Politecnico di Milano

Facoltà di Ingegneria dell'informazione

Corso di Laurea Magistrale in Ingegneria Informatica


# ARGO: UN FRAMEWORK PER IL SUPPORTO ALL'ADATTABILITÀ DI APPLICAZIONI IN ARCHITETTURE MULTIPROCESSORE

**Relatore:**
Prof.re Gianluca Palermo

**Tesi di Laurea di:**
Davide Gadioli
Matricola n. 765604

Anno Accademico 2012/2013

*Alla mia famiglia che mi ha supportato*

*durante tutti i miei studi,*

Davide.

# Indice

# Elenco delle figure

# Elenco degli algoritmi

# Sommario

Questo elaborato di tesi descrive il framework ARGO, il cui scopo è quello di fornire a un'applicazione la capacità di adattarsi all'evolversi della situazione, senza la necessità di coordinarsi con altri attori e in maniera trasparente rispetto allo sviluppatore.

Per ottenere questo risultato il framework permette allo sviluppatore di delineare dei vincoli entro i quali l'applicazione può operare e di definire le caratteristiche che determinano la qualità di un punto di lavoro; unendo queste informazioni all'introspezione fornita dai monitor durante l'esecuzione dell'algoritmo, l'Application-Specific Run-Time Manager è chiamato a individuare il punto di lavoro più adatto.

Nell'eventualità che l'applicazione abbia bisogno di operare in diverse modalità, il framework permette allo sviluppatore di creare stati distinti, ognuno con i propri vincoli e preferenze riguardo la bontà di un punto di lavoro. Per esempio si può definire uno stato in cui sono presenti dei requisiti sulle prestazioni e solo delle preferenze rispetto al consumo energetico, mentre in un altro stato si devono rispettare dei requisiti sul consumo energetico e avere solo delle preferenze sulle prestazioni.

Pensando all'utilizzo del framework anche in un contesto distribuito, viene fornita la possibilità di creare dei vincoli rilassabili. Essi non influenzano direttamente la scelta del punto di lavoro, ma nel caso non siano rispettati, l'Application-Specific Run-Time Manager intraprende un'azione definita dello sviluppatore; come ad esempio l'invio di una richiesta per modificare il funzionamento di un'altra applicazione che non è in esecuzione sulla stessa macchina.

Per valutare i risultati ottenuti dal framewwork è stata realizzata un'applicazione che forza l'esecuzione delle operazioni più dispendiose nella situzione peggiore. In questo modo si può ottenere il limite superiore dell'overhead introdotto; mentre per poter osservare le prestazioni del framework se applicato ad uno scenario generico, abbiamo utilizzato diverse applicazioni reali le cui funzioni variano dal calcolo di azioni finanziarie alla codifica di un filmato.

# Capitolo 1

# Introduzione

In questi ultimi anni si è verificata una rapida evoluzione nell'architettura dei sistemi embedded, diventando sempre più complessa e ricca di risorse. Per rendersi conto della profondità del cambiamento è sufficiente ripensare ai telefoni cellulari usati qualche anno fa e paragonarli con quelli che si trovano oggi sul mercato, dove l'impiego di multiprocessori e acceleratori hardware è diventato comune.

In aggiunta all'incremento di potenza computazionale, si è osservato anche una notevole diffusione dell'uso di sistemi embedded per svolgere compiti sempre più articolati e diversificati. L'esempio più immediato è pensare a quanti dispositivi elettronici sono comparsi nelle vetture moderne, dove il pilota viene assistito sempre di più durante la guida.

Anche se questi cambiamenti sembrano ripercorrere l'evoluzione che hanno effettuato i classici computer, esiste una differenza importante che distingue i due mondi. Quando si progetta l'architettura di un elaboratore si cerca sempre di massimizzare le prestazioni, lasciando tipicamente il consumo energetico in secondo piano. Siccome sono macchine che non hanno uno scopo preciso, ma devono eseguire compiti generici, avere più potenza di elaborazione aumenta il numero di funzioni che possono svolgere e di conseguenza poter essere considerati migliori.

Nei sistemi embedded invece si parte dall'obbiettivo che si vuole ottenere. Sapere qual'è la funzione che deve essere svolta, permette di scegliere, tra le soluzioni che raggiungono lo scopo, quella che più si adatta alle esigenze del progettista. Considerando che normalmente i dispositivi creati vengono alimentati a batterie o comunque hanno a disposizione risorse limitate, l'efficienza risulta essere una notevole discriminante nella scelta della soluzione da implementare.

Le applicazioni parametriche sono degli algoritmi che svolgono una determinata funzione lasciando la libertà a chi li esegue di specificare dei parametri che influenzano le prestazioni e la qualità dell'elaborazione. Generalmente esse continuano ad elaborare l'informazione che ricevono in ingresso per pro-

durre i dati desiderati in uscita. Riassumendo il tipico processo di design che un progettista di sistemi embedded utilizza si possono evidenziare i seguenti compiti: individuare la soluzione hardware/software più idonea per raggiungere lo scopo, fare un'attività di profiling rispetto ai parametri applicativi non ancora stabiliti per ottenere le prestazioni dell'applicazione e infine selezionarne i valori che meglio si adattano alla situazione.

Il problema di questo approccio è che una volta che si esegue l'applicazione con i parametri determinati sulla piattaforma scelta, essi non possono essere cambiati. Se la situazione reale aderisse perfettamente con quella ipotizzata a design time durante tutto il periodo di esecuzione, non sussisterebbe alcun problema; tuttavia ci possono essere svariati motivi che portano i due scenari a essere differenti. Per esempio la piattaforma reale che è stata utilizzata ha prestazioni differenti rispetto a quella analizzata; una parte fisica del sistema potrebbe non operare più come da specifica; nel tempo si è sostituita una parte della piattaforma con un componente simile, ma non identico, avendo di conseguenza prestazioni differenti. Un ulteriore difetto che deriva dall'usare un solo insieme di parametri è che non si possono inseguire le dinamiche delle informazioni in ingresso dell'applicazione. Sicuramente in fase di design sono stati selezionati i parametri ottimali considerando i dati in ingresso più probabili, oppure facendone una media, ma se l'applicazione si trova in un contesto di alta variabilità, l'efficienza dei parametri utilizzati potrebbe essere messa in discussione.

Il framework ARGO[1] presentato in questo elaborato di tesi è la continuazione del lavoro descritto in [52] e ha lo scopo di fornire ad un'applicazione la capacità di adattarsi all'evolversi della situazione senza bisogno di coordinarsi con altri attori, proponendosi come una soluzione ai problemi appena esposti. Il framework è partizionato in due moduli distinti: una parte di monitoring e una decisionale. Il modulo di monitoring ha la funzione di osservare delle grandezze selezionate dallo sviluppatore dell'applicazione, come il tempo di elaborazione o la quantità di memoria utilizzata, e il compito di rendere disponibili delle statistiche sui dati raccolti. Il modulo è pensato per essere utilizzato anche singolarmente, fornendo la possibilità allo sviluppatore di imporre degli obbiettivi e verificare se vengono rispettati. Il framework fornisce allo sviluppatore tre monitor che si occupano di misurare il throughput dell'applicazione, i tempi di elaborazione e il consumo di memoria; la struttura dei monitor è stata pensata per un uso generale, per cui lo sviluppatore può facilmente creare un proprio monitor personalizzato per misurare delle caratteristiche che dipendono dell'applicazione, come l'errore

[1] Il nome ARGO è stato preso in prestito dalla mitologia greca. Argo era la nave che portò Giasone e gli Argonauti alla conquista del vello d'oro, una pelle dorata con particolari proprietà. Come Argo è stato il mezzo per il raggiungimento dell'obbiettivo della spedizione, il framework ARGO rappresenta un supporto al raggiungimento dei goal delle applicazioni.

di elaborazione, avendo la garanzia che si integri con le restanti funzionalità del framework.

La seconda parte del framework ha il compito di fornire all'applicazione la migliore combinazione di parametri nell'istante in cui è chiamata a decidere. Per poter riuscire nel suo scopo occorre che lo sviluppatore dichiari gli obiettivi che l'applicazione deve raggiungere e fornire la propria definizione di "migliore". In questo modo il framework utilizza le informazioni ottenute dai monitor per selezionare il miglior insieme di parametri, tra quelli individuati a design time, per adattarsi alla situazione osservata. Il framework offre la possibilità allo sviluppatore di modificare, aggiungere o rimuovere gli obiettivi dell'applicazione o di fornire una nuova definizione di migliore in qualsiasi momento durante l'esecuzione. Pensando all'utilizzo del framework anche in un contesto distribuito, lo sviluppatore può definire degli obiettivi che non influenzano direttamente la scelta dei parametri per l'applicazione, ma nel caso non fossero raggiunti, notificano l'applicazione permettendogli di intraprendere l'azione desiderata. Per esempio avvisare un dispositivo remoto che l'applicazione al momento è congestionata. Il framework inoltre introduce il concetto di stato per fornire allo sviluppatore la possibilità di utilizzare dei comportamenti distinti secondo richiesta. Per esempio un'applicazione di videosorveglianza può voler evitare di consumare energia se non rileva minaccie, ma quando inizia a rilevare un'anomalia può richiedere di usare la capacità computazionale necessaria per riconoscere l'eventuale tentativo di intrusione. In questo caso è possibile definire uno stato in cui l'applicazione normalmente utilizza vincoli sul consumo energetico e preferenze sulle prestazioni; mentre quando rileva l'anomalia usarne un altro che ha dei vincoli sulle prestazioni e preferenze sul consumo energetico.

Il lavoro di tesi è iniziato separando il framework ARGO dal Run-Time Resource Manager descritto in [39]. Successivamente si è svolto un profondo lavoro sull'Application-Specific Run-TIme Manager, rendendone l'utilizzo più trasparente e migliorando la capacità di adattamento che fornisce all'applicazione, mantenendo il framework generico e flessibile. Per ottenere una misura dell'overhead massimo introdotto, è stata creata un'applicazione di benchmark che ricrea le condizioni peggiori per tutte le operazioni più dispendiose. Per avere una stima dell'overhead che il framework introduce in un'applicazione generica, sono state usate delle applicazioni di benchmark appartenenti alla suite PARSEC [6]; in particolare sono state utilizzate le seguenti applicazioni:

**Swaptions** Utilizza il framework Heath-Jarrow- Morton per stabilire il valore di un portfolio di swaption

**Bodytrack** È una applicazione di computer vision che traccia la posa 3D di una persona utilizzando quattro telecamere distinte, senza l'ausilio di marcatori.

**x264** È un'applicazione che codifica un video nel formato H264.

L'elaborato di tesi è suddivisono in sette capitoli. I primi tre capitoli descrivono il contesto in cui si inserisce il framework sviluppato, i restanti capitoli descrivono il lavoro realizzato. Il secondo capitolo introduce i concetti di background che costituiscono i prerequisiti necessari alla descrizione del problema. Il terzo capitolo descrive lo stato dell'arte rispetto al problema dell'adatattività, distinguendo le tecniche utilizzate a Design Time e a Run Time. Il quarto capitolo descrive la metodologia utilizzata nella soluzione sviluppata, evidenziano le scelte di progetto effettuate. Il quinto capitolo descrive l'implementazione della soluzione, descrivendo l'architettura e gli attori utilizzati. Il sesto capitolo contiene i risultati sperimentali del framework realizzato. Il settimo capitolo contiene le conlusioni e sviluppi futuri del lavoro di tesi. Nell'appendice viene riportato un caso d'uso per mostrare l'integrazione del framework in un'applicazione generica, e il comportamento dinamico del framework in risposta alle situazioni più comuni.

# Capitolo 2

# Background

In questo capitolo capitolo vengono descritti brevemente i concetti necessari per comprendere pienamente il contesto e le problematiche in cui si inserisce il framework descritto nei capitoli successivi. Il framework sviluppato in questa tesi si colloca a valle del processo di design dei sistemi embedded. Prima di poter descrivere la soluzione proposta e lo stato dell'arte è quindi necessario fornire un'inquadratura generale partendo dalla definizione di sistemi embedded. Successivamente si discuterà in generale del processo di design, evidenziando le differenze rispetto allo sviluppo di una applicazione general-purpose, introducendo il concetto di punto di lavoro. In seguito verrà descritta l'algebra di Pareto, spiegando il motivo per cui viene utilizzata. Infine verrà chiarito il concetto di self aware computing e presentato il progetto 2PARMA e il framework BOSP.

## 2.1 Definizione di sistemi embedded

Citando la definizione data in [20], un sistema embedded è un sistema basato su microprocessori costruito per svolgere una funzione, o un insieme di funzioni, e non è progettato per permettere all'utente di modificare la sua programmazione. A differenza di un normale computer dove è possibile aggiungere funzionalità installando altro software, diventando per esempio un elaboratore di testo o un riproduttore multimediale a seconda della volontà dell'utente, un sistema embedded svolgerà solamente la funzione per cui è stato pensato. Per questo motivo è possibile raggiungere un livello di ottimizzazione molto superiore a quello raggiunto da un pc normale. Tipicamente un sistema embedded è soggetto a vincoli ben precisi, richiedendo per esempio dei risultati in tempo reale. Considerando che spesso è un dispositivo portabile, in generale alimentato a batterie, qualsiasi funzionalità non necessaria è considerata uno svantaggio in termini di costo di produzione e consumo energetico.

## 2.2 Design dei sistemi embedded

Il processo di design di un'applicazione descritto dalle normali tecniche di ingegneria del software è parecchio differente rispetto al design dei sistemi embedded. Un'applicazione classica verrà eseguita da un'architettura general-purpose disponibile normalmente in commercio, quindi il processo di design è finalizzato a produrre solamente del codice. In un sistema embedded il processo di design comprende anche la progettazione dell'hardware necessario ad eseguire il software. Lo spazio delle possibili soluzioni aumenta significativamente rispetto a quello considerato nelle tecniche di ingegneria del software.

Per selezionare quella più adatta esistono due strade differenti: utilizzare un design Full-Custom, oppure un approccio Platform-Based. Nel primo caso il progettista ha il completo controllo sulle caratteristiche hardware e software che può adottare. In questo modo dispone virtualmente di illimitate soluzioni per raggiungere lo scopo prefissato. È possibile infatti determinare il numero e la natura dei processori utilizzati, se general-purpose o special-purpose, scegliere se utilizzare degli acceleratori hardware per velocizzare alcune operazioni e determinare il tipo di comunicazione che lega i componenti presenti nell'architettura. Una volta stabilito il sistema è possibile progettare e sintetizzare tutti i componenti fisici fino al livello di silicio, oltre a produrre il codice dell'applicazione che viene eseguita. Lo svantaggio di utilizzare questo approccio risiede nel fatto che il tempo di progettazione si allunga notevolmente. Tipicamente un sistema embedded è pensato per essere prodotto in massa, quindi il costo di progettazione è un aspetto fondamentale. Considerando inoltre che il mercato è guidato da standard, per esempio nei protocolli di comunicazione o nei formati multimediali, la tendenza è di abbandonare il design Full-Custom.

Come descritto in [28], nel design Platform-Based viene richiesto al progettista di porsi ad un livello di astrazione più alto in cui il confine tra hardware e software è più sottile. La piattaforma è un'architettura parzialmente definita, in cui è possibile mappare le funzionalità richieste (cosa il sistema deve fare) nell'architettura utilizzata (come il sistema deve fare). Il prodotto fisico realizzato è una particolare istanza di una particolare piattaforma. In questo modo il risultato non è dato da un assemblaggio di blocchi funzionali, ma è derivato da una "famiglia" di architetture pensate per risolvere la stessa classe di problemi del compito richiesto. Utilizzare questo tipo di design fornisce molti meno gradi di libertà rispetto a sintetizzare una soluzione hardware completa, ma comporta numerosi vantaggi. Una piattaforma è pensata per utilizzare le soluzioni migliori che risolvono la classe di problemi considerata, inoltre non comprende solamente la descrizione dell'hardware, ma include anche del software per gestirne le risorse in fase di Run-Time e dei simulatori che permettono di ottenere delle stime molto accurate delle dimensioni, prestazioni e consumo del prodotto finale. Dal punto di vista

economico, utilizzare l'approccio Platform-Based riduce considerevolmente il costo di design e accorcia il time to market del prodotto creato.

### 2.2.1 Definizione di punto di lavoro

Il processo di design di un sistema embedded ha lo scopo di trovare, fra tutte le possibili soluzioni del problema che si vuole risolvere, quella migliore. Nel caso dei sistemi embedded occorre caratterizzare tutti gli aspetti dell'applicazione e della piattaforma. Possiamo quindi definire Design Space, l'insieme delle possibili soluzioni. Un punto di tale insieme contiene i valori di tutti i parametri dell'architettura e dell'applicazione utilizzati in una determinata soluzione. Utilizzando il simulatore è possibile valutarne le qualità che interessano al progettista, come per esempio il throughput e il consumo energetico. Il valore della misura di tali grandezze definiscono la prestazione generale di quella configurazione di parametri. Il termine metrica o quantità utilizzato di seguito si riferisce a tali grandezze. In generale un punto di lavoro è l'unione di queste due informazioni, ovvero i valori dei parametri che caratterizzano la soluzione e la misura delle prestazioni che tale punto permette di ottenere.

