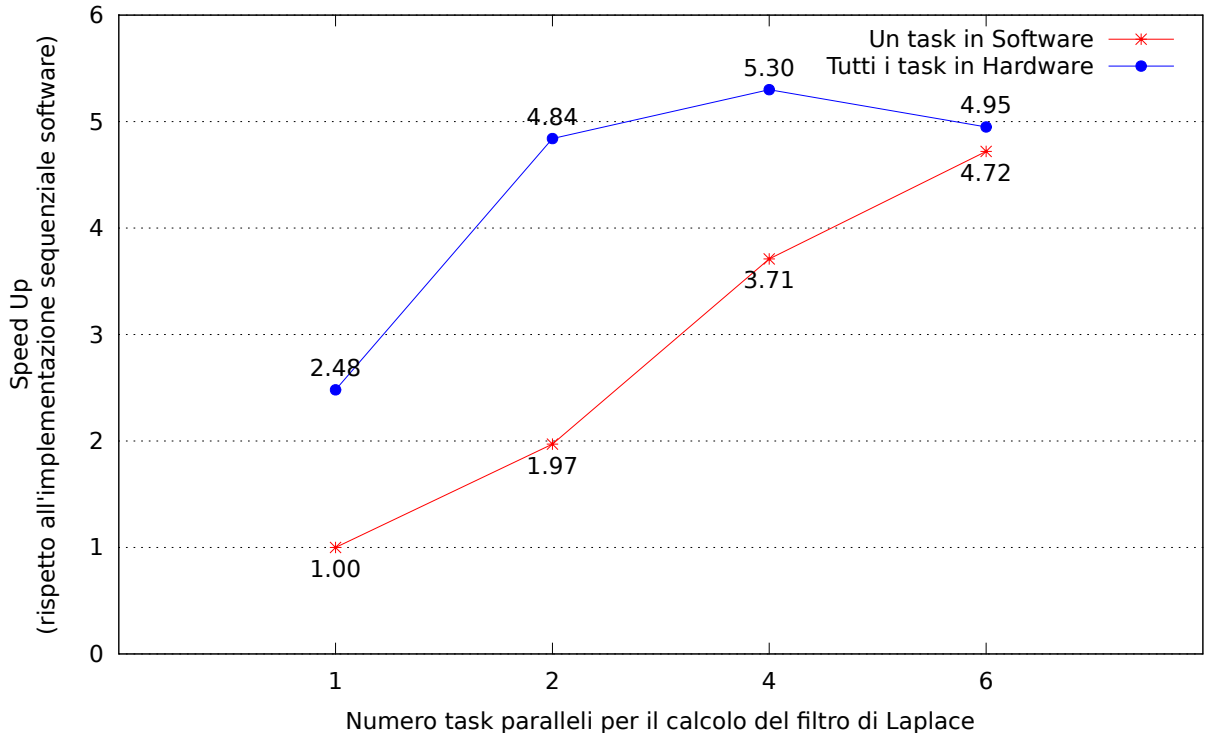

Figura 8.8: Speed-up del filtro di Laplace.

8.3.1 Il caso di studio

Il caso di studio consiste nel portare su dispositivo riconfigurabile gli algoritmi di rilevazione dei contorni (algoritmo di Canny) e l'algoritmo di rilevamento del movimento contemporaneamente. Ricordando, come mostrato nella Sezione 8.1, che ognuno dei due algoritmi è composta da quattro blocchi in serie, è facile notare come una loro implementazione in parallelo sull'architettura senza utilizzare la riconfigurazione sia chiaramente infattibile. Questo è dovuto al fatto che, anche utilizzando un solo task per ogni blocco dell'applicazione, si dovrebbero avere 8 blocchi contemporaneamente sul dispositivo che accedono alla memoria esterna, cosa chiaramente non possibile dato il limite di 6 dato dal core *MPMC*; una soluzione alternativa potrebbe consistere nell'implementare entrambe le applicazioni come un singolo core hardware, così facendo si avrebbero solo 2 core che accedono alla memoria, a discapito di dover però serializzare tutta la loro esecuzione.

Come si è visto dall'analisi del caso di studio precedente, applicazioni con algoritmi simili al filtro di Laplace, e quindi quelle prese in esame in questo caso di studio, ottengono il massimo speed-up rispetto ad un'implementazione totalmente in software quando vengono divise in quattro task che operano ognuno su di una parte diversa dell'immagine. Le due applicazio-

ni in esame sono quindi state divise secondo i risultati ottenuti al punto precedente ed i task graph così ottenuti sono illustrati in Figura 8.9. É evidente dai task graph che le due applicazioni condividono molte delle implementazioni per i loro task. Nella Tabella 8.2 sono mostrati i dettagli delle implementazioni ottenute per i vari task. Si fa presente che il tempo di riconfigurazione è uguale per tutti i task in quanto si riferisce all'area riconfigurabile che implementa il task, le quali sono identiche per costruzione.