La definizione di punto di lavoro appena fornita si basa su una generalizzazione troppo elevata rispetto alla natura dei parametri che compongono una soluzione, occorre quindi specializzare tale definizione. Una parte di essi definiscono alcune caratteristiche dell'architettura; come per esempio il numero di processori da implementare e la natura e dimensione della cache. Ovviamente tali parametri diventano delle costanti durante tutta la fase di Run-Time dell'applicazione. Altri parametri definiscono la frazione di risorse della piattaforma che il software ha a disposizione, come per esempio il numero di processori effettivamente utilizzati e la quantità di memoria che può allocare. Tali risorse possono essere cambiate in fase di Run-Time da un gestore di risorse, quindi i valori dei parametri uniti alla misura delle prestazioni definiscono un Working Mode. L'utilizzo di tali punti verrà descritto nel capitolo successivo. L'ultima classe di parametri sono quelli che caratterizzano l'applicazione software e quindi assumono significati molto differenti dipendenti dal contesto; possono appartenere a questa categoria il numero di thread utilizzati per parallelizzare il lavoro, la risoluzione del video da codificare o il numero di particelle da utilizzare in un filtro particellare. Tali parametri possono essere variati in fase di Run-Time dall'applicazione stessa, quindi i valori di tali parametri uniti alle prestazioni osservate definiscono un Operating Point (detti anche Dynamic Knobs [24]). Quest'ultima classe di punti di lavoro è quella utilizzata dal framework oggetto dell'elaborato di tesi.

In fase di design è necessario considerare il maggior numero di soluzioni possibile, per questo si utilizzano i punti di lavoro generici definiti inizialmente. Il processo che si occupa di esplorare tale spazio e di misurarne le

prestazioni è chiamata Design Space Exploration. Le problematiche e le tecniche utilizzate in tale operazione verranno spiegate in dettaglio nel capitolo successivo.

## 2.3 Algebra di Pareto

Nella definizione di punto di lavoro non si è discusso di un criterio per stabilire se un punto è migliore di un altro. Se le prestazioni di un punto di lavoro fossero definite solo da una metrica, si supponga la velocità di elaborazione, sarebbe possibile effettuare un ordinamento dei punti forniti dalla Design Space Exploration e l'individuazione del punto di lavoro ottimo si ridurrebbe ad una semplice ricerca del massimo fra un insieme di elementi. Se questo può accadere nel caso di un'applicazione general-purpose, nello sviluppo di sistemi embedded purtroppo non si è mai così fortunati: spesso le prestazioni sono composte da diverse metriche in contrasto fra di loro. Si supponga di aggiungere alla definizione di prestazioni dell'esempio precedente anche la qualità di elaborazione, in questo caso la ricerca di un punto ottimo non è più banale. Tipicamente più la qualità di elaborazione è elevata, più il sistema impiega tempo ad eseguirla e viceversa. In questo caso non è più possibile un ordinamento stretto dei punti rispetto ad entrambe le metriche, ma si deve giungere ad un compromesso.

Se si vuole aggiungere alla definizione di prestazioni dell'esempio anche il consumo energetico, la situazione si complica maggiormente: il compromesso che si era trovato considerando solamente velocità e qualità di elaborazione implica verosimilmente l'utilizzo di molte risorse. Il consumo energetico quindi introduce una nuova dimensione di conflittualità perché induce il sistema ad utilizzare meno risorse possibili, obbligando di conseguenza a diminuire le restanti metriche. In generale la definizione di prestazione è composta da più metriche correlate fra loro. Alcune di esse avranno un rapporto di conflittualità, altre di dipendenza, come velocità di elaborazione e throughput.

La ricerca del punto di lavoro ottimale non può quindi selezionare un singolo punto di lavoro basandosi solo sulla valutazione delle prestazioni, ma deve selezionare un insieme di punti che costituiscono un buon compromesso fra tutte le metriche; un qualsiasi punto appartenente a questo insieme può essere definito ottimo. Per esprimere questa idea, l'algebra di Pareto introduce il concetto di dominazione e in [34] è presente la formalizzazione matematica, di seguito viene riportata la definizione.

**Definizione 1** (Quantità)**.** *Una quantità è un insieme $Q$ con un ordine parziale $\preceq_Q$. Se la quantità è chiara dal contesto questo ordine è indicato solo con $\preceq$. Per semplificare l'esposizione si assume che il problema sia di minimizzazione e quindi che valori piccoli siano preferibili a valori grandi. La variante irriflessiva di $\preceq_Q$ è denotata con $\prec_Q$.*

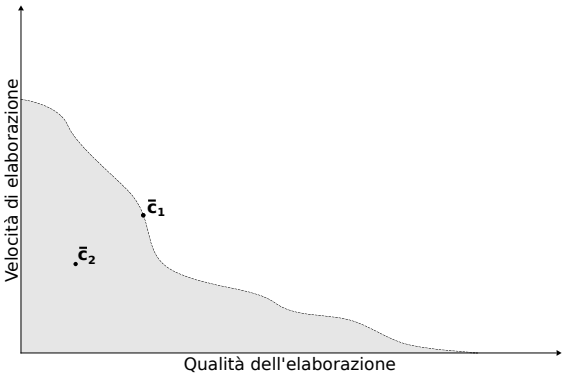



Figura 2.1: Curva di Pareto

**Definizione 2** (Spazio di configurazione)**.** *Uno spazio di configurazione $S$ è il prodotto cartesiano $Q_1 \times Q_2 \times \ldots \times Q_n$ di un numero finito di quantità.*

**Definizione 3** (Configurazione)**.** *Una configurazione $\bar{c} = (c_1, c_2, \ldots, c_n)$ è un elemento dello spazio di configurazione $Q_1 \times Q_2 \times \ldots \times Q_n$. Verranno usati $\bar{c}(Q_k)$ o $\bar{c}(k)$ per denotare $c_k$.*

**Definizione 4** (Dominanza)**.** *Se $\bar{c}_1, \bar{c}_2 \in S$, allora $\bar{c}_1 \preceq \bar{c}_2$ se e solo se per ogni quantità $Q_k$ di $S$, $\bar{c}_1(Q_k) \preceq_{Q_k} c_2(Q_k)$. Se $\bar{c}_1 \prec \bar{c}_2$, allora $\bar{c}_1$ è detto dominante rispetto a $\bar{c}_2$.*

**Definizione 5** (Insieme dominante)**.** *Un insieme $C_1$ di configurazioni dello spazio di configurazione $S$ domina un insieme $C_2$ facente parte dello stesso spazio di configuraizone se e solo se per ogni $\bar{c}_2 \in C_2$ c'è qualche $\bar{c}_1 \in C_1$ tale che $\bar{c}_1 \preceq \bar{c}_2$. Questa proprietà è denotata come $C_1 \prec C_2$.*

**Definizione 6** (Pareto equivalente)**.** *Due insiemi di configurazioni $C_1$ e $C_2$ da uno spazio di configurazione $S$ sono Pareto equivalenti se se e solo se dominano l'un l'altro: $C_1 \preceq C_2$ e $C_2 \preceq C_1$. Questa proprietà è denotata come $C_1 = C_2$.*

Il significato di dominanza è più intuitivo da comprendere da un punto di vista geometrico. Per semplicità si consideri la definizione di prestazione formata da due metriche dell'esempio precedente. Si supponga di aver effettuato la Design Space Exploration e che il grafico delle prestazioni misurate sia rappresentato in Figura 2.1. In particolare tutti i punti individuati sono contenuti nella zona colorata di grigio. I punti dominanti, ovvero quelli che offrono un compromesso ottimale, sono esattamente quelli che giacciono sul limite tratteggiato di tale regione, denominata curva di Pareto. Se si considera un punto generico interno, per esempio quello indicato nella figura con $\bar{c}_2$, ne esiste sempre almeno un altro appartenete alla frontiera, indicato con $\bar{c}_1$, che offre prestazioni migliori rispetto ad entrambe le metriche. Utilizzare la

configurazione di parametri $\bar{c}_2$ rispetto a $\bar{c}_1$comporta solamente un peggioramento. Questo ragionamento viene facilmente esteso al caso $n$-dimensionale, dove $n$ è il numero di metriche considerate per caratterizzare le prestazioni di un punto di lavoro.

## 2.4 Self aware computing

La complessità dei sistemi utilizzati in informatica sta aumentando costantemente, di conseguenza determinare a priori le caratteristiche necessarie per permettere al sistema di operare in maniera ottima durante tutta la fase di Run-Time diventa sempre più complicato. Da questo bisogno nasce l'esigenza del self aware computing, ovvero cercare di rendere i sistemi più autonomi, adattandosi all'evoluzione della situazione. In [11] viene introdotto il concetto di Autonomic Computing per semplificare l'amministrazione di sistemi informatici complessi, utilizzando il paradigma self-X: auto-recupero, auto-organizzazione, auto-ottimizzazione e auto-protezione. Nel caso ideale, l'onere del progettista si riduce a specificare delle politiche di alto livello che definiscono come il sistema può cambiare per rispettare gli obbiettivi prefissati. Per poter ottenere tali risultati, in generale, un sistema deve possedere le seguenti proprietà:

**Adattabilità** Il concetto espresso da questa caratteristica è la capacità di poter modificare un comportamento osservabile dell'applicazione, tale cambiamento deve essere originato dall'applicazione stessa.

**Consapevolezza** Questa caratteristica è un prerequisito per l'adattabilità, in quanto si riferisce alla possibilità dell'applicazione di osservare l'ambiente o il proprio stato, modello, etc. nel caso di auto-consapevolezza. Un particolare caso importante è il monitoraggio, che permette la scoperta dell'errore commesso nella rappresentazione di uno stato e di conseguenza utilizzare l'adattamento per cercare di porvi rimedio.

**Dinamicità** Se l'adattabilità stabilisce la possibilità di un sistema di modificare il proprio comportamento in generale, la dinamicità esprime la capacità di aggiungere, rimuovere o modificare funzionalità (anche solo interne) durante la fase di Run-Time, senza dover interrompere il flusso di lavoro.

**Autonomia** Nessuna delle precedenti proprietà avrebbe senso se l'applicazione non avesse l'autonomia di prendere delle decisioni senza un intervento esterno. Il meccanismo più semplice a cui pensare è una serie di regole if_then_else che ne modificano il comportamento.

**Robustezza** Misura la capacità del sistema di comportarsi in modo ragionevole a fronte di situazioni impreviste. Questa caratteristica è

fondamentale e viene considerata in praticamente tutti i metodi di design.

**Mobilità** Interessa tutte le parti di un sistema: dai servizi offerti, ai dispositivi utilizzati fino al sistema stesso. Permette la ricerca e utilizzo immediato di nuove risorse, trasferimento di funzionalità tra dispositivi, etc. in maniera automatica.

**Tracciabilità** Indica la possibilità di effettuare una mappatura non ambigua tra la parte logica e quella fisica del sistema, in modo da poter favorire l'adattabilità del sistema: prendere decisioni sul modello astratto del sistema, implica la possibilità di poterle effettuare su quello fisico.

Fino ad ora lo scopo del self aware computing sembra incentrato sul miglioramento della configurazione del sistema. Un'ulteriore ragione che favorisce lo sviluppo di tale caratteristica è la variabilità della richiesta di un servizio. Se non fosse presente nessun meccanismo di adattamento, la presenza di picchi di richieste costringe il sistema a due scelte: utilizzare una configurazione ottimale per la maggior parte del tempo, ma inadatta a smaltire un'elevata mole di lavoro; oppure utilizzare un'altra configurazione che garantisca risorse sufficienti a coprire i picchi di richieste, ma inefficiente nella maggior parte dei casi. In [43] viene trattato questo problema, mostrando come l'adattabilità aumenti sia l'efficienza, che la qualità del servizio offerto.

## 2.5 Progetto 2PARMA

La tendenza principale nelle architetture computazionali è quella di integrare in un unico chip un insieme eterogeneo di core di elaborazione collegati tra loro da una Network on Chip (NoC). Data questa tendenza tecnologica ci si aspetta di passare nei prossimi anni da architetture multicore ad architetture manycore. Per far fronte al crescente numero di core integrati in un singolo chip è necessario un ripensamento globale all'approccio con il quale si sviluppano software e hardware.

Il progetto 2PARMA si concentra sulle architetture parallele e scalabili chiamate Many-Core Computing Fabric (MCCF). La classe MCCF permette di aumentare le prestazioni, la scalabilità e la flessibilità di progettazione solo se affiancata da opportune tecniche di programmazione parallele. 2PARMA cerca di superare questo limite sfruttando le architetture manycore focalizzandosi su un modello di programmazione parallelo che utilizza un'architettura eterogenea di core. Questo progetto di ricerca ha i seguenti obiettivi:

- Migliorare le performance del software proponendo un modello di programmazione che sfrutti il parallelismo hardware.

- Analizzare la relazione potenza/prestazioni e offrire un Run-Time Manager per la loro gestione e ottimizzazione.
- Migliorare l'affidabilità del sistema offrendo la possibilità di riconfigurarlo dinamicamente.
- Aumentare la produttività del processo di sviluppo di software parallelo usando tecniche che permettono l'estrazione semi-automatica del parallelismo e estendendo il modello OpenCL ai sistemi paralleli.

Il Run-Time Resource Manager (RTRM) progettato per il progetto 2PARMA è chiamato Barbeque [39]; fa parte del progetto open-source che ha come acronimo Barbeque Open Source Project (BOSP) [1]. Questo manager è stato sviluppato per funzionare su diverse architetture: l'architettura 2PARMA, architetture di tipo Simultaneous MultiThreading (SMT) e anche la piattaforma Android. Barbeque basa il suo funzionamento su alcuni file chiamati ricette che forniscono ad esso la conoscenza necessaria sulle modalità di funzionamento di ciascuna applicazione. Le ricette rappresentano un vero e proprio contratto con il resource manager e definiscono la quantità massima di risorse utilizzabili per ogni Working Mode di un'applicazione. Barbeque esercita un controllo su queste risorse e provvede a limitarle secondo le quantità pattuite nelle ricette. Il modello di programmazione di Barbeque si compone di un interfaccia con la quale è possibile instaurare un dialogo tra l'applicazione e il manager permettendo lo scambio di informazioni tra i due componenti. Questa comunicazione è necessaria a garantire un'efficace gestione delle risorse e dei vincoli di qualità specificati dall'applicazione.

# Capitolo 3

# Stato dell'arte

In questo capitolo vengono illustrati i due grandi argomenti che costituiscono lo stato dell'arte rispetto al lavoro svolto in questa tesi. In primo luogo verranno descritte le metodologie applicate in fase di design, evidenziando il procedimento per individuare gli Operating Point, utilizzati come punto di partenza dal framework ARGO. Il secondo argomento trattato è il Run-Time management, evidenziando le tecniche utilizzate nei diversi livelli di astrazione presi in considerazione.

## 3.1 Tecniche utilizzate a Design-Time

Come mostrato nel capitolo precedente, la fase di design si può astrarre come un problema di ottimizzazione multi-dimensionale multi-obbiettivo. Se la finalità della fase di design consiste nel selezionare un singolo punto di lavoro che ottimizza i requisiti e preferenze imposte dal progettista, essa si può ricondurre al problema dello zaino ben noto in letteratura. La formulazione classica prevede la massimizzazione/minimizzazione di una funzione obbiettivo che descrive le preferenze del progettista, soggetta a vincoli sui valori che possono assumere i parametri e sulle prestazioni che si desiderano raggiungere. Per esempio nel caso si desideri massimizzare la qualità dell'elaborazione, imponendo dei vincoli sulla velocità di elaborazione e sul consumo energetico. La risoluzione di tale problema è un compito NP difficile, in particolare la complessità è esponenziale rispetto al numero di insiemi di configurazioni[32]. Per questo motivo molti algoritmi reali si basano su euristiche per velocizzarne la risoluzione [19, 10] e poter essere impiegate in fase di Run-Time, come vedremo nella prossima sezione.