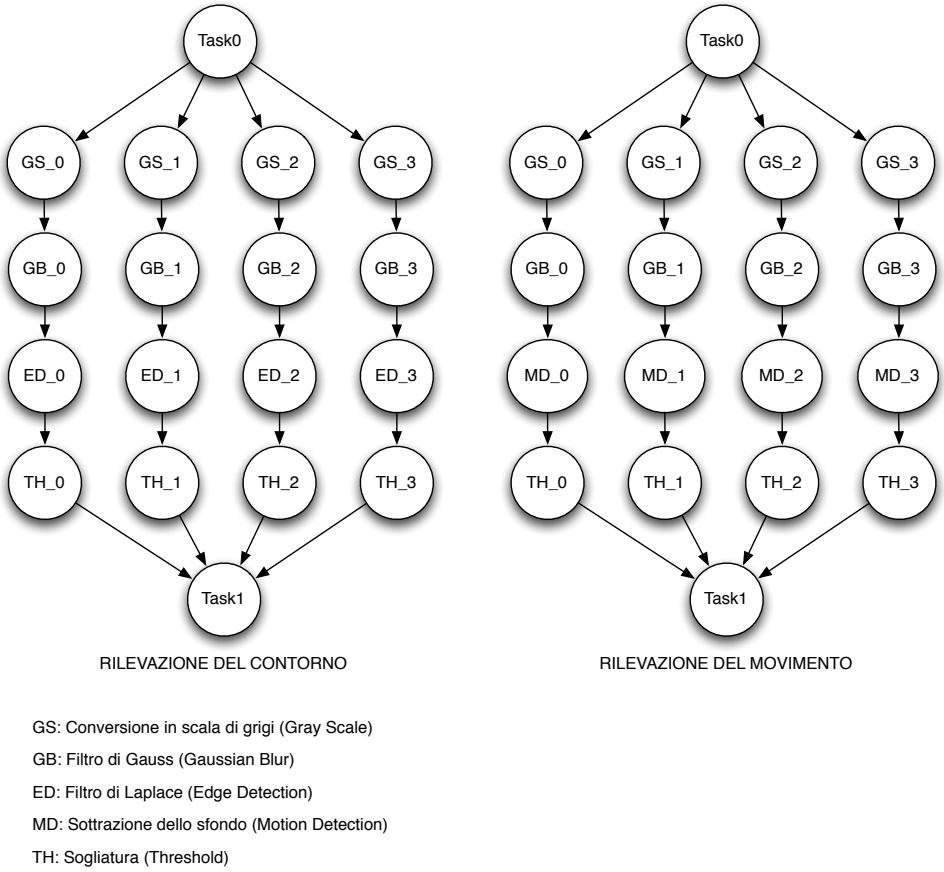

Figura 8.9: Task graph delle applicazioni in analisi.

8.3.2 Analisi dei diversi algoritmi

Analizzeremo ora le prestazioni del sistema creato dal framework relativamente all'esempio appena illustrato. In primo luogo analizzeremo come si comportano gli algoritmi in relazione al numero di aree che devono gestire, e al numero di applicazioni da eseguire iterativamente. Infine si mostrerà anche come sia possibile, date le statistiche sui tempi di esecuzione e riconfigurazione dei task mostrate nella Tabella 8.2, definire un assegnamento ottimale per questo problema che poi verrà gestito a runtime dall'applicazione.

Tabella 8.2: Tempi di esecuzione e riconfigurazione stimati ed occupazione d'area dei task.

Task	Tempo di esecuzione [cicli di clock]	Tempo di riconfigurazione [cicli di clock]	Slice
GS_i	88 713	390 745	5 120
GB_i	90 434	390 745	3 613
ED_i	78 234	390 745	2 723
MD_i	52 348	390 745	3 354
TH_i	47 688	390 745	3 224

Numero di aree riconfigurabili

Analizziamo ora il comportamento dei tre diversi algoritmi in relazione al numero di aree che devono gestire così da analizzare come varia il loro overhead con l'aumentare delle aree da gestire. Il test effettuato consiste nell'eseguire le due applicazioni in oggetto una singola volta in modo parallelo, al variare del numero delle aree sulle quali i task possono essere implementati, al fine di verificare la capacità delle diverse soluzioni di adattarsi al numero delle aree disponibili. Ogni task è stato associato a tutte le aree disponibili in ogni esperimento. In Tabella 8.3 sono riportati i tempi di esecuzione ottenuti su una media di 10 campioni. Gli stessi

Tabella 8.3: Tempo di esecuzione, in cicli di clock, del sistema di gestione a runtime in relazione al numero di aree riconfigurabili e all'algoritmo utilizzato.

Numero di aree	First Fit	Best Fit	Best Fit v2
1	126 448 339	126 685 149	126 883 673
2	151 760 785	154 761 067	151 992 707
3	171 131 790	171 128 844	165 233 448
4	171 131 823	171 132 525	164 717 794
5	171 133 820	171 137 544	164 932 297
6	171 130 269	171 129 938	167 856 040

dati sono mostrati nella Figura 8.10. Come si può notare l'algoritmo *Best First v2* riesce all'aumentare del numero di aree riconfigurabili ad essere più rapido degli altri due nell'eseguire il test. Questo fatto può essere spiegato ricordando che l'algoritmo in questione non cerca di

assegnare un task ad un'area (ed eventualmente riconfigurarla) solo quando il task è pronto, ma cerca di farlo in precedenza. A conferma di quanto detto, si può notare la differenza tra la curva di questo algoritmo e le altre due è di circa 500 000 cicli di clock che è un valore molto vicino a quello del tempo di riconfigurazione di una singola area riconfigurabile che quindi risulta mascherata durante l'esecuzione.

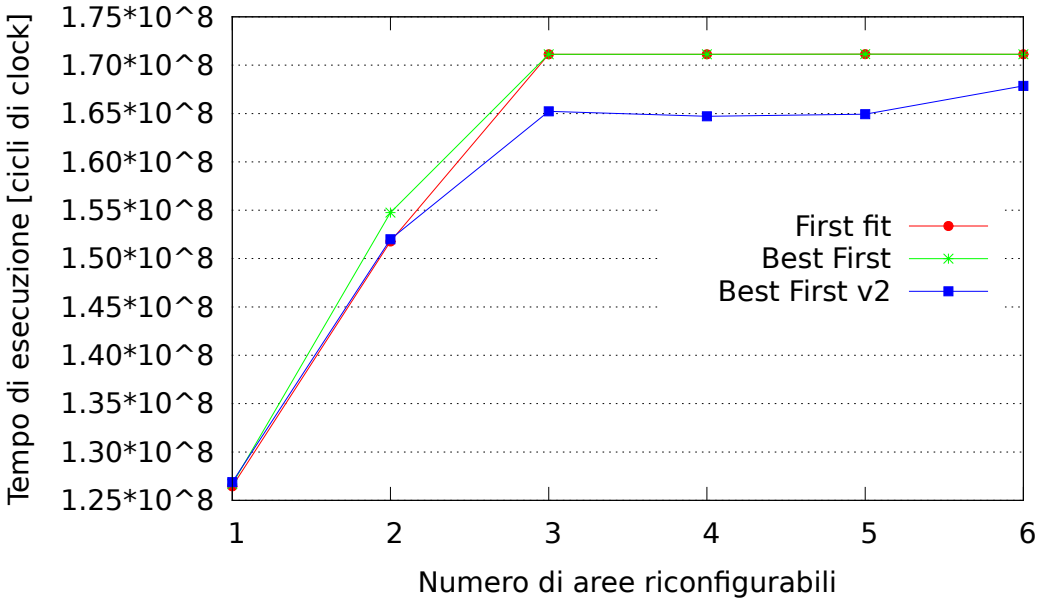

Figura 8.10: Analisi del sistema a runtime al variare del numero di aree riconfigurabili.

Iterazioni successive

Osserviamo ora come si comportano i tre diversi algoritmi all'aumentare del numero di applicazioni da eseguire. In particolare con questo esempio si vuole far notare come gli algoritmi sono in grado di riutilizzare le implementazioni già presenti sul dispositivo quando le applicazioni vengono iterate più volte. Nella Tabella 8.4 sono riportati i dati relativi a questo test; gli stessi sono poi rappresentati in Figura 8.11. La tabella riporta i dati relativi all'esecuzione in successione di 1, 2 oppure 3 istanze delle due applicazioni quando tutti i task presenti nel sistema sono mappati su tutte le 6 aree riconfigurabili. Dalla figura si nota chiaramente come l'algoritmo *Best Fit v2* riesca a garantire un maggior riutilizzo dei core già riconfigurati andando così a diminuire il tempo totale di esecuzione all'aumentare del numero delle iterazioni compiute. Inoltre, anche se non chiaramente visibile in figura, dalla tabella si può notare come nel caso dell'algoritmo *First Fit* il tempo di esecuzione tenda a peggiorare più che linearmente;

Tabella 8.4: Tempo di esecuzione, in cicli di clock, del sistema di gestione a runtime in relazione al numero di applicazioni da eseguire.

Numero di istanze	First Fit	Best Fit	Best Fit v2
1	171 130 269	171 129 938	167 856 040
2	356 954 343	336 956 672	334 012 319
3	519 552 341	479 561 534	418 012 319

questo andamento si ha appunto in virtù del fatto che il riutilizzo dei core hardware per questo algoritmo, come evidenziato nella Sezione 7.2, è dovuto solamente all'ordine in cui i task vengono eseguiti e non ad un'esplorazione tra le possibili soluzioni da parte dell'algoritmo stesso.

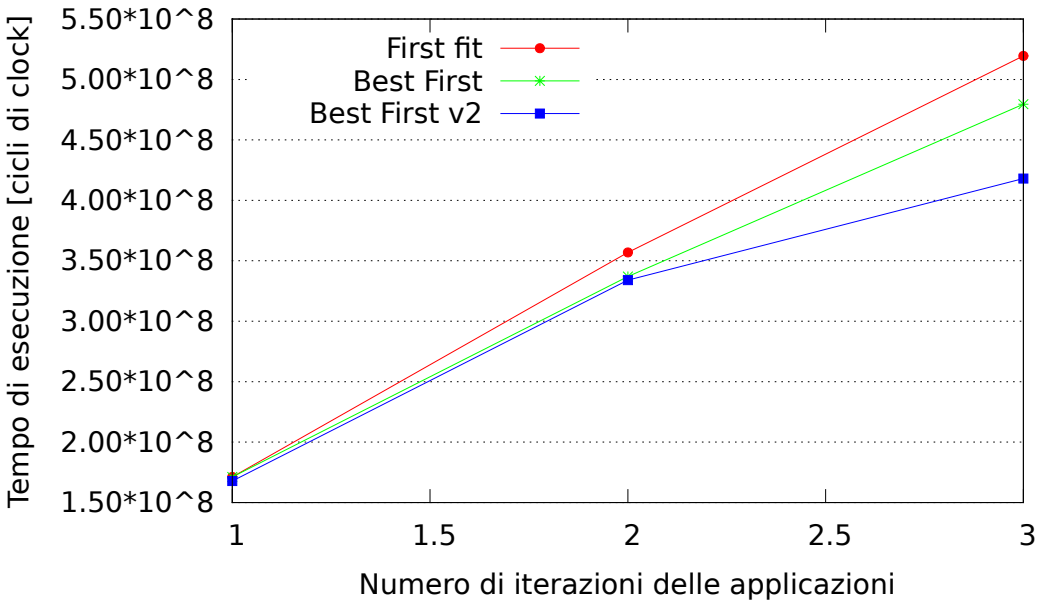

Figura 8.11: Analisi del sistema a runtime al variare del numero di applicazioni eseguite.