Un prerequisito necessario al framework ARGO per poter fornire all'applicazione la capacità di adattarsi, è avere a disposizione un modello per predire le prestazioni del sistema. Per poter decidere un cambiamento razionale, occorre infatti prevederne le conseguenze [43]. La definizione di Operating Point fornita nel capitolo precedente include le informazioni nec-

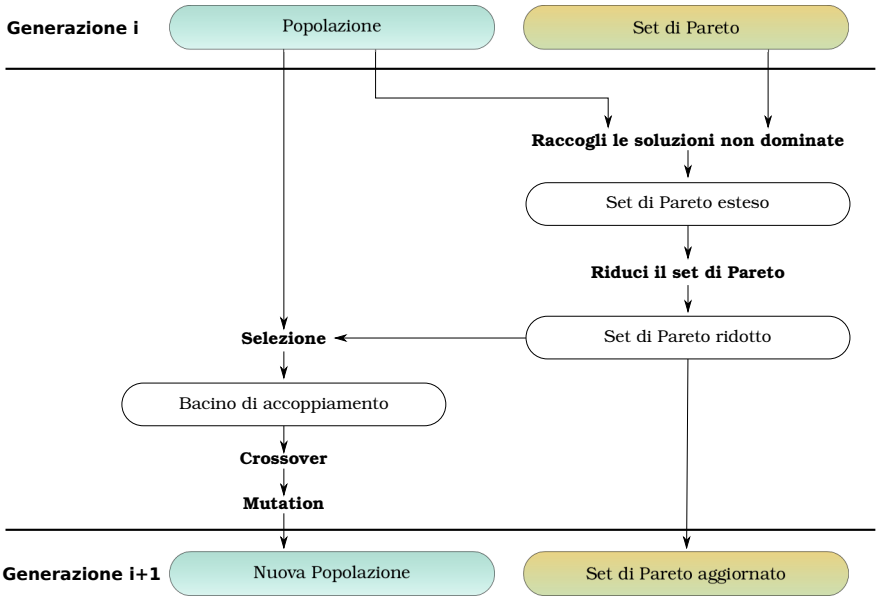



Figura 3.1: Processo evolutivo in SPEA

essarie a poter modificare il comportamento (i valori dai parametri applicativi) e prevederne le conseguenze (la misura delle prestazioni) direttamente dell'applicazione.

Per queste ragioni in questa sezione descriveremo le tecniche utilizzate a Design-Time per identificare i punti di lavoro che compongono la curva di Pareto, ponendo particolare enfasi rispetto a quella utilizzata nel lavoro di tesi precedente.

### 3.1.1 Algoritmi evolutivi

Invece di utilizzare convenzionali tecniche di ottimizzazioni basate sull'utilizzo del gradiente o sul metodo del simplesso, in letteratura sono presenti degli algoritmi genetici in grado di individuare i punti di lavoro che delineano la curva di Pareto. Il primo lavoro presentato è Strength Pareto Evoltionary Algorithm, descritto in [14]. L'idea base è quella di codificare i valori dei parametri dei punti di lavoro nella popolazione, utilizzando le prestazioni raggiunte per determinare quali siano quelli non dominati. Utilizzando una specifica funzione di fitness, l'evolversi delle generazioni favorisce la convergenza verso la vera curva di Pareto di questi ultimi punti.

Nella Figura 3.1 viene rappresentato il processo di evoluzione dalla generazione $i$ alla generazione $i + 1$. Il procedimento si divide in due passi differenti: il primo è aggiornare il Set di Pareto che contiene tutti i punti di lavoro non dominati, il secondo consiste nell'effettiva evoluzione della popolazione.

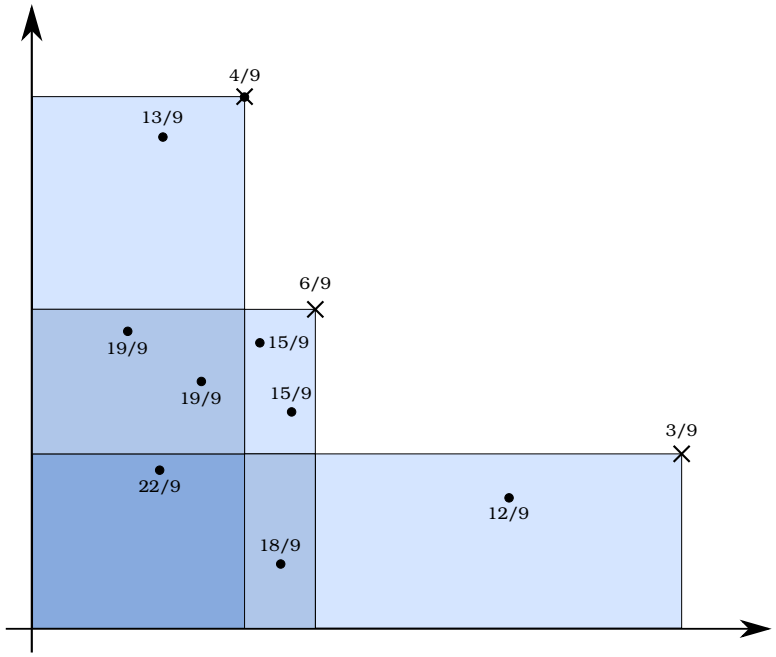



Figura 3.2: Esempio di attribuzione del valore di fitness

Come rappresentato in figura i punti di lavoro che compongono il Set di Pareto della generazione $i$-esima sono separati da quelli che formano la popolazione. Per ottenere il Set di Pareto esteso si unisco i due insiemi e attraverso l'uso della logica di Pareto (descritta nel capitolo precedente) si scartano tutti i punti dominati, in questo modo rimangono solo quelli che delineano la curva di Pareto della generazione $(i + 1)$-esima. Il numero di questi punti può essere elevato, per cui si effettua un'operazione chiamata clusterizzazione che ha l'obbiettivo di raggruppare i punti di lavoro simili in modo da ridurne il numero, riducendo di conseguenza alcuni effetti collaterali che potrebbero compromettere la convergenza verso la vera curva di Pareto. Il nuovo insieme di punti non dominati viene definito come Set di Pareto ridotto.

Generalmente l'utilizzo di algoritmi evolutivi tende a favorire un unico tipo di soluzione, quindi per evitare di far collassare la curva cercata in unico punto viene introdotto il concetto di strength come un numero reale $s \in [0, 1)$ assegnato ad ogni soluzione presente nel Set di Pareto; tale numero è proporzionale al numero di membri della popolazione che tale soluzione domina e coincide con il loro valore di fitness. Per gli elementi che compongono la soluzione la funzione di fitness è calcolata come la sommatoria della strength dei punti che lo domina a cui viene aggiunto uno. Quest'ultimo accorgimento ha lo scopo di favorire nella selezione per l'accoppiamento i punti dominanti.

Per maggiore chiarezza si consideri un esempio in cui vengono utilizzate

due metriche per definire le prestazioni. In Figura 3.2 viene mostrato il grafico che raffigura la popolazione (ogni elemento è indicato dal pallino) e i punti che sono inclusi nel Set di Pareto (rappresentati da una croce), a fianco è presente il valore di fitness calcolato per ciascun elemento. Nella figura vengono evidenziati i sub-spazi originati dai punti dominanti, in questo modo l'elemento della popolazione che appartiene a quello dominato maggiormente ha il valore di fitness più alto: $\frac{22}{9}$ ( che è ottenuto da $\frac{4}{9} + \frac{6}{9} + \frac{3}{9} + 1$ come descritto precedentemente). In questa formulazione avere un valore di fitness basso è considerato vantaggioso. Dall'esempio è possibile notare come i sub-spazi densamente popolati abbiano un valore di fitness maggiore rispetto a quelli scarsamente popolati, questo favorisce la diversificazione dei punti.

La procedura di selezione per l'accoppiamento utilizzata è un binary tournament selection, a cui partecipano gli elementi che appartengo sia alla popolazione che al Set di Pareto ridotto appena ottenuto in modo che l'elemento con minor fitness ha maggiori probabilità di essere selezionato. Vengono usati normali operatori di crossover e mutation. Con il primo operatore si sfruttano le soluzioni già trovate per cercare di ottenere migliori individui, con la mutation si effettua la Design Space Exploration.

Come tutti gli algoritmi evolutivi non è garantita la convergenza del Set di Pareto ottenuto verso la curva reale, tuttavia generalmente sono più veloci. Questa metodologia è stata ripresa e migliorata: in [36] viene introdotto l'operatore di immunizzazione con lo scopo di confinare alcune degenerazioni evolutive, in [47] viene ripreso l'intero algoritmo aumentando le prestazioni e introducendo la minimizzazione della dimensione dell'applicazione prodotta come secondo obbiettivo affiancato a quelli riguardanti la funzionalità.

Un ulteriore esempio di Algoritmo Evolutivo è descritto in [27]. Nel lavoro descritto precedentemente per mantenere la diversità nelle soluzioni veniva utilizzata la funzione di fitness, in questa tecnica viene utilizzato l'operatore di crowding-distance per creare delle nicchie evolutive. In questo modo è possibile definire una densità della popolazione che appartiene a ciascuna nicchia. Utilizzando tale metodo è possibile definire l'operazione di crowded-comparison per stabilire un ordine tra due soluzioni in base alla sua capacità di dominare le altre soluzioni (utilizzando l'algebra di Pareto) e nel caso di parità si favoriscono soluzioni appartenenti a nicchie scarsamente popolate (utilizzando la crowding-comparison).

Viene inoltre introdotto il concetto di constraint-domination per offrire la possibilità di specificare dei vincoli che una soluzione deve soddisfare per essere considerata valida. In particolare la soluzione $i$ domina la soluzione $j$ se:

1. La soluzione $i$ è valida e $j$ non lo è.

2. La soluzione $i$ e $j$ sono entrambe non valide, ma la violazione totale dei vincoli di $i$ è minore di quella di $j$.

3. La soluzione $i$ e $j$ sono entrambe valide, ma $i$ domina (usando l'algebra di Pareto) la soluzione $j$.

### 3.1.2 Pareto Ant Colony Optimization

L'Ant Colony Optimization imita il comportamento esibito da formiche vere quando sono in cerca di cibo. Le formiche comunicano le notizie riguardo le fonti di cibo attraverso delle quantità di essenze chiamate feromoni, che secernano mentre si muovono. Nel tempo il percorso diretto più corto tra il cibo e il nido verrà usato più spesso di uno lungo. Di conseguenza sarà marcato da una maggiore quantità di feromoni che a sua volta attrarrà più formiche. Le formiche artificiali non solo imitano il comportamento descritto, ma utilizzando le informazioni fornite da specifiche euristiche del problema, sono in grado di esprimere preferenze. In [18] vine provata la convergenza di un generico algoritmo basato su Ant System Scheme.

La formalizzazione di questa metodologia è descritta in [26], che definisce il problema di ottimizzazione multi-dimensionale multi-obbiettivo dalla quadripletta $(X, D, C, F)$, dove $X$ è un vettore $n$-dimensionale che rappresenta le $n$ variabili decisionali, i.e $X = (x_1, \ldots, x_n)$; $D$ è un vettore con $n$ elementi, dove ciascuno di essi rappresenta il dominio di ciascuna variabile, i.e. $D = (d_1, \ldots, d_n)$; $C$ è un set di vincoli rispetto $X$, per esempio un set di relazioni che impediscono l'assegnazione simultanea di certi valori alle variabili decisionali; infine $F$ è un vettore di $m \geq 2$ funzioni obbiettivo, $F = (f_1(X), f_2(X), \ldots, f_m(X))$, senza perdita di generalità si assuma che tali funzioni debbano essere minimizzate.

Lo spazio delle possibili soluzioni, indicato con $E(X, D, C)$, è il set di tutti i vettori dei valori $v \in D$ che soddisfano tutti i vincoli di $C$. Su tale spazio, utilizzando l'algebra illustrata nel capitolo precedente, si definisce la relazione di ordine parziale utilizzando il concetto di dominanza di Pareto. L'obbiettivo di tale è algoritmo è quindi quello di trovare il set di tutte le soluzioni non dominate, i.e $\{v \in E(X, D, C) | \forall v' \in E(X, D, C), \neg(v' \prec v)\}$.

Per trovare una soluzione, le formiche utilizzano un grafo di costruzione $G = (V, E)$ la cui definizione dipende dal problema che si vuole risolvere, come la scelta se associare i feromoni ai vertici o agli spigoli dello stesso. L'idea base è che la soluzione venga costruita selezionando dei vertici di tale grafo. In particolare ad ogni iterazione viene scelto un vertice di $G$ fra i possibili candidati $Cand$; esso viene aggiunto alla soluzione $S$ e il set dei possibili candidati viene aggiornato rimuovendo i vertici che violano i vincoli di $C$. Prima di spiegare il procedimento per selezionare un vertice occorre definire tre concetti:

**Scia di feromoni** La quantità di feromoni lasciata sopra un componente rappresenta l'esperienza passata della colonia rispetto alla scelta di inclusione o meno dello stesso. Nel caso siano presenti diverse funzioni

obbiettivo in $F$, occorre decidere in che modo organizzare la struttura di feromoni. Il primo metodo consiste nel raggruppare tali funzioni e utilizzare un'unica struttura di feromoni. Il secondo metodo prevede l'associazione di un formicaio ad ogni obbiettivo e usando una struttura di feromoni per ogni colonia.

**Soluzioni da premiare** Quando vengono aggiornati i valori delle scie di feromoni, occorre stabile come distribuire tale quantità. Una possibilità consiste nel premiare tutte le soluzioni non dominate. Rimane da decidere se premiare tutte quelle presenti nel Set di Pareto, oppure solamente quelle individuate nel ciclo corrente. Una seconda possibilità è quella di premiare le soluzioni che trovano i migliori valori per ogni criterio nel ciclo corrente.

**Fattori euristici** L'idea base di una euristica in questo contesto è fornire delle indicazioni rispetto alla qualità di una scelta. In generale per definire un fattore euristico possono esserci due strategie. Nella prima si considera un'aggregazione di differenti obbiettivi in una singola informazione euristica. Nella seconda strategia si considera ogni obbiettivo separatamente, tipicamente in questo caso c'è una colonia differente per ogni obbiettivo.

Dalla definizione dei concetti precedenti si è visto che si può parametrizzare questo algoritmo rispetto al numero di colonie di formiche $\#Col$ e al numero di strutture di feromoni $\#\Gamma$. Di seguito si indicherà con la lettera $c$ una generica colonia appartenente all'insieme delle colonie.

Il vertice $v_i$ viene aggiunto casualmente alla soluzione $S$ dalle formiche della colonia $c$ con probabilità $p_S^c(v_i)$ definita come segue:

$$p_S^c(v_i) = \frac{[\tau_S^c(v_i)]^\alpha \cdot [\eta_S^c(v_i)]^\beta}{\sum_{v_j \in Cand}[\tau_S^c(v_i)]^\alpha \cdot [\eta_S^c(v_i)]^\beta}$$

dove $\tau_S^c(v_i)$ e $\eta_S^c(v_i)$ sono rispettivamente i fattori di feromone e di euristica del vertice candidato $v_i$, e $\alpha$ e $\beta$ sono due parametri che ne determinano la relativa importanza. La definizione di questi due parametri dipendono dal problema e dai parametri $\#Col$ e $\#\Gamma$.

Una volta che tutte le formiche hanno costruito la loro soluzione, vengono aggiornate le scie di feromoni distribuendo tale quantità sui componenti da premiare. Più precisamente la quantità $\tau^i(c)$ della $i$-esima struttura di feromoni versata sul componente $c$ è aggiornata utilizzando la seguente formula:

$$\tau^i(c) \leftarrow (1-p) \cdot \tau^i(c) + \Delta\tau^i(c)$$

dove $p$ è il fattore di evaporazione $\in (0,1)$, e $\Delta\tau^i(c)$ è la quantità di feromone aggiunta al componente. Anche questa definizione dipende dai

parametri $\#Col$ e $\#\Gamma$. Di seguito viene fornito un esempio di scelta di questi parametri dove $\#Col = m + 1$ e $\#\Gamma = m$ dove $m = |F|$ è il numero di obbiettivi da ottimizzare.

L'idea è quella di utilizzare un formicaio per ogni obbiettivo singolo, utilizzando la propria struttura di feromoni; un ulteriore colonia di formiche viene utilizzata per ottimizzare tutti gli obbiettivi. In questo caso abbiamo le seguenti impostazioni:

**Fattore di feromone** Il fattore di feromone $\tau_S^i(v_j)$ considerato dalla $i$-esima colonia, che ha lo scopo di minimizzare la $i$-esima funzione obbiettivo $f_i$, è definito rispetto alla $i$-esima struttura di feromoni. Il fattore di feromone $\tau_S^{m+1}(v_j)$ considerata dalla colonia aggiuntiva è il fattore di feromone $\tau_S^r(v_r)$ della $r$-esima colonia, dove $r$ è scelto casualmente. In questo modo, ad ogni passaggio di costruzione di $S$, si sceglie casualmente un obbiettivo da minimizzare.