Assegnamento ottimo

Analizzando i dati della Tabella 8.2 si può notare come il tempo di riconfigurazione dell'area riconfigurabile sia superiore al tempo necessario per eseguire i quattro task paralleli dello stesso tipo. Risulta quindi evidente che un assegnamento vantaggioso dei task alle aree sia quello che, sfruttando sole 5 aree riconfigurabili, assegna ogni task ad un'area differente;

con questo accorgimento è possibile evitare che un task venga rimosso da un'area in favore di un altro. In questo caso è necessario effettuare la riconfigurazione delle aree una sola volta in quanto poi nessuna delle aree verrà più riconfigurata e gli algoritmi potranno sfruttare appieno la loro capacità di riutilizzare i core già configurati. Questo test è fatto eseguendo più iterazioni della coppia di applicazioni in esame come fatto nel caso precedente; i risultati sono riportati nella Tabella 8.5. Come si può notare dal confronto diretto con la Tabella 8.4 l'assegnamento risulta migliore di quello del caso precedenti (dove ogni task era assegnato ad ogni core), infatti i tempi totali di esecuzione sono ora diminuiti di circa un terzo per tutti i casi. Infine si può notare come in quest'ultimo esempio l'algoritmo che produce i peggiori risultati sia il *Best Fit v2*; in questo caso infatti non devono essere effettuati calcoli di alcun tipo e la soluzione ottima viene trovata anche dall'algoritmo *First Fit*, in quanto essendo ogni task assegnato ad una singola area la prima adatta a realizzarlo è proprio quella cercata; questa situazione evidenzia come l'algoritmo *Best Fit v2* abbia generalmente un overhead superiore agli altri due, anche se nella maggioranza dei casi tale algoritmo grazie alla capacità di anticipare le riconfigurazioni sia superiore agli altri due.

Tabella 8.5: Tempo di esecuzione, in cicli di clock, del sistema di gestione a runtime con l'assegnamento ottimo.

Numero di istanze	First Fit	Best Fit	Best Fit v2
1	116 129 376	116 139 428	115 843 850
2	231 332 316	231 338 157	233 299 265
3	346 538 561	346 535 812	348 535 812

Considerazioni finali

In questa sezione si sono mostrate le caratteristiche degli algoritmi atti alla gestione a runtime del sistema ed è stato effettuato un confronto tra le loro caratteristiche. Benché si sia evidenziata la bontà degli algoritmi in questione, soprattutto per il *Best Fit v2*, va fatto presente che il tempo totale d'esecuzione delle applicazioni in oggetto rimane molto elevato e quindi la generazione automatica e la gestione del sistema con questi algoritmi va analizzata caso per caso. Una scelta opportuna sarebbe ad esempio quella di ricorrere alla generazione del sistema riconfigurabile solo nel caso in cui il tempo d'esecuzione dei task sia effettivamente superiore a

quello di riconfigurazione; oppure quando vi è l'effettiva necessità di dare al dispositivo finale un'elevata flessibilità nel numero di applicazioni da gestire pur contenendone la dimensione.

8.4 Sommario

In questo capitolo si sono mostrati i risultati del lavoro svolto. In primo luogo si è mostrato l'intero flusso di lavoro del framework su di un caso di studio semplice, ma che al contempo ha messo in evidenza tutte le peculiarità del flusso proposto. Si è poi mostrato come il framework sviluppato possa essere usato dal progettista come supporto per la fase di esplorazione dello spazio delle soluzioni; in questo modo esso può validare diverse possibilità architetturali o di partizionamento dell'applicazione direttamente sul dispositivo e non facendo ricorso a simulatori o modelli che spesso non riescono a fornire un'analisi completa e corretta della situazione. Infine si sono discusse le proprietà dei diversi algoritmi messi a disposizione del progettista per la gestione del sistema a runtime; si sono mostrati diversi test effettuati al fine di verificarne i pregi e i difetti così come sono stati descritti nella Sezione 7.2.

Capitolo 9

Conclusioni

In questo capitolo verranno analizzate le conclusioni relative al lavoro svolto in questa tesi. Dapprima viene ripreso e descritto il flusso di lavoro proposto illustrando come esso, a fronte dei risultati presentati nel Capitolo 8, sia di effettivo aiuto ai progettisti hardware o software al fine di creare un sistema Multi Processors System on Chip (MPSoC). Nell'affrontare questa analisi verranno anche ripresi gli aspetti, già sottolineati in precedenza, per i quali il flusso di lavoro proposto può essere ulteriormente migliorato. Infine verranno proposti sviluppi futuri volti in parte a cercare di risolvere i problemi appena evidenziati ed in parte ad estendere il framework sviluppato al fine di riuscire ad affrontare ulteriori problemi e progettare architetture diverse da quelle prese in considerazione in questo lavoro.

9.1 Considerazioni finali

L'incessante necessità di ottenere miglioramenti delle performance in ogni ambito applicativo, ha fornito una spinta fondamentale alla ricerca nel campo delle architetture hardware; le architetture ad oggi disponibili, infatti, non sempre sono adatte per garantire tali performance. Spesso sono necessarie architetture dedicate in grado di ottimizzare l'esecuzione di una singola applicazione o di un singolo dominio applicativo. Inoltre i progressi tecnologici nei dispositivi hardware, soprattutto nel campo dei dispositivi riconfigurabili, sono ad oggi poco presi in considerazione per la mancanza di strumenti che assistono il progettista nello sviluppo.

In questo lavoro è stato proposto un flusso di lavoro in grado di assistere un progettista hardware nella creazione di un sistema MPSoC per dispositivi Field Programmable Gate Array (FPGA), oppure un progettista software che vuole provare a verificare se l'applicazione in fase di sviluppo possa beneficiare dell'esecuzione su di un'architettura dedicata. In particolare si è cercato di sviluppare uno strumento che rispondesse ai bisogni di tre tipologie di utenti ovvero:

- il progettista hardware: esso ha la necessità di sviluppare architetture avanzate che riescano a garantire le performance stringenti ad oggi richieste; esso quindi deve riuscire a testare l'architettura in fase di sviluppo con un elevato numero di casi di studio senza però perdere troppo tempo nel realizzare i singoli core hardware manualmente;

- l'esperto di un dominio applicativo: questa tipologia di utente corrisponde allo sviluppatore software che necessita di eseguire una determinata applicazione nel modo più efficiente e che spesso non si appoggia ad architetture hardware dedicate in quanto non è in possesso dell'esperienza necessaria per programmarle;

- il programmatore a livello di sistema operativo: questa ultima tipologia di utente ha la necessità di sviluppare politiche di scheduling e, nel caso di sistemi riconfigurabili, di supporto alla riconfigurazione dinamica parziale a runtime; in questo caso l'architettura e le applicazioni sono solo un mezzo per questo tipo di utente per testare le politiche e gli algoritmi sviluppati e quindi non deve essere necessario per questo utente perdere troppo tempo per creare il sistema per il testing.