**Fattore euristico** Il fattore euristico $\eta_S^i(v_j)$ considerato dalla $i$-esima colonia, che ha lo scopo di minimizzare la $i$-esima funzione obbiettivo $f_i$, è la $i$-esima informazione euristica definita da $\eta^i$. Il fattore euristico $\eta_S^{m+1}(v_j)$ della colonia multi-obbiettivo è la somma delle informazioni euristiche associate a tutti gli obbiettivi, i.e. $\eta_S^{m+1}(v_j) = \sum_{i=1}^{m} \eta_S^i(v_j)$.

**Aggiornamento dei feromoni** Sia $S^i$ la migliore soluzione della $i$-esima colonia che minimizza $f_i$ nel ciclo corrente, inoltre si indichi con $S_{best}^i$ la soluzione dell'$i$-esima colonia che minimizza $f_i$ rispetto a tutte le soluzioni trovati, a partire dal primo ciclo (includendo l'attuale). La quantità di feromone depositata sopra il componente $c$ dell'$i$-esima struttura di feromoni è definita come:

$$\Delta\tau^i(c) = \begin{cases} \frac{1}{1+f_i(S^i)-f_i(S_{best}^i)} & c \in S^i \\ 0 & altrimenti \end{cases}$$

La colonia multi-obbiettivo mantiene un set di soluzioni: una soluzione migliore per ogni obbiettivo. Depositerà feromoni su ogni struttura di feromoni, per ogni componente, utilizzando la formula sopra descritta.

L'utilizzo di questa tecnica permette quindi di effettuare una Design Space Exploration cercando di utilizzare solamente soluzioni possibili, utilizzando i feromoni e l'euristiche per guidare il processo di esplorazione del dominio verso la vera curva di Pareto.

### 3.1.3 Response Surface-Based Pareto Iterative Refinement (ReSPIR)

Il punto di partenza utilizzato da questa metodologia parte da una preliminare valutazione dei parametri significativi da usare per l'esecuzione delle

varie simulazioni e nel processo di decisione di quante e quali configurazioni utilizzare effettivamente. Questa fase è chiamata Design of Experiments (DoE) e verrà dettagliata più approfonditamente nel resto della sezione.

Il passo logico successivo da effettuare è la Design Space Exploration (DSE), dove viene eseguita l'applicazione teoricamente in tutti i punti dello spazio dei parametri definiti dalla DoE, con lo scopo di delineare la curva di Pareto. Successivamente, una fase di analisi di questi punti verrà effettuata tramite tecniche di pareto-filtering e clusterizzazione.

Un Design of Experiment, consiste nella pianificazione del layout dei punti da selezionare nello spazio di configurazione per massimizzare le informazioni sul Design Space. I punti ottenuti sono quelli che prenderanno parte alle simulazioni. Come indicato in [16] e in [41], questa fase è molto importante in quanto la campionatura statistica dello spazio di progettazione può influire notevolmente sul tempo di simulazione con una trascurabile perdita di significatività del risultato finale. Una scelta corretta di un piccolo sottogruppo, anche solo il 5% dei punti del design space, insieme alle tecniche di Design Space Exploration può caratterizzare il sistema complessivo con una precisione del 95%. Utilizzando strategie standard presenti nell'ambito della statistica e proposte in [16], si considerino quattro diversi possibili design dei punti dello spazio di configurazione. La scelta di uno di questi rispetto ad un altro comporterà differenze nel tempo di simulazione e nella quantità di informazioni ricavate dalla fase di Design Space Exploration. I quattro design utilizzabili all'interno della metodologia sono:

**Random** In questo caso, le configurazioni dello spazio di design sono raccolte in modo casuale seguendo una funzione di densità di probabilità; nel lavoro di tesi precedente è stata utilizzata una distribuzione uniforme.

**Full factorial** Un esperimento di tipo Full Factorial è un esperimento costituito da due o più parametri, ognuno con diversi valori discreti chiamati anche livelli, in cui i diversi esperimenti assumono tutte le possibili combinazioni dei livelli per tali parametri. Tale esperimento consente la valutazione degli effetti di ciascun parametro e degli effetti causati dalle interazioni tra i parametri sulle relative variabili d'uscita.

**Central Composite Design** Questo design è utile nel caso vi siano più di due livelli possibili per i diversi parametri. Il set di punti rappresentanti gli esperimenti da effettuare è formato dall'insieme di tre diversi gruppi di punti generati da un design Full Factorial a due livelli, o uno frazionario, insieme ad altri centrali e ai bordi.

**Box-Behnken** Il design Box-Behnken è una DoE le cui combinazioni di parametri scelte sono al centro dei bordi dello spazio esaminato e alle quali viene aggiunto un altro punto con tutti i parametri al centro.

Il vantaggio principale di questa tecnica è dato dal fatto che viene evitata la scelta di combinazioni di parametri che sono entrambi a valori estremi, in contrasto con il Central Composite Design. Questa tecnica potrebbe essere adatta ad evitare la possibile generazione di punti singolari che potrebbero deteriorare il modello creato.

Dopo che una Design of Experiment è stata selezionata, la DSE esegue tutte le simulazioni necessarie per determinare le prestazioni del punto di lavoro. Considerando che la simulazione di un singolo punto può impiegarci ore, o giorni con dei sistemi realistici, minimizzare il numero di simulazioni da effettuare per delineare la curva di Pareto è molto vantaggioso. Prima di poter descrivere la metodologia utilizzata occorre citare le formalizzazioni necessarie contenute in [16].

Indicando con $n$ il numero di variabili decisionali che compongono un punto di lavoro, possiamo definire il vettore delle configurazioni:

$$x = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} \in X$$

dove $X$ è un dominio discreto finito, tipicamente un sottospazio di $N_0^n$. Il problema di ottimizzazione può essere definito da un set di $m$ funzioni da minimizzare:

$$\min_{x \in X} \quad f(x) = \begin{bmatrix} f_1(x) \\ \vdots \\ f_m(x) \end{bmatrix} \in R^m$$

soggetta a un set di $k$ di vincoli, che senza perdita di generalità, possono essere espressi come

$$e(x) = \begin{bmatrix} e_1(x) \\ \vdots \\ e_k(x) \end{bmatrix} \leq \begin{bmatrix} 0 \\ \vdots \\ 0 \end{bmatrix}$$

Utilizzando questa formulazione si nota che il codominio di $f(x)$ è correlato alle prestazioni della configurazione $x$. Per questo motivo tale funzione non può essere espressa analiticamente, ma deve essere calcolato ogni suo punto. L'idea base di questa metodologia è quella di cercare di costruire l'espressione analitica di $r(x)$ definita come:

$$f(x) = r(x) + \epsilon$$

dove $\epsilon$ rappresenta l'errore di approssimazione commesso. In questo modo utilizzando tale funzione è possibile calcolare analiticamente i punti del set di Pareto, senza effettuare simulazioni. Tuttavia tale affermazione ha senso se

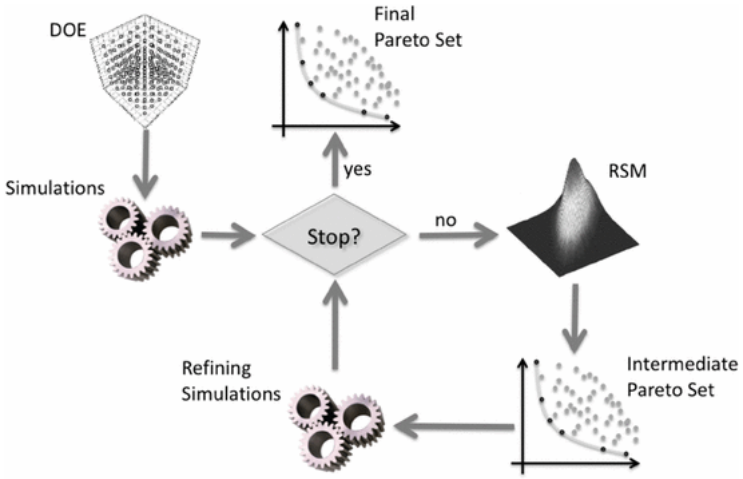



Figura 3.3: Design Space Exploration in ReSPIR

$\epsilon$ possiede delle proprietà statistiche ben precise, come media nulla e bassa varianza. Per esprimere $r(x)$ analiticamente possono essere utilizzati dei modelli derivati da:

1. Regressione Lineare
2. Shepard-Based Interpolation
3. Artificial neural network
4. Radial basis functions

In questo modo utilizzando un numero di archivi di tuple $< x, f(x) >$ aggiornate ogni volta che una simulazione di una nuova configurazione $x$ è stata eseguita con successo, è possibile migliorare l'accuratezza di $r(x)$ nell'approssimare la vera $f(x)$. Il flusso di lavoro di questa metodologia è raffigurato in Figura 3.3.

All'inizio si esegue la DoE per ottenere le configurazioni iniziali, si effettuano le simulazioni per ottenerne le prestazioni che vengono usate per ottenere una prima approssimazione di $r(x)$. Attraverso tale funzione si calcola l'Intermediate Pareto Set, si effettuano delle simulazioni su tali punti per verificare se è presente un miglioramento della copertura rispetto al set precedente. Si ripete questa procedura se non è stato stato raggiunto il limite di iterazioni e se il nuovo Intermediate Pareto Set presenta un miglioramento rispetto a quello precedente.

### 3.1.4 Multicube

Questa non è solo una metodologia per effettuare la Design Space Exploration, ma come descritto in [12, 51], è un framework che permette un'-

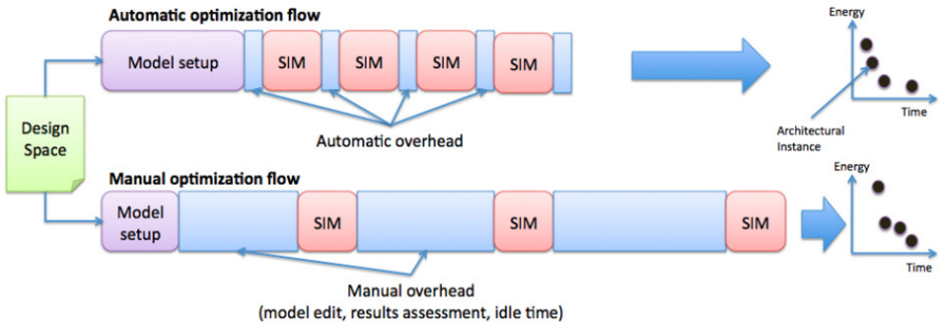



Figura 3.4: Design Space Exploration manuale ed automatica

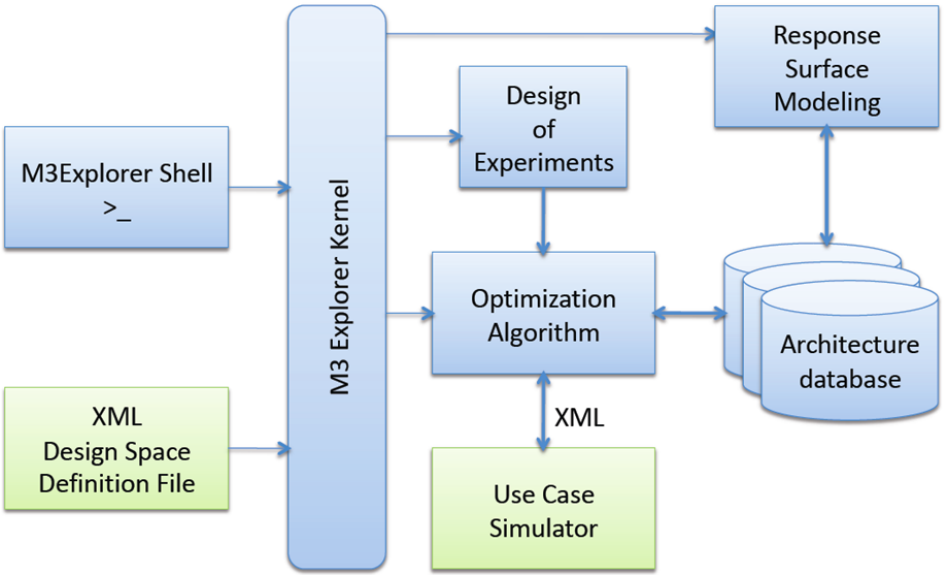



Figura 3.5: Struttura del framework Multicube explorer

efficiente Design Space Exploration, ed è lo strumento usato nel precedente lavoro di tesi per ottenere i punti di lavoro. Esso consiste in un'infrastruttura per l'esplorazione automatica dello spazio di progetto così da ottenere una velocizzazione ed ottimizzazione della fase di design. I vantaggi di un'esplorazione automatica rispetto ad una manuale sono rappresentati in Figura 3.4 dove è possibile osservare come automatizzare questa procedura permetta di diminuire al minimo i tempi morti tra una simulazione e l'altra, garantendo un'accurata analisi delle applicazioni in tempi minori. Multicube è un framework avanzato per l'ottimizzazione multi-obiettivo, interamente comandato da linea di comando o da script. Mediante la configurazione degli opportuni file xml è possibile modellizzare una qualsiasi piattaforma e definirne il suo simulatore. La Figura 3.5 rappresenta una panoramica della struttura di questo framework. Come già descritto precedentemente, Multicube integra

un'interfaccia a linea di comando (M3Explorer Shell nell'immagine) che permette la costruzione automatica delle strategie d'esplorazione. Il kernel dell'infrastruttura si occupa invece di leggere i file di configurazione ed esporre i parametri del Design Space ai vari moduli che implementano il processo d'ottimizzazione, quali i blocchi Design of Experiment, Optimization Algorithm e Use Case Simulator. La Design Space Exploration è eseguita utilizzando il livello d'astrazione esportato dal modello dell'architettura e seguendo la tecnica descritta nella sezione precedente. Queste informazioni verranno usate dal modulo di ottimizzazione per eseguire l'esplorazione. A questo punto le metriche del sistema verranno visualizzate nella shell di Multicube e salvate su disco.

La curva di Pareto delineata utilizzando tale algoritmo può essere composta da parecchi punti, occorre quindi ridurre tale numero per rendere l'operazione di management a Run-Time più efficiente. Tipicamente tale operazione viene effettuata utilizzando una tecnica di k-means clusterization [33]. Il numero di cluster, quindi di punti di lavoro, è una scelta fatta dallo sviluppatore. Dovrebbe essere un buon compromesso tra un numero elevato, che consente di avere una granularità fine nell'assegnazione delle risorse, ed un numero basso.

A questo livello un punto della fase di design potrebbero contenere parametri di differente natura, come definiti dal capitolo precedente. Quelli utilizzati dal framework ARGO descritto in questo elaborato di tesi sono solamente i parametri applicativi.

## 3.2 Tecniche utilizzate a Run-Time

Lo scopo di questo capitolo è quello di illustrare lo stato dell'arte nel campo del Run-Time management, resource management e su varie tecniche di monitoring. Nonostante il problema della gestione autonoma a Run-Time sia noto da tempo, le diverse tecniche proposte sono per la maggior parte lavori molto specializzati che si focalizzano solo su problemi o sistemi particolari. Queste soluzioni spesso forniscono buoni risultati ma sono applicabili soltanto in alcuni scenari ben definiti. Soltanto pochi sistemi sono capaci di provvedere ad una soluzione più generale alla crescente domanda di self-awareness. Queste sistemi spesso operano su più livelli di astrazione, tuttavia è possibile classificarli in base al componente di più basso livello che utilizzano:

**Livello hardware** Il sistema è principalmente composto da componenti hardware ad-hoc che svolgono funzioni di monitoraggio e di decisione.

**Livello di sistema operativo** Il sistema operativo stesso si occupa di monitorare le applicazioni e di prendere delle decisioni in base alle informazioni contenute in quel livello.

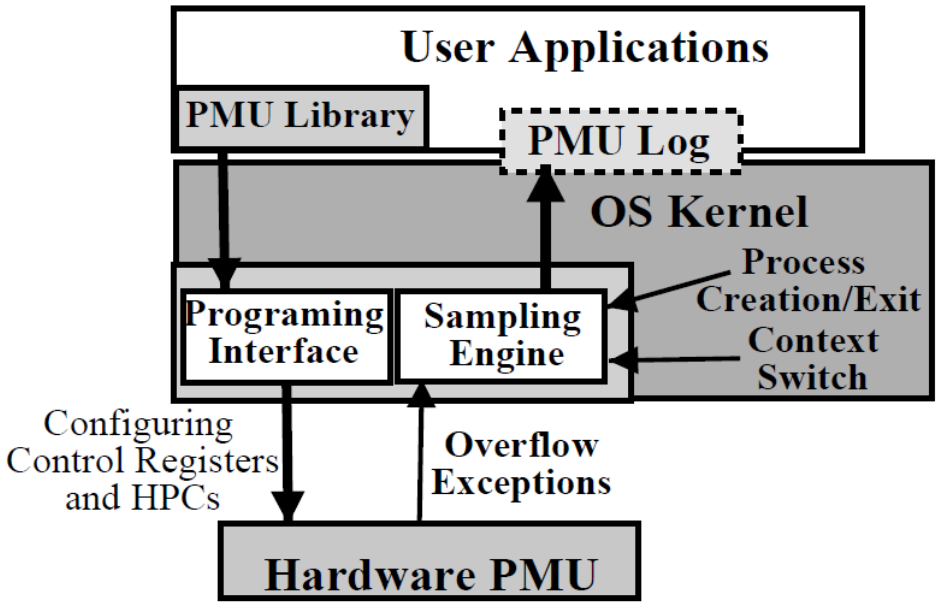



Figura 3.6: Schema architetturale del lavoro di Azimi et al.

**Livello applicativo** le funzionalità di monitoring e di decisione sono effettuate completamente in user-space.

Nelle prossime sezioni vedremo di illustrare alcuni esempi di sistemi di monitoring e/o gestione delle risorse e delle applicazioni Run-Time per i diversi livelli introdotti precedentemente.

### 3.2.1 Livello hardware

I sistemi di monitoring e di resource Run-Time management situati a livello hardware sono sistemi che fanno principalmente uso di caratteristiche strettamente di basso livello. Queste caratteristiche si trovano nell'hardware già presente o in componenti specifici atti a svolgere le nuove funzionalità. Questa sezione presenterà alcuni interessanti lavori di ricerca per illustrare lo stato dell'arte a livello hardware.

Uno degli esempi più classici riguardo al monitoring delle applicazioni è l'uso degli Hardware Performance Counter (HPC) forniti dal processore. Essi fanno generalmente parte di alcune unità del processore chiamate Performance Monitoring Units (PMU). In [42] ad esempio, Azimi et al. ottengono delle metriche dell'applicazione (cache miss, errori di predizione e altri) mediante il campionamento statistico di alcuni di questi contatori presenti nel sistema. Questo campionamento dei valori dei performance counter viene effettuato al costo del 2% di overhead (per una CPU da 2 GHz).

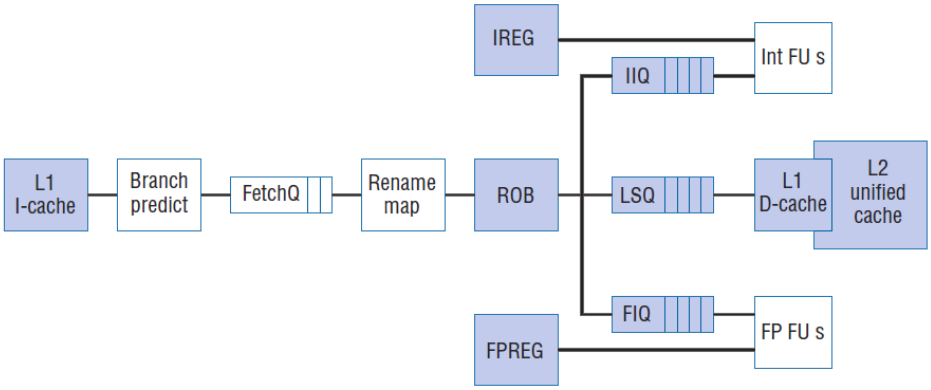



Figura 3.7: Sistema adattivo proposta da Albonesi et al.

La Figura 3.6 mostra uno schema a blocchi dell'architettura in questione. Come è possibile notare, le applicazioni si interfacciano con il sistema tramite una libreria a livello utente. A sua volta a livello di sistema operativo viene effettuato il campionamento e multiplexing dei contatori richiesti per restituire il risultato a Run-Time direttamente all'applicazione o tramite un file di log. Nonostante i valori forniti possano essere utilizzati per poter profilare il codice di un'applicazione in modo da valutarne i suoi colli di bottiglia, bisogna tenere conto che sono altamente dipendenti dall'architettura e pertanto potrebbero fornire risultati diversi da una macchina all'altra. Inoltre è stato dimostrato da Alameldeen e Wood [4] che l'utilizzo di metriche altamente dipendenti dall'architettura e dallo scheduling fornito dal sistema operativo, come il numero di istruzioni per ciclo (IPC), possono portare a considerazioni completamente sbagliate nel caso di programmi multiprocesso e multithread dove la comunicazione tra i processi e le varie primitive di locking rendono frequenti i casi in cui in base allo scheduling scelto, un thread possa eseguire più o meno istruzioni nello stesso intervallo di tempo.

Nel lavoro di Albonesi et al. [13], gli autori propongono di far attivare o disattivare l'hardware in base al tipo e alla quantità di carico attuale. Per raggiungere questo scopo non vengono usate delle FPGA (Field Progammable Gate Array), nel quale un cambiamento dello stato dell'hardware deve essere giustificato da elevati guadagni in consumo di potenza e/o prestazioni, ma dei componenti hardware formati a loro volta da sotto moduli. Essi possono essere attivati e disattivati in base ai dati forniti dal monitoring dei contatori hardware presenti nel sistema. La loro proposta (Figura 3.7) consiste in un processore superscalare con diversi componenti adattativi: le code di issue, load e store, il reorder buffer, il register file e le cache. In questo caso l'obiettivo principale è il minore consumo di potenza e allo stesso tempo la limitazione delle perdite di prestazioni dovute al minore hardware disponibile nel caso in cui dei moduli siano stati disattivati. I dati forniti dall'esecuzione di 14 benchmark dichiarano un risparmio di potenza massimo

di circa il 60% con una perdita di prestazioni massima di circa il 4%.

Nel lavoro di Padmanabhan et al. [38] viene proposto di scalare dinamicamente il voltaggio e la frequenza dei processori (DVS). Tale scelta deriva da due semplici considerazioni: quasi tutti i processori sono basati sulla tecnologia CMOS e i picchi di prestazioni richiedono un throughput più alto rispetto a quello normalmente richiesto. Nella tecnologia CMOS è relativamente semplice variare la frequenza di lavoro, che permette l'utilizzo di una minore tensione; considerando che la potenza dissipata è proporzionale al quadrato del voltaggio ($E \propto V^2$[48]), tale tecnica permette un considerevole risparmio energetico. L'innovazione rispetto alle normali tecniche di DVS è l'utilizzo delle stesse in un contesto di sistemi embedded soggetti a requisiti di real-time, dove è necessaria la garanzia sul tempo di elaborazione. Per ottenere tali risultati è necessaria un'integrazione con il task scheduler del sistema operativo. Nel loro lavoro viene proposta l'integrazione con i due maggiori scheduler per sistemi operativi real-time: Rate Monotonic e Earliest-Deadline-First, utilizzando tre tipi di algoritmo per ottenere un risparmio energetico sempre più considerevole. Il primo considera un adattamento statico, in cui viene utilizzato sempre il caso peggiore di ogni applicazione per determinare la frequenza da utilizzare. Il secondo algoritmo considera i task di ogni applicazione per selezionare la frequenza esatta. Infine l'ultimo algoritmo utilizza delle tecniche di look-ahead per valutare la futura capacità elaborativa richiesta. Dai risultati ottenuti dall'uso di queste tecniche in un sistema reale, è stato registrato un risparmio energetico dal 20% al 40%.

Bitirgen et al. [40] a loro volta propongono un altro sistema hardware per l'allocazione di risorse. Il loro framework utilizza tecniche di machine learning per creare un modello predittivo delle performance del sistema a partire da quelle delle singole applicazioni. Le informazioni relative alle applicazioni sono prese a loro volta dai performance counter mentre la rete neurale usata per fare machine learning è implementata in hardware. Le metriche usate per ottimizzare le performance sono tutte basate direttamente o indirettamente sul numero di istruzioni per ciclo, con tutte le limitazioni del caso. La Figura 3.8 rappresenta l'interazione tra il resource manager globale e le reti neurali che si occupano di predire le performance dell'applicazione. Il funzionamento di questa architettura è basato sul ricalcolo delle risorse per ogni applicazione ad ogni intervallo di tempo prefissato. Una volta concluso questo intervallo, il global resource manager interroga le applicazioni fornendo informazioni riguardo la loro storia passata e le risorse disponibili. La risposta consiste in una previsione delle performance previste nel prossimo intervallo. A questo punto il global manager aggregherà queste informazioni per determinare una predizione delle performance del sistema. Ripetendo queste interrogazioni un determinato numero di volte , con numero di risorse disponibili diverse, il resource manager prende la decisione che comporta il più alto incremento delle prestazioni.

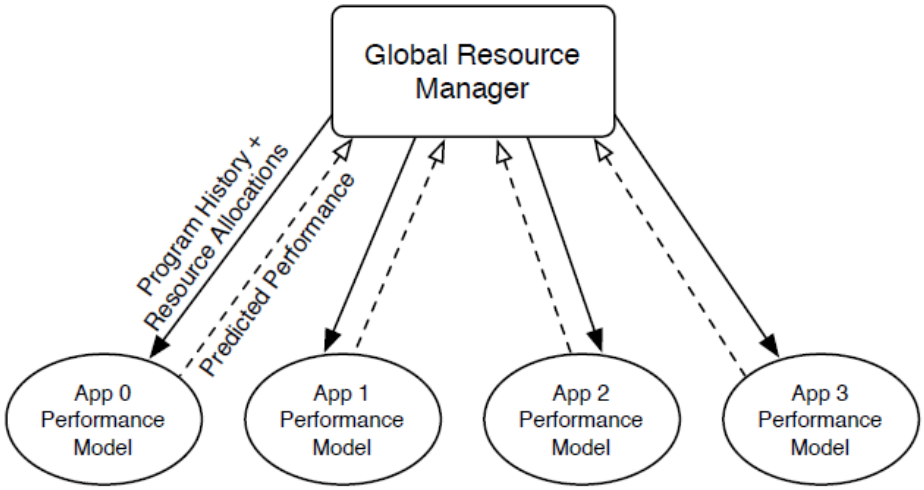



Figura 3.8: Interazione del framework proposto in Bitirgen et al.

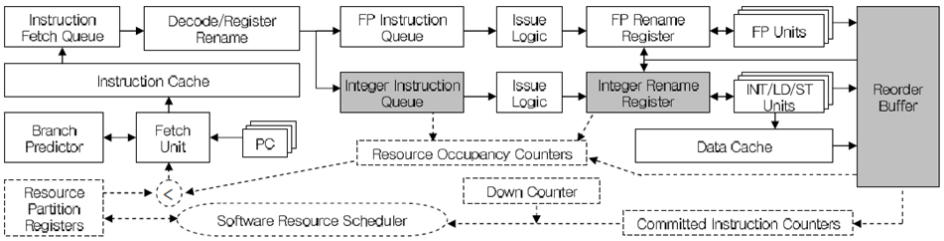



Figura 3.9: Diagramma a blocchi del modello di processore SMT

Anche nel lavoro di Choi e Yeung [45] viene proposta una soluzione simile. L'approccio è sempre basato sul machine learning ma in questo caso la soluzione descritta consiste in un processore ad-hoc basato su un modello di processore SMT (Simultaneous Multithreaded processor). In questo caso l'aggiunta di alcuni moduli al processore permette all'algoritmo di hill-climbing (un metodo di ottimizzazione iterativa) di raggiungere una distribuzione ottimale delle risorse arrivando fino al 19% d'incremento delle prestazioni. Anche in questo caso le metriche usate per la valutazione dei goal sono derivate dall'IPC. La Figura 3.9 rappresenta lo schema dell'architettura del processore SMT proposto dagli autori. I blocchi con bordo tratteggiato sono quelli aggiunti al processore per effettuare machine learning mentre quelli ombreggiati sono le risorse del processore suddivise dall'algoritmo.

Nell'ambito del monitoring hardware, El Shobaki in [46] descrive un sistema di monitoring specifico per sistemi operativi accelerati via hardware. In questo caso il monitoring è effettuato online su un sistema operativo real-time accelerato tramite un co-processore RTU (Real-Time Unit) che fornisce il sistema operativo e i vari servizi del kernel. Ogni applicativo a sua volta

deve includere un sistema operativo real-time minimale usato per interfacciarsi con il sistema operativo hardware. A questo punto, un'altra unità hardware dedicata, l'IPU (Integrated Probe Unit) si occupa di tenere traccia di vari eventi hardware e software che verranno mandati ad un computer host per un'analisi offline dei dati. Questa soluzione proposta è chiaramente complessa e si appresta ad un uso veramente specifico come quello dei sistemi operativi real-time accelerati via hardware.

Rimanendo sempre nell'ambito dei sistemi real-time, ambienti prestati per definizione alla gestione dei goal (in questo caso le deadline) delle applicazioni, Kohout et al. [37] propongono un blocco hardware, chiamato Real-Time Task Manager (RTM), che svolge i compiti di task scheduling, gestione di eventi e del tempo. Questi compiti vengono gestiti in tempo costante, diminuendo l'uso del processore da parte del sistema operativo del 90% e il tempo di risposta massimo fino all'81%.

Un ultimo esempio di sistema autonomo a livello hardware è quello proposto da Fiorin et al. [31]. Nel loro lavoro viene proposta un'architettura di monitoring per Networks on Chip (NoCs) che fornisce delle informazioni utili ai designer di MultiProcessor System onChip (MPSoC) per poter sfruttare a pieno ed efficacemente le risorse disponibili. Una Network on Chip (NoC) è una microrete tra i componenti di un System on Chip (SoC) ed è un'astrazione della comunicazione tra i componenti stessi. Le NoC vengono usate con l'obiettivo di soddisfare una certa qualità del servizio in termine di affidabilità, prestazioni e consumo di potenza [29]. Per garantire queste caratteristiche, soprattutto in sistemi complessi come quelli multiprocessore, una delle funzionalità più importanti è quella di monitoring. In [31], gli autori si occupano di implementare queste funzionalità direttamente in hardware. Facendo riferimento alla Figura 3.10, è possibile notare la struttura eterogenea del sistema, nel quale alle interfacce di rete (NI nell'immagine) sono state aggiunte delle sonde (P nell'immagine) per monitorare il throughput dei dati trasmessi e ricevuti dai core, l'utilizzo delle risorse, quali i buffer, e la latenza di comunicazione. Inoltre, una PMU (Probe Management Unit) è presente per gestire le diverse sonde, ottenerne i loro valori misurati ed effettuare le elaborazioni necessarie. Questo sistema permette di monitorare la NoC al costo di un'area aggiuntiva 0.58 $mm^2$ e dello 0.2% in più di traffico nella NoC.

### 3.2.2 Livello di Sistema Operativo

Com'è possibile intuire dalla descrizione fatta nella sezione precedente, non è sempre possibili utilizzare soluzioni di tipo hardware. Lo sviluppo di hardware dedicato comporta un allungamento del time-to-market di un prodotto e aumenta la probabilità che la soluzione proposta diventi obsoleta o troppo specifica e non usabile in diversi ambiti. Il riutilizzo delle architetture già progettate è altamente importante in ambiti in cui il prodotto che prima ar-

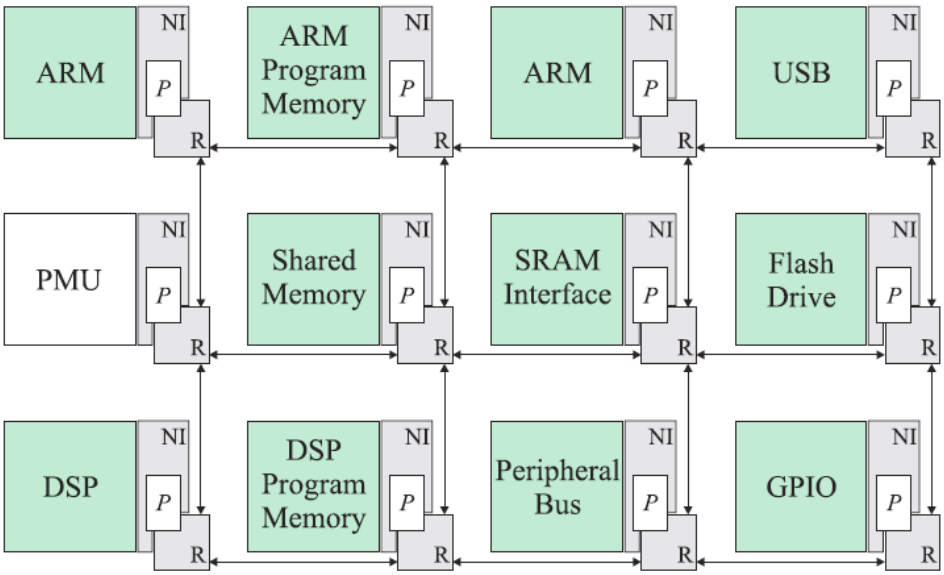



Figura 3.10: Panoramica dell'architettura di monitoring per NoC

riva sul mercato conquista la quota di utilizzatori maggiori. In questo caso è quindi ideale agire ad un livello d'astrazione più alto, che sia a livello del sistema operativo o quello applicativo. Questa sezione presenterà dei progetti di ricerca focalizzati a livello di sistema operativo (SO). La scelta di operare a questo livello è spesso dettata dalla eccessiva complessità di sviluppare un sistema hardware dedicato con la contemporanea necessità di avere accesso ad informazioni difficili o impossibili da recuperare in user-space.