Il framework si propone come uno strumento semi automatico al fine di riuscire a sopperire alle necessità di questi utenti. In particolare il flusso proposto è completamente integrato con i flussi di lavoro di *Xilinx Platform Studio* e *PlanAhead*. Il progettista hardware può quindi disegnare la sua architettura con questi strumenti, sviluppare le applicazioni di test in un

linguaggio ad alto livello e quindi provare diverse possibili configurazioni di assegnamento dei task ai processori presenti al fine di valutarne le differenti performance. In quest'ottica il progettista potrebbe ad esempio sviluppare un nuovo controllore della memoria ed utilizzare il framework per effettuare un test analogo a quello mostrato nella Sezione 8.2.2 ed ad esempio ottenere che il nuovo controllore riesce a garantire un livello di parallelismo superiore.

La seconda tipologia di utente invece, essendo esperto del singolo dominio applicativo, è in grado di scrivere l'applicazione dividendola in task in modo che questa tragga il maggior beneficio possibile dall'architettura sottostante. Per questa tipologia di utente sono stati proposti un modello architetturale, nelle varianti con supporto o meno alla riconfigurabilità dinamica parziale, che il progettista può utilizzare per testare il funzionamento su sistema dedicato della sua applicazione; per effettuare questo test lo sviluppatore non necessita di scrivere una singola riga di codice VHDL o Verilog in quanto il framework si prende cura di generare automaticamente i core hardware necessari, le interfacce e di integrarli all'interno del sistema.

Infine il programmatore interessato a sviluppare politiche di scheduling, o per gestire la riconfigurazione, può concentrarsi solamente nello sviluppare tali politiche modificando le parti relative nel layer di gestione software ed utilizzando le strutture dati, illustrate nella Sezione 7.1 che sono automaticamente generate dal framework nella fase di integrazione dell'applicazione nell'architettura finale; anche in questo caso al programmatore non è richiesta la scrittura di codice di descrizione hardware per testare i propri algoritmi. Ovviamente in un team di sviluppo possono coesistere queste figure ed ognuno parteciperà sviluppando una delle singole parti al fine di ottenere un sistema che rispetti le performance richieste.

Pur assistendo lo sviluppatore effettuando il flusso di lavoro in modo automatico a partire dagli input, e pur semplificando notevolmente la specifica degli input permettendo tramite degli script di generare automaticamente i file XML necessari a partire dal progetto di *Xilinx Platform Studio* e dalle annotazioni nel codice dell'applicazione, il framework presenta alcune limitazioni. In primo luogo il *backend* necessita di essere personalizzato in relazione all'architettura di riferimento; quindi a fronte dell'introduzione di una nuova architettura è necessario di riscrivere parte del *backend*, che corrisponde alla fase 3 del flusso, in quanto ad ora non sono supportate tutte le interfacce e le soluzioni architetturali possibili. In particolare qualora si decida di utilizzare una nuova interfaccia per i core non ancora supportata dal framework è necessario inserirla e personalizzare sia la sua generazione in base alla struttura del core sia la parte relativa alla scrittura dei driver. L'altra limitazione evidenziata dal lavoro è dovuta alle interfacce dei core riconfigurabili; in particolare nella Sezione 6.3 si è evidenziato come la

scelta di standardizzare le interfacce si ripercuota sulla necessità del programmatore di riscrivere il codice dell'applicazione in modo di passare per riferimento ogni parametro che non sia convertibile in un valore intero senza perdere informazione.

9.2 Sviluppi futuri

Possibili sviluppi futuri legati a questo lavoro sono volti in primo luogo a cercare di risolvere i limiti mostrati nella sezione precedente. Per quanto riguarda il supporto a diversi modelli architetturali sarebbe utile estendere il framework per supportare un'architettura in cui il passaggio dati tra i core possa avvenire anche attraverso un bus condiviso o con altre topologie di connessione e non per forza attraverso la memoria esterna. Questa modifica sebbene non sia complicata da effettuare dal punto di vista architetturale e neanche nella sua automatizzazione obbliga a rivedere gli algoritmi per la gestione a runtime in quanto si deve esplicitamente aggiungere il vincolo che non è possibile riconfigurare un'area fino a quando il core in essa implementato non ha trasferito tutti i dati ai core successivi. Per quanto riguarda l'interfaccia standard è invece possibile estendere il framework in modo che esso sia in grado di identificare in fase di generazione dei core hardware i task critici, ovvero quei casi in cui i parametri ad esempio di tipo *float* sono passati al task in oggetto per valore, e di conseguenza riscrivere il prototipo della funzione associata facendo in modo di allocare il dato in memoria e trasferire solamente il puntatore all'area di memoria associata. Altra soluzione in questo caso è invece quella di utilizzare invece che una struttura dati con degli input di tipo fisso, una struttura dati interamente allocata in memoria e che al core venga passato solamente l'indirizzo di questa struttura dati; questa soluzione obbliga come nel caso precedente ad effettuare la riscrittura del prototipo della funzione e l'inserimento nel corpo delle funzioni delle istruzioni necessarie a recuperare i valori dalla struttura in memoria condivisa.

Altri lavori si possono invece concentrare sull'ottimizzazione dell'architettura sviluppata in questo lavoro al fine di migliorarne le performance. In primo luogo si può cercare di diminuire il tempo di riconfigurazione necessario utilizzando un core hardware per gestire la riconfigurazione invece che utilizzare il secondo Microblaze presente nell'architettura. Inoltre considerata la necessità di trasferire dati attraverso la memoria condivisa può essere utile dotare i core di un livello di *cache* che eventualmente effettui il *prefetch* dei dati così da sfruttarne la località spaziale in memoria; località che di sicuro si realizza nel caso in cui al core venga

passato l'indirizzo di una struttura dati relativa ai suoi parametri in ingresso, ma che è valida anche per le applicazioni di elaborazione di immagini presentate nel Capitolo 8.

Infine un'ultima serie di lavori futuri può concentrarsi sul miglioramento delle politiche di gestione a runtime del sistema. Si può pensare ad esempio di superare il limite imposto in questo lavoro nell'avere in esecuzione al massimo un'istanza di ogni applicazione ed estendere il livello software per gestire l'esecuzione concorrente di un numero non predefinito di applicazioni eventualmente anche più istanze della stessa. Si può inoltre cercare in questo ambito di introdurre la possibilità di avere processori software riconfigurabili così da poterli istanziare a runtime qualora il gestore del sistema ritenga che sia opportuno eseguire l'implementazione software di un task piuttosto che quella hardware. Infine si può cercare di capire la convenienza di implementare la parte di gestione del sistema a runtime in hardware invece che in software al fine di aumentarne l'efficienza; in particolare data la necessità di supportare l'utente che voglia definire nuove politiche di gestione sarebbe interessante non solo portare in hardware il livello di gestione del sistema, ma anche cercare di riuscire a generarlo automaticamente a partire dagli algoritmi forniti dallo sviluppatore.

Bibliografia

[1] G. Blake, R.G. Dreslinski, and T. Mudge. A survey of multicore processors. *Signal Processing Magazine, IEEE*, 26(6):26–37, 2009.

[2] IBM. 50 years of eda. `http://blog.dac.com/post/2012/01/19/IBM-video-50-years-of-EDA.aspx`.

[3] J. Xiong, T.M. Nguyen, and QM Wu. Fpga implementation of blob recognition. In *Computer and Robot Vision (CRV), 2011 Canadian Conference on*, pages 125–131. IEEE, 2011.

[4] M. Xu, W. Zhu, and Y. Zou. Design of a reconfigurable robot controller based on fpga. In *Embedded Computing, 2008. SEC'08. Fifth IEEE International Symposium on*, pages 216–222. IEEE, 2008.

[5] Q. Zhou, K. Yuan, H. Wang, and H. Hu. Fpga-based colour image classification for mobile robot navigation. In *Industrial Technology, 2005. ICIT 2005. IEEE International Conference on*, pages 921–925. IEEE, 2005.

[6] X. Lu, D. Ren, and S. Yu. Fpga-based real-time object tracking for mobile robot. In *Audio Language and Image Processing (ICALIP), 2010 International Conference on*, pages 1657–1662. IEEE, 2010.

[7] C. Huang and F. Vahid. Server-side coprocessor updating for mobile devices with fpgas. In *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*, pages 125–134. ACM, 2010.

[8] Intel. 60 years of the transistor: 1947 - 2007. `http://www.intel.com/technology/timeline.pdf`.

[9] V.V. Zhirnov, R.K. Cavin III, J.A. Hutchby, and G.I. Bourianoff. Limits to binary logic switch scaling-a gedanken model. *Proceedings of the IEEE*, 91(11):1934–1939, 2003.

[10] Intel. Intel's vision. `http://download.intel.com/newsroom/kits/idf/2011_fall/pdfs/Kirk_Skaugen_DCSG_MegaBriefing.pdf#page=21`.

[11] T.G. Mattson, R. Van der Wijngaart, and M. Frumkin. Programming the intel 80-core network-on-a-chip terascale processor. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, page 38. IEEE Press, 2008.

[12] Adapteva. `http://www.adapteva.com/`.

[13] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, 2008.

[14] A. Montone, V. Rana, M.D. Santambrogio, and D. Sciuto. Harpe: a harvard-based processing element tailored for partial dynamic reconfigurable architectures. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–8. IEEE, 2008.

[15] M. Ranjan, L. Gopalakrishnan, K. Srihari, and C. Woychik. Die cracking in flip chip assemblies. In *Electronic Components & Technology Conference, 1998. 48th IEEE*, pages 729–733. IEEE, 1998.

[16] J. Lach, W.H. Mangione-Smith, and M. Potkonjak. Efficiently supporting fault-tolerance in fpgas. In *Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays*, pages 105–115. ACM, 1998.

[17] Milind Girkar and Constantine D. Polychronopoulos. The hierarchical task graph as a universal intermediate representation. *International Journal of Parallel Programming*, 22(5):519–551, October 1994.

[18] C.D. Polychronopoulos. The hierarchical task graph and its use in auto-scheduling. In *Proceedings of the 5th international conference on Supercomputing*, pages 252–263. ACM, 1991.

[19] Xilinx. `http://www.xilinx.com/`.

[20] Altera. `http://www.altera.com/`.

[21] T.H. Cormen. *Introduction to algorithms*. The MIT press, 2001.

[22] A.V. Aho, M.S. Lam, R. Sethi, and J.D. Ullman. *Compilers: principles, techniques, and tools*. 2007.

[23] Torsten Kempf, Kingshuk Karuri, Stefan Wallentowitz, Gerd Ascheid, Rainer Leupers, and Heinrich Meyr. A SW performance estimation framework for early system-level-design using fine-grained instrumentation. pages 468–473, March 2006.

[24] Péter Arató, Zoltán Ádám Mann, and András Orbán. Algorithmic aspects of hardware/software partitioning. *ACM Transactions on Design Automation of Electronic Systems*, 10(1):136–156, January 2005.