Un approccio possibile a questo tipo di sistemi è quello presentato da Almeida et al. in [15]. In questo caso gli autori, basandosi sulla teoria del controllo, hanno implementato un controllore PID (Proporzionale- Integrale-Derivativo) applicandolo ad MPSoC composto da diverse Network Processing Unit (NPU). Lo scopo di questo sistema è quello di fornire un controllo ad anello chiuso che possa mantenere il throughput richiesto dalle applicazioni riducendo la sovrallocazione delle risorse. Il controllore proposto è stato implementato come servizio del sistema operativo. La Figura 3.11 rappresenta una panoramica del sistema proposto. Com'è possibile notare dall'immagine, ogni controllore PID è associato ad un processo e riceve come input i dati forniti dal monitor di throughput. Il controllore PID prenderà quindi una decisione cambiando la frequenza del processore in base ai parametri di configurazione $k_p$, $k_i$ e $k_d$ e all'errore dato dalla differenza tra il throughput richiesto e quello effettivo.

Un'altra soluzione a livello di sistema operativo è Odissey [7], un progetto di ricerca che estende il sistema operativo NetBSD [3] per offrire una collaborazione maggiore con le applicazioni. Le decisioni riguardo l'allocazione

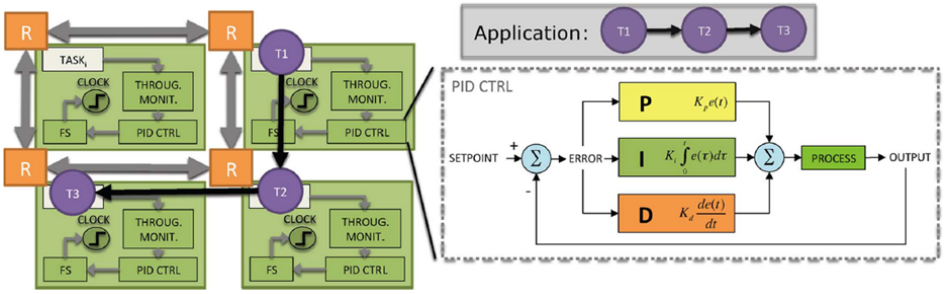



Figura 3.11: Vista architetturale del sistema proposto con controllori PID

di risorse sono prese interamente dal sistema operativo e tengono conto anche delle informazioni riguardanti i diversi livelli di qualità del servizio che un'applicazione può raggiungere. Una volta che queste decisioni sono state prese, il sistema operativo notificherà le applicazioni che modificheranno il loro comportamento in base alle risorse allocate dal sistema.

Un esempio di sistema autonomo implementato a livello di sistema operativo è AQuoSA [30, 50]. Il framework proposto dagli autori consiste in un insieme di modifiche non invasive in cima al kernel Linux. Queste modifiche hanno l'obiettivo di fornire uno scheduling ottimale per applicazioni real-time e per tutte quelle dove i risultati sono dipendenti dal tempo d'esecuzione, come ad esempio applicazioni multimediali. Il supporto a vecchie applicazioni che non fanno uso del framework è garantito tramite l'utilizzo dello scheduler standard di Linux. L'architettura di controllo è basata su un modello ad eventi discreti con controllo decentralizzato. Le applicazioni richiedono l'uso di una certa percentuale di banda del processore tramite un controllore, uno per ogni applicazione, che a sua volta comunica con un supervisore a livello di sistema che garantisce l'allocazione della banda corretta alle varie applicazioni. L'architettura di AQuoSA è visibile in Figura 3.12 ed è composta da diverse componenti tra le quali:

- Una patch, chiamata GSP (Generic Scheduler Patch), al kernel che permette di intercettare eventi, tra i quali la creazione o la distruzione di un processo, gli eventi di lock e unlock su risorse condivise.

- Il Kernel Abstraction Layer (KAL), un insieme di macro scritte in C che permettono di astrarre alcune funzionalità del kernel necessarie all'interno del framework.

- Il modulo QoS Reservation che definisce, tramite un modulo del kernel e una libreria a livello userspace, uno scheduler di tipo EDF (Earliest Deadline First) e permette di utilizzarne le sue funzionalità.

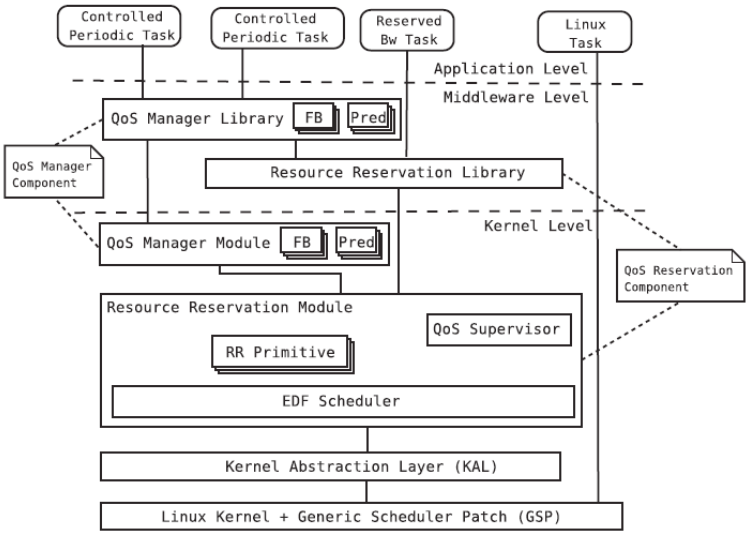



Figura 3.12: Architettura di AQuoSA

- Il modulo QoS manager, anch'esso implementato in parte nel kernel ed in parte in userspace. Si occupa di fornire le varie funzionalità di controllo basate sul feedback e su alcuni predittori definiti dal sistema.

Le varie applicazioni dovranno usare le API fornite dal framework per definire le richieste d'uso del processore, i diversi parametri riguardanti le strategie di controllo e i predittori usati. L'applicazione quindi dovrà attivarsi periodicamente e chiamare le varie funzionalità del sistema per controllare le sue perfomance e fornire questi risultati al predittore e al controllore dell'applicazione. I test degli autori mostrano che questa tecnica fornisce uno scheduling corretto con soltanto il 3% di overhead sul tempo d'esecuzione.

### 3.2.3 Livello Applicativo

Questo è il livello in cui si inserisce il framework trattato in questo elaborato di testi. La caratteristica fondamentale di questo livello è che il controllo esercitato avviene interfacciandosi al sistema operativo. Di conseguenza gli unici parametri che si possono variare sono quelli applicativi e le risorse di sistema che vengono esposte, il vantaggio è che tale approccio è virtualmente indipendente dall'hardware e software utilizzato dal sistema. Tali tecniche sono state concepite per essere utilizzate nel contesto dei sistemi embedded, ma possono essere utilizzate anche in altri scenari dove il self-awareness è una caratteristica importante, per esempio nei server per far fronte ai picchi di richiesta. Questa versatilità non è possibile nei livelli precedenti, dove l'hardware e software sono più opachi rispetto alla tecnica utilizzata.

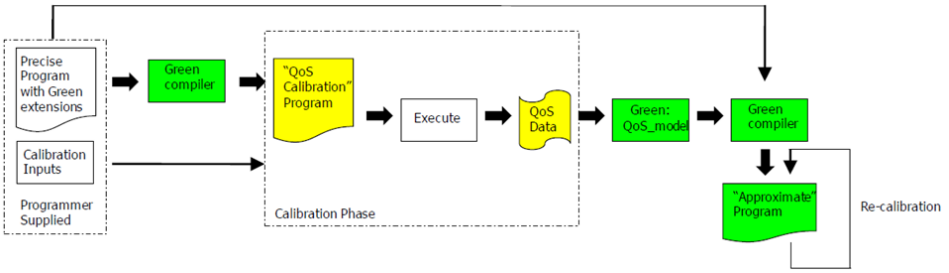



Figura 3.13: Schema del framework Green

Il lavoro di Baek e Chilimbi consiste in un framework, chiamato Green, che tramite l'estensione del linguaggio C/C++ permette di implementare una modalità di approssimazione per le funzioni e i cicli [53]. Lo scopo di questo framework è quello di fornire una modalità per dichiarare semplicemente la qualità del servizio (QoS) al variare di diverse versioni approssimate delle stesse funzioni o di cicli eseguiti con meno iterazioni. Come è possibile notare nella Figura 3.13, il flusso di sviluppo all'interno di Green si svolge in più fasi. Prima di tutto lo sviluppatore deve fornire sia le diverse approssimazioni delle funzioni che le informazioni necessarie per determinare il guadagno o la perdita di qualità in base all'approssimazione scelta. La fase successiva consiste nella calibrazione dell'applicazione con lo scopo di creare un modello della QoS basato sul variare delle approssimazioni. Mentre le funzioni devono essere fornite in diverse versioni, tante quante le approssimazioni volute, i cicli vengono invece approssimati riducendo il numero di iterazioni mediante una conclusione anticipata del ciclo stesso. La stessa tipologia di approssimazione è eseguita nel lavoro di Misailovic et al. [44] dove vengono saltate regolarmente alcune iterazioni dai cicli ma non viene effettuata nessuna approssimazione delle funzioni. Per concludere, una volta che il modello è stato creato, l'applicazione viene ricompilata assieme a queste informazioni; in questo modo l'applicazione varierà la qualità dei suoi risultati in base ai vincoli precedentemente forniti e allo stato della QoS monitorata ad intervalli regolari.

Il middleware ControlWare è un'architettura basata sulla teoria del controllo specializzata su servizi internet. La Figura 3.14 delinea la metodologia di sviluppo per applicazioni che sfruttano ControlWare. La prima fase consiste nella specifica dei diversi livelli di qualità del servizio che l'applicazione può fornire. Successivamente il blocco chiamato nell'immagine QoS mapper si occupa di leggere ed interpretare le precedenti informazioni trasformandole in un insieme di valori obbiettivo e di cicli di controllo a feedback. Questa fase è svolta tramite l'utilizzo di una libreria di modelli, ognuno che descrive una diversa caratteristica di QoS e definisce un sistema di controllo ad anello chiuso (come in [5]) atto a risolvere tale problema. La fase successiva consiste nella configurazione dei vari monitor e degli attuatori forniti da ControlWare

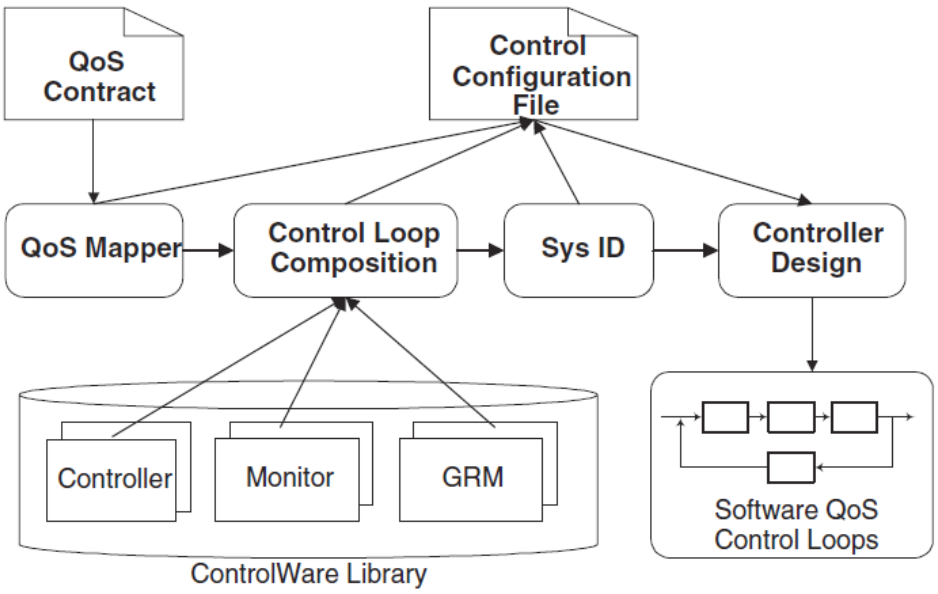



Figura 3.14: Metodologia di sviluppo di ControlWare

in modo da soddisfare la configurazione del QoS mapper. Dopo aver stimato matematicamente un modello per il sistema tramite le sue prestazioni generali, il controllore dell'applicazione viene creato ed ottimizzato per garantire la stabilità e la risposta ai disturbi dovuti alle variazioni del carico. Tutte queste informazioni verranno poi usate dal GRM (Generic Resource Manager) per decidere come allocare le risorse alle varie applicazioni. Il carico di lavoro di un'applicazione verrà inserito in opportune code, una per ogni classe di servizio, che verranno monitorate dal GRM il quale prenderà le dovute decisioni in base alla quantità di task presenti nella coda e il numero di risorse assegnate per quel processo.

Un'altra strategia di gestione delle applicazioni a Run-Time è basata sulle proposte descritte in [10, 9, 8]. Il lavoro descritto dagli autori utilizza una fase di Design Space Exploration dettagliata ed automatizzata, atta a fornire le informazioni necessarie ad un Run-Time manager che opera a basso overhead e che alloca le diverse risorse alle applicazioni. L'input del sistema è un insieme di diverse versioni della stessa applicazione sviluppata con livelli di parallelizzazione differenti. Queste versioni della stessa applicazione vengono usate per analizzare il comportamento dell'applicazione alle diverse frequenze dei processori. L'idea base è quella di utilizzare delle euristiche costruite a design time, per riuscire a risolvere in tempi accettabili il problema di ottimizzazione, riconducibile al problema dello zaino, come descritto precedentemente in fase di Design-Time. La fase d'esplorazione viene effettuata interfacciandosi con dei simulatori della piattaforma a due diversi livelli. Il primo simulatore è usato per fornire una prima classifi-

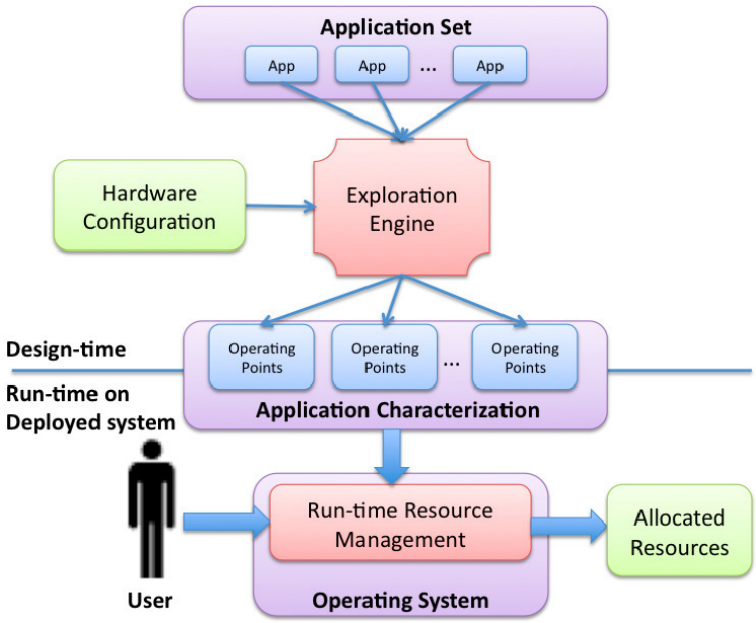



Figura 3.15: Flusso di progettazione nel framework ARTE

cazione dell'applicazione ad alto livello e per fornire le caratteristiche di un numero elevato di configurazioni. Il secondo, invece, effettua una simulazione molto più accurata, quindi più lenta, per le sole configurazioni precedenti che sono Pareto-ottimali rispetto ai vari parametri. A questo punto tramite le informazioni ricavate a Design-Time, il Run-Time resource manager deciderà quale configurazione di frequenze e parallelizzazione usare per ogni applicazione tramite uno scheduling guidato dalle deadline delle applicazioni stesse. La configurazione scelta sarà quella col minor consumo di potenza e la capacità rispettare le deadline.

Il framework ARTE, sviluppato da Mariani et al. [17], consiste invece in un sistema per la gestione a Run-Time delle applicazioni in base alla conoscenza fornita dai dati acquisiti a design-time. Il lavoro utilizza la teoria delle code e ha lo scopo di migliorare le prestazioni delle applicazioni dal punto di vista del tempo di risposta medio. La Figura 3.15 rappresenta il flusso di progettazione di un'applicazione all'interno del framework. In ARTE, lo spazio di progettazione è formato da diverse versioni della stessa applicazioni ma con parallelizzazione diversa se il programma lo prevede. Il motore d'esplorazione si occupa di generare una lista di punti operativi che caratterizzano l'applicazione in base al tipo di parallelizzazione scelta. In base a questi punti e allo stato del sistema il Resource Manager presente nel sistema operativo deciderà quante risorse e quanti thread allocare per ogni applicazione.