[25] Ahmed Zaki Semar Shahul and Oliver Sinnen. Scheduling task graphs optimally with A*. *The Journal of Supercomputing*, 51(3):310–332, March 2010.

[26] Fabrizio Ferrandi, Pier Luca Lanzi, Christian Pilato, Donatella Sciuto, and Antonino Tumeo. Ant Colony Heuristic for Mapping and Scheduling Tasks and Communications on Heterogeneous Embedded Systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 29(6):911–924, June 2010.

[27] Ralf Niemann and Peter Marwedel. An Algorithm for Hardware Software Partitioning Using Mixed Integer Linear Programming. *Work*, pages 1–34, 1991.

[28] Fabrizio Ferrandi, Christian Pilato, Donatella Sciuto, and Antonino Tumeo. Mapping and scheduling of parallel C applications with Ant Colony Optimization onto heterogeneous reconfigurable MPSoCs. In *2010 15th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 799–804. IEEE, January 2010.

[29] Antonino Tumeo, Christian Pilato, Fabrizio Ferrandi, Donatella Sciuto, Pier Luca Lanzi, and Politecnico Milano. Ant Colony Optimization for Mapping and Scheduling in Heterogeneous Multiprocessor Systems. *System*, pages 142–149, 2008.

[30] G. Wang, W. Gong, and R. Kastner. System level partitioning for programmable platforms using the ant colony optimization. In *International Workshop on Logic & Synthesis (IWLS'04), Temecula, California*. Citeseer, 2004.

[31] R.P. Dick and N.K. Jha. Mogac: a multiobjective genetic algorithm for hardware-software cosynthesis of distributed embedded systems. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 17(10):920–935, 1998.

[32] T. Lei and S. Kumar. A two-step genetic algorithm for mapping task graphs to a network on chip architecture. In *Digital System Design, 2003. Proceedings. Euromicro Symposium on*, pages 180–187. IEEE, 2003.

[33] W. Jigang, Thambipillai Srikanthan, and Guang Chen. Algorithmic aspects of hardware/software partitioning: 1d search algorithms. *Computers, IEEE Transactions on*, 59(4):532–544, April 2010.

[34] B.W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, 49(2):291–307, 1970.

[35] Frank Vahid. Extending the Kernighan / Lin Heuristic for Hardware and Software Functional Partitioning. *Design*, 261:237–261, 1997.

[36] Sudarshan Banerjee, Elaheh Bozorgzadeh, and Nikil D. Dutt. Integrating Physical Constraints in HW-SW Partitioning for Architectures With Partial Dynamic Reconfiguration. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 14(11):1189–1202, November 2006.

[37] JA Clemente and Javier Resano. A hardware implementation of a run-time scheduler for reconfigurable systems. *(VLSI) Systems, IEEE*, 42(7):1263–1276, 2010.

[38] Diana Göhringer, Michael Hübner, E.N. Zeutebouo, and Jürgen Becker. CAP-OS: Operating system for runtime scheduling, task mapping and resource management on reconfigurable multiprocessor architectures. *Proc. of RAW at the IPDPS*, 2010.

[39] Xilinx. Autoesl. `http://www.xilinx.com/tools/autoesl.htm`.

[40] Mitsuhisa Sato. OpenMP: parallel programming API for shared memory multiprocessors and on-chip multiprocessors. In *System Synthesis, 2002. 15th International Symposium on*, pages 109–111. IEEE, 2002.

[41] H Nikolov, M Thompson, T Stefanov, A Pimentel, S. Polstra, R. Bose, C. Zissulescu, and E. Deprettere. Daedalus: toward composable multimedia MP-SoC design. In *Proceedings of the 45th annual Design Automation Conference*, pages 574–579. ACM, 2008.

[42] S. Verdoolaege, H. Nikolov, and T. Stefanov. Pn: a tool for improved derivation of process networks. *EURASIP Journal on Embedded Systems*, 2007(1):19–19, 2007.

[43] A.D. Pimentel, C. Erbas, and S. Polstra. A systematic approach to exploring embedded system architectures at multiple abstraction levels. *Computers, IEEE Transactions on*, 55(2):99–112, 2006.

[44] C. Erbas, A.D. Pimentel, M. Thompson, and S. Polstra. A framework for system-level modeling and simulation of embedded systems architectures. *EURASIP Journal on Embedded Systems*, 2007(1):2–2, 2007.

[45] H. Nikolov, T. Stefanov, and E. Deprettere. Systematic and automated multiprocessor system design, programming, and implementation. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 27(3):542–555, 2008.

[46] H. Nikolov, T. Stefanov, and E. Deprettere. Multi-processor system design with espam. In *Hardware/Software Codesign and System Synthesis, 2006. CODES+ ISSS'06. Proceedings of the 4th International Conference*, pages 211–216. IEEE, 2006.

[47] G. Kahn. The semantics of a simple language for parallel programming. *proceedings of IFIP Congress74*, 74:471–475, 1974.

[48] Sjoerd Meijer, H. Nikolov, and T. Stefanov. Combining process splitting and merging transformations for polyhedral process networks. In *Embedded Systems for Real-Time Multimedia (ESTIMedia), 2010 8th IEEE Workshop on*, pages 97–106. IEEE, 2010.

[49] Sjoerd Meijer, Hristo Nikolov, and Todor Stefanov. Throughput modeling to evaluate process merging transformations in polyhedral process networks. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 747–752. European Design and Automation Association, 2010.

[50] Dmitry Nadezhkin, H. Nikolov, and T. Stefanov. Translating affine nested-loop programs with dynamic loop bounds into Polyhedral Process Networks. In *Embedded Systems for Real-Time Multimedia (ESTIMedia), 2010 8th IEEE Workshop on*, pages 21–30. IEEE, 2010.