Il framework PowerDial descritto in [23] utilizza il PowerDial Control

System per formare un anello chiuso con l'applicazione. L'idea generale è utilizzare un meccanismo di feedback che controlla la velocità di esecuzione dell'applicazione in fase di Run-Time e, utilizzando i dynamic knobs, il controllore decide la configurazione che permette all'esecuzione di rispettare i goal stabiliti. PowerDial utilizza il framework Heartbeats come sistema di feedback, che permette di fare il profiling del ciclo che impiega più tempo in modo da ottenere la sua frequenza di esecuzione a Run-Time. Utilizzando la differenza fra le prestazioni desiderate e quelle misurate il controllore calcola lo speed-up da attuare variando il valore dei dynamic knobs. Considerando che tale valore appartiene ai numeri reali, può accadere che la granularità di selezione degli speed-up possibile non possa garantire tale valore. Per questa ragione viene introdotto il concetto di time quantum, definita come il tempo impiegato ad effettuare venti heartbeats, con l'obbiettivo di definire uno spazio temporale minimo in cui regolare il valore di speed-up necessario. Per esempio se il controllore impone uno speed-up di 1.5, ma i dynamic knobs possono portare l'applicazione ad uno speed-up di 1 o 2, la scelta realmente attuata è di utilizzare uno speed-up di 1 per dieci heartbeats e uno speed-up di 2 per il restante tempo, in modo da portare lo speed-up medio rispetto al time quantum di 1.5.

Come ultimo esempio di adattabilità a livello applicativo presentiamo il framework SEEC. Tra i vari sistemi autonomi a livello applicativo è uno tra quelli con le potenzialità maggiori. SEEC è un framework sviluppato in collaborazione tra il Computer Science and Artificial Intelligence Laboratory del Massachusetts Institute of Technology e dal Dipartimento di Elettronica e Informazione del Politecnico di Milano. Consiste in un sistema autoadattativo capace di gestire a Run-Time applicazioni che espongono diversi goal al sistema [35, 22]. Uno dei perni principali di questo framework è il ciclo ODA Observe-Decide-Act, tipico dei sistemi autoadattivi, che consiste nelle tre tipiche fasi della teoria del controllo ad anello chiuso. Queste tre fasi coincidono in parte con quelle del ciclo MAPE [25], dove la fase di decisione è formata dai blocchi Analyze e Plan. In Figura 3.16 è possibile vedere il ciclo di controllo di un'applicazione self-aware composto da tre diverse fasi:

**Osserva** in questa fase viene monitorata l'esecuzione dell'applicazione. Vengono impiegati dei sensori, chiamati anche monitor, che osservano lo stato interno fornendo un'indicazione utile al blocco di decisione.

**Decidi** nella fase decisionale vengono elaborati i dati ricavati dalla fase di osservazione e vengono valutate le modifiche dei parametri di controllo per la successiva esecuzione del programma.

**Agisci** questa fase si occupa della modifica effettiva dei parametri, di sistema o dell'applicazione, decisi dal blocco precedente.

Riprendendo la classificazione fornita in [21], le caratteristiche interessanti di questo framework sono:

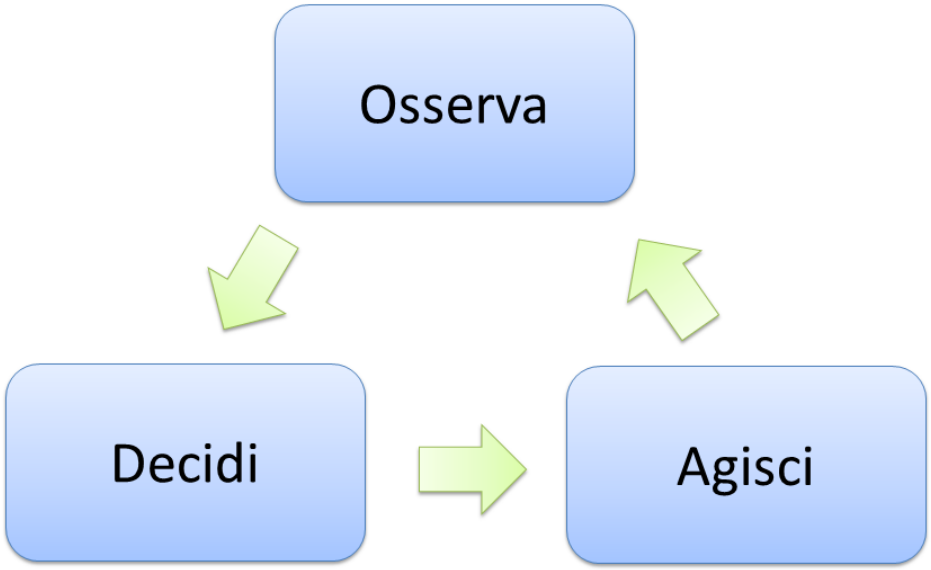



Figura 3.16: Ciclo Observe-Decide-Act

- L'inclusione di meccanismi di dichiarazione dei goal e del feedback necessario al controllo.
- L'utilizzo di un controllo adattativo abile nel rispondere velocemente a nuove applicazioni o a diverse fasi operative delle applicazioni stesse.
- L'implementazione di un motore decisionale capace di decidere se allocare proporzionalmente le risorse o minimizzare il tempo d'esecuzione di una applicazione fornendo più risorse e permettendo quindi all'applicazione di liberare il sistema il prima possibile.
- L'inclusione di tecniche basaste sul feedback e sul reinforcement learning per adattare il modello del sistema dinamicamente.

SEEC utilizza un'infrastruttura di monitoring sviluppata dal stesso gruppo: Application Heartbeat , descritto precedentemente. La Figura 3.17 fornisce uno schema concettuale della modalità di funzionamento del framework SEEC. É possibile vedere come il Run-Time manager implementato in SEEC fornisca diversi livelli possibili d'adattamento. Essi variano da uno schema di controllo ad anello chiuso fino ad un più raffinato sistema di machine learning. La scelta di utilizzare un livello rispetto ad un altro dipenderà dal massimo overhead richiesto e dalle caratteristiche del modello del sistema fornito al framework. Ad esempio, nel caso in cui questo modello sia accurato, l'utilizzo delle tecniche di machine learning comporterebbe soltanto un peggioramento del tempo d'esecuzione, dovuto alla fase di training, mentre l'uso di uno dei livelli sottostanti fornirebbe gli stessi risultati ma in

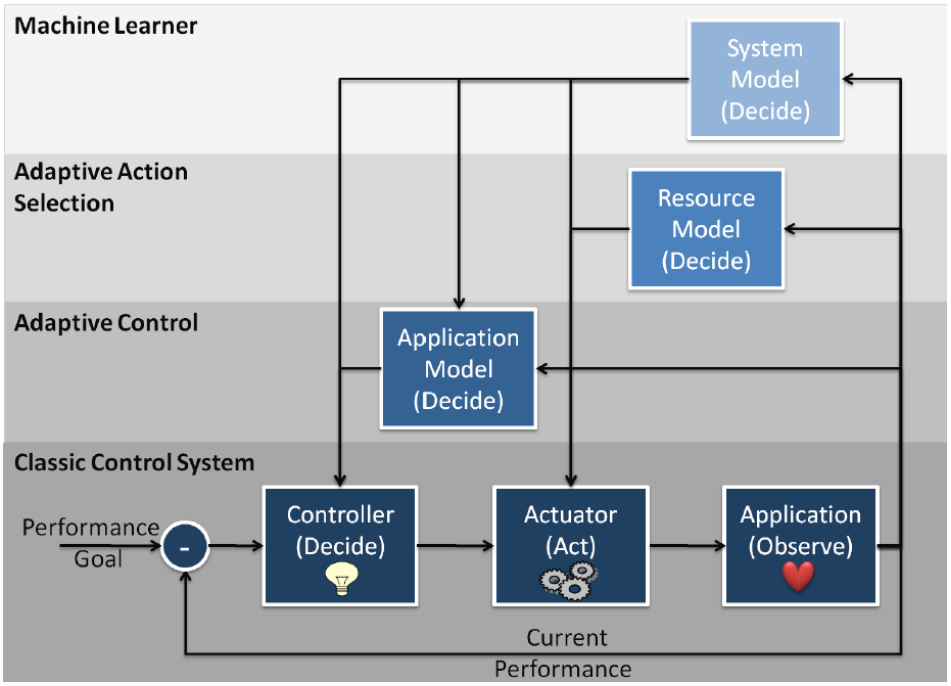



Figura 3.17: Schema concettuale del framework SEEC

minor tempo. Il controllore usato in SEEC utilizza i dati forniti dal blocco Observe, quindi da Heartbeat, assieme alle informazioni fornite dai diversi livelli di controllo per decidere come cambiare l'allocazione di risorse o la configurazione dei parametri dell'applicazione. Queste operazioni verranno effettuate dal blocco di attuatori (Actuator nell'immagine).

## 3.3 Contributo del framework ARGO

n questo capitolo sono stati illustrati alcuni lavori dello stato dell'arte nei diversi ambiti a cui fa riferimento questa tesi. In primo luogo si è parlato delle tecniche utilizzate a design time per delineare la curva di Pareto, punto di partenza per l'utilizzo del framework ARGO. Successivamente sono stati descritte le tecniche utilizzate a Run-Time per fornire all'applicazione (o al sistema) capacità di self-awareness. Da questa ricerca è emerso che le metodologie utilizzate sono altamente specializzate, oppure necessitano di specifici modelli per funzionare correttamente. Il framework trattato in questo lavoro di tesi cerca di fornire la più alta adattabilità ad una applicazione generica, permettendo di definire vincoli rispetto a qualsiasi genere di metrica e parametro, richiedendo solamente di esplicitare l'insieme di punti di lavoro Pareto-ottimali. Considerando che tale framework è totalmente in user-space risulta indipendente dall'architettura hardware e può

coesistere con altre tecniche di livello più basso, minimizzando l'overhead computazionale introdotto.

# Capitolo 4

# Metodologia proposta

In questo capitolo verrà descritta la metodologia utilizzata dal framework ARGO, precisando le scelte progettuali effettuate. In primo luogo verrà discusso il contesto in cui si inserisce. Successivamente verrà descritta l'architettura di riferimento della struttura del framework realizzato. Verrà inoltre descritta la metodologia proposta, analizzando il problema che si cerca di risolvere, e in seguito verranno elencate alcune osservazioni riguardo l'interazione fra i monitor e l'Application-Specific Run-Time Manager. Infine verrà descritto il procedimento di integrazione del framework nell'applicazione.

## 4.1 Contesto generale

Lo scenario principale che fa da contesto all'applicazione è il mondo dei sistemi embedded. Il framework ARGO è stato infatti pensato per essere incluso in una generica applicazione che ha uno scopo preciso. Questa assunzione implica la presenza di una parte di codice che continua a ripetere un'elaborazione su dei dati. Tipicamente un algoritmo che appartiene a questa classe di programmi è partizionato in tre fasi distinte: nella prima effettua tutte le operazioni richieste per l'inizializzazione dell'applicazione; nella seconda effettua l'elaborazione scandendo tutti i dati in ingresso, generalmente utilizzando un ciclo; nell'ultima fase annulla tutte le operazioni fatte nell'inizializzazione. Con il termine applicazione si farà riferimento a un programma strutturato come appena descritto.

Il concetto di punto di lavoro rientra nella definizione di Operating Point data nel Capitolo 2, ovvero è l'associazione della misura delle metriche misurate utilizzando una particolare scelta dei valori dei parametri. Siccome il framework non utilizza un Resource-Manager, gli unici tipi di parametri che è possibile variare sono quelli applicativi, per questa ragione da questo capitolo si usa indistintamente il termine punto di lavoro, configurazione oppure Operating Point (OP).

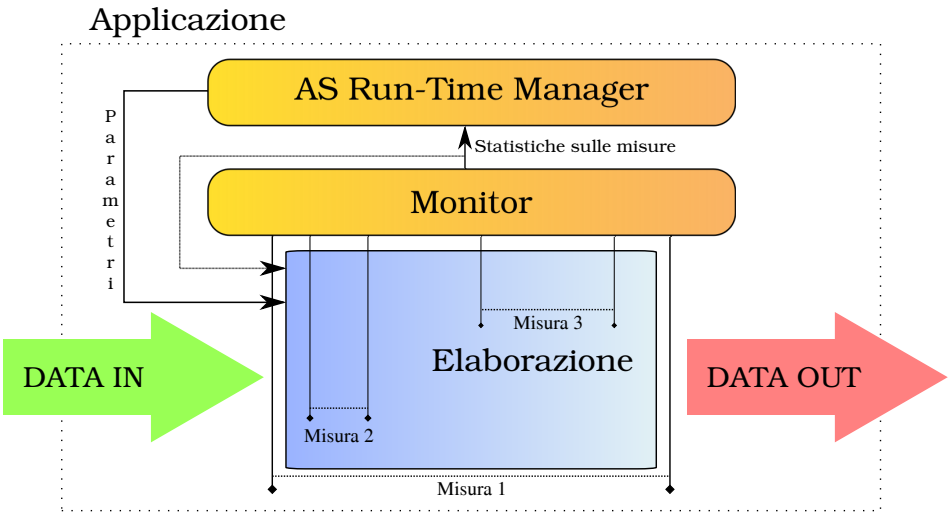



Figura 4.1: Architettura del framework ARGO

## 4.2 Architettura di riferimento

Il framework è composto da due componenti distinti: i Monitor e l'Application-Specific Run-Time Manager (AS-RTM). Questi due moduli forniscono all'applicazione diverse caratteristiche tipiche del self-awarness descritte nel Background: i Monitor hanno il compito di fornire all'applicazione la consapevolezza delle proprie prestazioni, misurando le metriche selezionate dallo sviluppatore; l'AS-RTM invece si occupa di garantire all'applicazione l'autonomia riguardo alla scelta della configurazione da adottare, la robustezza considerando differenze tra prestazioni aspettate e quelle misurate in tutte le scelte che effettua e, infine, la dinamicità esponendo all'applicazione delle funzioni per modificare le politiche decisionali di alto livello imposte dal progettista. La lista dei punti di lavoro fornisce all'applicazione la capacità di tracciare le modifiche, fungendo da modello predittivo delle prestazioni. L'insieme di questi elementi fornisce all'applicazione la capacità di adattamento.

In Figura 4.1 è rappresentata l'architettura di riferimento del framework. Come è possibile notare dal riquadro tratteggiato, il framework è totalmente contenuto nell'applicazione stessa, in questo modo ogni istanza del programma è in grado di riconfigurarsi in modo indipendente. I blocchi colorati in azzurro raffigurano il codice che si occupa di effettuare l'elaborazione, mentre in giallo sono raffigurati i blocchi che costituiscono ARGO. Se questi ultimi non fossero presenti è possibile notare come l'elaborazione sia in anello aperto: i dati arrivano in ingresso, viene effettuata l'elaborazione, mantenendo la configurazione con cui è stata inizializzata, producendo i dati desiderati in uscita. Il framework realizzato chiude tale anello, fornendo all'applicazione,

all'inizio di ogni elaborazione, la configurazione di parametri migliore a fronte dell'evoluzione della situazione e/o delle statistiche sulle misure raccolte dai monitor.

Nella figura sono rappresentati due anelli in retroazione: il primo comprende solamente la parte di monitoring, mentre l'anello esterno include l'AS Run-Time Manager. Il motivo di tale scelta è di fornire allo sviluppatore un approccio modulare all'uso del framework, in base alle sue necessità. Se per esempio il codice di elaborazione è molto piccolo, magari con un solo parametro, non occorre che ci sia un manager, è sufficiente utilizzare le informazioni raccolte dai monitor per adattare il proprio funzionamento. I monitor sono l'unico modo per avere informazioni riguardo le prestazioni in fase di Run-Time, permettendo di chiudere l'anello. L'AS-RTM invece sfrutta le informazioni contenute nella lista dei punti di lavoro, adattandole alla realtà percepita dai monitor, per selezionare il punto di lavoro migliore. Se non fossero presenti dei Monitor, l'AS-RTM si potrebbe paragonare a un database di Operating Point dove lo sviluppatore ne può selezionare uno in base ai vincoli e preferenze impostate.

Nel lavoro di tesi precedente si è lavorato molto sulla struttura di monitoring, mentre l'AS-RTM prodotto in alcune situazioni manifestava un comportamento troppo rigido e la sua integrazione nell'applicazione risultava opaca. Per queste ragioni in questo lavoro di tesi si è rivisto completamente l'algoritmo decisionale partendo dalla fase di design, mentre l'architettura di monitoring è stata solamente modificata per adattarsi al nuovo AS-RTM.