[51] Dmitry Nadezhkin and Todor Stefanov. Identifying communication models in Process Networks derived from Weakly Dynamic Programs. *2010 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*, pages 372–379, July 2010.

[52] Dmitry Nadezhkin and Todor Stefanov. Automatic Derivation of Polyhedral Process Networks from While-Loop Affine Programs. *liacs.nl*, pages 102–111, 2011.

[53] Silvia Ozana, Todor Stefanov, and Koen Bertels. Loop Unrolling and Shifting for Reconfigurable Architectures. In *Memory*, pages 167–172, 2008.

[54] Mohamed Bamakhrama and Todor Stefanov. Hard-real-time scheduling of data-dependent tasks in embedded streaming applications. In *Embedded Software (EMSOFT), 2011 Proceedings of the International Conference on*, pages 195–204. IEEE, 2011.

[55] Diana Göhringer. High Performance Reconfigurable Multi-Processor-Based Computing on FPGAs. In *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing, Workshops and Phd Forum (IPDPSW'10)*, pages 1–4, 2010.

[56] S.L. Graham, P.B. Kessler, and M.K. Mckusick. Gprof: A call graph execution profiler. *ACM Sigplan Notices*, 17(6):120–126, 1982.

[57] D. Gohringer, J. Luhmann, and J. Becker. Generatercs: A high-level design tool for generating reconfigurable computing systems. In *Very Large Scale Integration (VLSI-SoC), 2009 17th IFIP International Conference on*, pages 159–164. IEEE, 2009.

[58] IBM. Powerpc microprocessor family: The programming environments for 32-bit microprocessors. `https://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/852569B20050FF778525699600719DF2/$file/6xx_pem.pdf`.

[59] Xilinx. Microblaze processor reference guide. `http://www.xilinx.com/support/documentation/sw_manuals/mb_ref_guide.pdf`.

[60] Xilinx. Picoblaze 8-bit embedded microcontroller user guide. `http://www.eng.auburn.edu/~strouce/class/elec4200/ug129.pdf`.

[61] D. Go hringer, Bin Liu, M. Hubner, and J. Becker. Star-wheels network-on-chip featuring a self-adaptive mixed topology and a synergy of a circuit-and a packet-switching communication protocol. In *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, pages 320–325. IEEE, 2009.

[62] D. Gohringer, Oliver Oey, M. Hubner, and J. Becker. Heterogeneous and Runtime Parameterizable Star-Wheels Network-on-Chip. In *Embedded Computer Systems (SAMOS), 2011 International Conference on*, pages 380–387. IEEE, 2011.

[63] Jason Cong, Yiping Fan, Guoling Han, Wei Jiang, and Zhiru Zhang. Platform-based behavior-level and system-level synthesis. In *SOC Conference, 2006 IEEE International*, pages 199–202. IEEE, 2006.

[64] F.E. Allen. Control flow analysis. In *ACM Sigplan Notices*, volume 5, pages 1–19. ACM, 1970.

[65] A. Orailoglu and D.D. Gajski. Flow graph representation. In *Proceedings of the 23rd ACM/IEEE Design Automation Conference*, pages 503–509. IEEE Press, 1986.

[66] Deming Chen, Jason Cong, and Yiping Fan. Optimality study of resource binding with multi-Vdds. *Proceedings of the 43rd annual Design*, 2006.

[67] J. Cong. Bitwidth-aware scheduling and binding in high-level synthesis. *Proceedings of the ASP-DAC 2005. Asia and South Pacific Design Automation Conference, 2005.*, 2:856–861, 2005.

[68] J. Cong. Architecture and compilation for data bandwidth improvement in configurable embedded processors. *ICCAD-2005. IEEE/ACM International Conference on Computer-Aided Design, 2005.*, pages 263–270, 2005.

[69] Jason Cong, Yiping Fan, Guoling Han, and Ashok Jagannathan. Instruction set extension with shadow registers for configurable processors. *FPGA*, 2005.

[70] Jason Cong, Yiping Fan, Guoling Han, and Xun Yang. Architecture and synthesis for on-chip multicycle communication. *Computer-Aided Design*, 23(4):550–564, 2004.

[71] Jason Cong, Yiping Fan, and Wei Jiang. Platform-Based Resource Binding Using a Distributed Register-File Microarchitecture. *2006 IEEE/ACM International Conference on Computer Aided Design*, 1(c):709–715, November 2006.

[72] petalogix. `http://www.petalogix.com/`.

[73] Xilinx. Platform studio user guide. `http://www.xilinx.com/ise/embedded/edk6_2docs/platform_studio_ug.pdf`.

[74] Xilinx. Planahead user guide. `http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/PlanAhead_UserGuide.pdf`.

[75] Xilinx. Logicore ip multi-port memory controller (mpmc). `http://www.xilinx.com/support/documentation/ip_documentation/mpmc.pdf`.

[76] J.G. Siek, L.Q. Lee, and A. Lumsdaine. *Boost Graph Library: User Guide and Reference Manual, The*. Addison-Wesley Professional, 2001.

[77] J. Ellson, E. Gansner, L. Koutsofios, S. North, and G. Woodhull. Graphviz—open source graph drawing tools. In *Graph Drawing*, pages 594–597. Springer, 2002.

[78] R.C. Gonzalez and E. Richard. Woods, digital image processing, 2002.

[79] W. He and K. Yuan. An improved canny edge detector and its realization on fpga. In *Intelligent Control and Automation, 2008. WCICA 2008. 7th World Congress on*, pages 6561–6564. IEEE, 2008.

[80] Q. Zhou and J.K. Aggarwal. Tracking and classifying moving objects from video. In *Proceedings of IEEE Workshop on Performance Evaluation of Tracking and Surveillance*. Hawaii, USA, 2001.