## 4.3 Metodologia

Lo scopo del framework sviluppato è quello di fornire ad una applicazione la capacità di lavorare nella migliore configurazione, adattandosi a cambiamenti nell'ambiente in cui viene eseguita e a variazioni della quantità di dati in ingresso, senza la necessità di coordinarsi con altri attori esterni. Considerando che un'applicazione è soggetta a dei vincoli, l'Operating Point migliore deve essere *valido*, ovvero rispettare tutte le restrizioni imposte. Fra questi punti lo sviluppatore deve definire un metodo per esprimere le sue preferenze. Per questo motivo è stata introdotto il concetto di rank: associare ogni Operating Point a un valore che ne determina la bontà. Per ottenere questo scopo viene fornita la possibilità di definire una funzione di ranking che deve essere massimizzata o minimizzata.

Nella definizione di migliore appena descritta, si è supposto che esista almeno un punto di lavoro che soddisfi tutti i vincoli, ma se tale assunzione non aderisse con la realtà, la definizione di migliore non sarebbe più valida. Nel lavoro precedente si adottava un approccio conservativo: siccome non c'è nessun punto di lavoro che permette di portare a termine il lavoro nelle modalità richieste, non esiste un'azione che può risolvere il problema, di

conseguenza non si effettuano modifiche. Se questo procedimento può essere corretto nel caso di applicazioni critiche, nel caso generale si desidera portare l'applicazione più vicina possibile al comportamento valido.

Un *vincolo* esprime un singolo limite, inferiore o superiore, che può assumere una grandezza, sia essa un parametro, una metrica o una misura rilevata dai monitor. L'insieme di tali restrizioni delineano la zona in cui l'applicazione deve operare, ma non tutti i vincoli hanno la stessa priorità. Infatti lo sviluppatore deve fornire un ordinamento stretto dei vincoli inseriti. Questa distinzione può sembrare bizzarra, perché se si vuole specificare un intervallo di valori che può assumere una grandezza occorre utilizzare due vincoli, e questo implica che un limite è più importante dell'altro. Tuttavia questa distinzione permette al framework di indicare in che zona ripiegare nel caso non fosse possibile agire nell'intervallo desiderato.

Si supponga per esempio che una metrica dei punti di lavoro sia la pressione di un serbatoio e che la zona in cui il sistema lavori sia compresa tra 3 e 5 bar. Se non fosse possibile lavorare in tale intervallo occorre scegliere fra pressioni inferiori a 3 bar, oppure superiori a 5 bar. Nel caso si scelga di operare a basse pressioni probabilmente il sistema non funzionerà correttamente, tuttavia è preferibile rispetto a lavorare con alte pressioni. In questo caso il vincolo che esprime il limite superiore ha priorità più alta rispetto al vincolo che esprime quello inferiore.

Nel caso non esistano punti di lavoro validi, il comportamento voluto dal framework è quello di avvicinare il più possibile l'elaborazione alla zona definita dai vincoli. Teoricamente per raggiungere tale scopo, sarebbe sufficiente scorrere tutti i punti di lavoro e per ognuno contare quanti vincoli contigui rispetta, partendo da quello di priorità massima, e selezionare il punto che ne soddisfa in numero maggiore. A parità di tale valore si utilizza il punteggio di rank. Il problema di questo metodo è che ha complessità lineare rispetto al numero di Operating Point in ogni situazione. Il framework utilizza un approccio più efficiente che permette di ottenere lo stesso risultato, ma controllando la metà dei punti nel caso generale, potenzialmente anche un solo punto di lavoro nel caso migliore. Nel caso peggiore si visiteranno tutti i punti di lavoro. L'algoritmo implementato verrà descritto nel capitolo successivo.

Possiamo quindi riassumere la definizione di migliore in base alla validità degli Operating Point, in particolare il punto di lavoro migliore è quello che ha punteggio di rank più alto se esiste almeno un Operating Point valido, altrimenti è quello che avvicina il sistema il più possibile alla zona di validità.

Durante la Design Space Exploration viene individuata la curva di Pareto che contiene l'insieme di tutti i punti di lavoro che forniscono all'applicazione un compromesso ottimale fra le prestazioni. La lista di tali punti costituisce il punto di partenza nell'integrazione del framework nell'applicazione. Durante la fase di inizializzazione dell'elaborazione, lo sviluppatore crea il manager fornendogli la lista dei punti di lavoro. Appena creato non ha infor-

mazioni rispetto la validità o la qualità dei punti di lavoro, per cui seleziona arbitrariamente il primo punto di lavoro della lista come migliore. Questa è considerata la *situazione base* per il framework. Da questo punto viene adottata una politica differenziale. Se viene aggiunto/rimosso/modificato un vincolo, definita una funzione di ranking o viene chiamato ad individuare i parametri migliori, il framework analizza la differenza tra la situazione in cui è stato deciso l'ultimo punto di lavoro migliore e la situazione attuale, decidendo se cambiare Operating Point oppure mantenere quello corrente.

Una funzionalità aggiunta rispetto al framework precedente è il concetto di *vincolo rilassabile*, ovvero un vincolo che non influenza la scelta del punto di lavoro, ma nel caso non venga rispettato esegue una chiamata di callback definita dalla sviluppatore. Lo scopo di questo vincolo è di fornire all'applicazione la capacità di adattarsi in un contesto distribuito. Si supponga che un'applicazione abbia una coda in ingresso di dati che deve elaborare e inviare ad un altro dispositivo, e che lo sviluppatore imponga un limite sulla velocità di elaborazione e sulla qualità minima, oltre all'utilizzo di un vincolo rilassabile sul riempimento della coda in ingresso. Si supponga inoltre che esista un monitor che osserva a Run-Time la velocità di elaborazione e lo stato della coda. Ad un certo punto si verifica un guasto al processore e la velocità di elaborazione si abbassa notevolmente. Il framework suggerirà all'applicazione i parametri che offrono il throughput maggiore per compensare, a scapito della qualità e del consumo energetico. Si supponga che questo cambiamento non fosse sufficiente a smaltire la mole di dati in ingresso. In questo caso la coda inizierà a riempirsi e il vincolo rilassabile non sarà più rispettato, per cui viene effettuata la chiamata di callback che notifica l'applicazione della situazione, che in risposta inoltra la richiesta per diminuire la mole di dati da elaborare.

Da questo esempio è possibile notare come i vincoli rilassabili siano uno strumento dell'applicazione da usare nel caso cambiare configurazione non fosse sufficiente a garantire l'adattabilità dell'esecuzione. Considerando che la chiamata di callback è bloccante, si suggerisce di utilizzare un approccio che assomiglia all'interrupt handling in un kernel GNU/Linux, ovvero mantenere la chiamata di callback "leggera" e affidare l'azione da intraprendere ad un'altra funzione più "pesante". Riprendendo l'esempio precedente, se la funzione di callback contenesse tutta la procedura di comunicazione, si avrebbero parecchi tempi morti, dovuti alla latenza dei messaggi, in cui il sistema non effettua nessuna elaborazione. Durante la progettazione del framework si è assunto che l'azione tipica effettuata da una chiamata di callback è quella di modificare alcuni flag dell'applicazione.

La metodologia presentata fino ad ora permette allo sviluppatore di esprime una singola affermazione riguardo alle caratteristiche che deve avere il punti di lavoro migliore. A volte può esserci la necessità di voler utilizzare diversi requisiti e preferenze in base allo stato dell'elaborazione. Per esempio se il dispositivo è alimentato dalla rete fissa il consumo energetico risulta una

metrica di importanza secondaria rispetto a quando utilizza delle batterie; oppure nel caso un'applicazione multimediale venga eseguita a tutto schermo oppure in finestra ci può essere o meno un vincolo sulla risoluzione minima richiesta. In generale un'applicazione può avere la necessità di cambiare le sue politiche decisionali.

Per questo motivo è stato introdotto il concetto di stato. Il framework permette di creare differenti manager che hanno differenti vincoli e preferenze, in modo da poterle cambiare facilmente a Run-Time senza un elevato overhead. Come vedremo nel Capitolo 6, aggiungere/rimuovere vincoli e definire una funzione di ranking, sono le operazioni più dispendiose e tendenzialmente andrebbero eseguite solamente nella fase di inizializzazione.

L'introduzione dello stato richiede una ridefinizione di punto di lavoro: oltre ai parametri e alle metriche, un punto di lavoro può dichiarare anche a quali stato appartiene. Se un punto di lavoro non indica tale informazione, viene automaticamente inserito in quello di *default*. Quando verrà creato il manager, dedurrà automaticamente dalla lista di Operating Point il numero di stati e distribuirà di conseguenza i punti di lavoro.

## 4.4 Interazione fra Monitor e AS-RTM

Nella sezione precedente è stato introdotto il vincolo, come un concetto singolo. Il framework tuttavia suddivide i vincoli in due categorie: i vincoli dinamici e quelli statici. I vincoli statici sono i vincoli che si basano esclusivamente sui valori contenuti nella lista di punti di lavoro e possono avere come soggetto sia metriche che parametri.

I vincoli dinamici si basano oltre che sui valori contenuti nella lista di Operating Point, anche sulle misurazioni effettuate dai monitor, per questo motivo possono avere come soggetto solamente le metriche. Sono questi vincoli che forniscono l'effettiva capacità di adattamento all'applicazione: è proprio la differenza tra la misura osservata e quella aspettata che permette al framework di adattarsi nel caso di guasti o cambiamenti nella piattaforma.

Nel capitolo precedente si è accennato che i punti di lavoro rappresentano la predizione delle prestazioni nel caso si usasse una particolare configurazione di parametri. Se è presente una differenza tra valore atteso e valore riscontrato, quello che viene effettuato è propagare linearmente tale differenza. Si supponga che nella configurazione corrente un'applicazione dovrebbe avere un throughput di $10\,^{\text{lavori}}/_{\text{sec}}$, ma i monitor registrano un throughput dimezzato; in questo caso viene dimezzato il valore del throughput anche degli altri punti di lavoro, in modo che se ci fosse un vincolo che impone un limite inferiore di $8^{\text{lavori}}/_{\text{sec}}$, verrebbe selezionata una configurazione che permette di raggiungere almeno $16^{\text{lavori}}/_{\text{sec}}$. L'idea alla base di questo ragionamento è che il motivo che ha causato tale differenza, penalizzerà nella

stessa misura anche le altre configurazioni, per cui selezionandone una che vada al doppio si otterrebbero i risultati voluti.

L'architettura di monitoring è concepita per estrarre delle valutazioni statistiche, come media e varianza, dagli ultimi dati osservati. Il numero di dati massimo da mantenere è definito dallo sviluppatore, si supponga che nell'esempio precedente il monitor utilizzi le ultime 100 misure per fornire un risultato e che venga selezionata la nuova configurazione che permette di raggiungere i $16^{\text{lavori}}/_{\text{sec}}$. Quello che succede al prossimo ciclo di elaborazione è che i monitor osservano 99 valori utilizzati con la precedente configurazione e solamente un valore con la nuova. Di conseguenza il framework osserva che le prestazioni sono ulteriormente dimezzate e cerca di selezionare una configurazione con throughput più alto; questo andamento continua fino a quando la media ritorna ad abbassarsi in seguito all'introduzione di nuovi valori. A questo punto i dati osservati dal monitor conterranno altissimi valori di throughput originati dall'uso di configurazioni ad alte prestazioni, per cui il framework noterà che le prestazioni sono migliori rispetto a quelle previste, cercando di conseguenza punti di lavoro che forniscano un throughput meno elevato. Tuttavia la media non inizierà a scendere fino a quando i nuovi valori non rimpiazzeranno quelli vecchi. Di conseguenza si innescherà un meccanismo di oscillazione certamente non desiderato. Per questa ragione ogni volta che si cambia Operating Point vengono cancellati i dati osservati dai monitor.

Generalmente lo sviluppatore è interessato alla media degli ultimi dati osservati, tuttavia può capitare che sia interessato ad altre caratteristiche statistiche come la varianza. In quest'ultimo caso per ottenere una misura corretta occorre avere a disposizione un numero minimo di dati. Per questa ragione nella creazione di un vincolo dinamico, viene offerta la possibilità allo sviluppatore di specificare il numero minimo di dati da utilizzare per considerare una misurazione validata. Se la misurazione non è valida, l'AS-RTM ignora tale valore e utilizza solamente il valore contenuto nella lista di punti di lavoro.

Un'ultima considerazione riguardo all'interazione fra Monitor e AS-RTM riguarda la funzione di rank. La sua definizione può comprendere anche una metrica utilizzata in un vincolo dinamico, di conseguenza osservata da un Monitor. Il framework non considera il rank come una funzione dinamica, ma viene calcolata staticamente utilizzando solamente i valori della lista di Operating Point, anche se si ha a disposizione la possibilità di monitorarle a Run-Time. Il motivo di questa scelta è conseguenza del fatto che si utilizza una propagazione lineare dell'errore effettuato a simulazione, per questa ragione il punteggio di rank verrebbe aumentato o diminuito della stessa quantità per tutti gli Operating Point. Considerando che il rank viene utilizzato per ordinare i punti di lavoro, il fatto che essi diventino più o meno buoni è irrilevante per il suo scopo. Il ragionamento sarebbe differente se si avesse la possibilità di misurare a Run-Time le prestazioni di tutte le

configurazioni simultaneamente.

## 4.5 Integrazione nell'applicazione

L'ultimo punto che rimane da descrivere è l'integrazione del framework nell'applicazione. Il primo passo da effettuare per utilizzare ARGO in un'applicazione generica è effettuare una Design of Experiments per individuare le metriche significate a caratterizzare le prestazioni e i parametri più adatti per influenzare la l'evoluzione dell'applicazione, al fine ottenere la lista dei punti di lavoro. Essi possono essere il risultato di una Design Space Exploration, oppure più semplicemente di un'attività di profiling.

Il passo successivo è decidere se è necessario utilizzare diversi stati, oppure se l'applicazione richiede solamente una modalità di funzionamento. Nel caso in cui si utilizzano gli stati occorre arricchire gli Operating Point specificando ogni punto a che stato appartiene. Successivamente si analizza il codice di elaborazione e si decide quali metriche è necessario monitorare in fase di Run-Time.

Completata questa fase di design si creano i Monitor e l'AS-RTM nell'inizializzazione dell'applicazione, definendone i vincoli e le preferenze per ogni stato. Nella parte dell'applicazione che effettua l'operazione è necessario solamente richiedere i parametri migliori prima di iniziare e indicare ai monitor i punti in cui effettuare le misure. Il framework si prende la responsabilità di fornire all'applicazione i parametri migliori rispetto alla situazione corrente.

Quella che è stata descritta sembra un'integrazione semplice, ma alcuni dettagli possono influenzare significativamente l'adattabilità all'applicazione. Il primo fattore essenziale è determinare correttamente i vincoli e preferenze che esprimo esattamente le intenzioni dello sviluppatore. Inoltre occorre decidere dove posizionare i monitor per misurare la quantità desiderata, per esempio se nella misura della velocità di elaborazione si tiene conto dell'overhead introdotto dal framework o meno. Anche la dimensione delle finestre di osservazioni dei monitor e il numero di dati necessari per considerare una misurazione valida influenzano notevolmente la sensibilità e il tempo di risposta del framework. Inoltre se si usano vincoli rilassabili è necessario implementare tutto il codice necessario per affrontare la situazione.

Nell'Appendice A verrà mostrata un'integrazione a livello di codice e descritto come il framework reagisce agli scenari più frequenti.

# Capitolo 5

# Architettura del framework

In questo capitolo verrà descritta dettagliatamente la soluzione implementata. In primo luogo verrà descritta l'architettura di monitoring evidenziando le differenze rispetto a quanto svolto nel lavoro di tesi precedente. Nella seconda sezione verrà invece trattato l'AS-RTM, descrivendone l'architettura e l'algoritmo implementato che rispecchia la soluzione descritta nel capitolo precedente.

## 5.1 Architettura di Monitoring

La progettazione e lo sviluppo di questo modulo è stata eseguita nel lavoro di tesi precedente [52], ed è stato necessario solamente modificarlo per adattarsi al nuovo AS-RTM presentato in questo elaborato di tesi. Per questa ragione la descrizione contiene le informazioni necessarie per comprendere il framework, evidenziando le differenze. Lo scopo di questo modulo è di fornire allo sviluppatore la possibilità di misurare una qualsiasi grandezza che ritiene interessante, poter esprimere dei Goal sulle proprietà statistiche degli ultimi dati osservati e poter verificare se sono rispettati. Quest'ultima funzionalità è necessaria nel caso lo sviluppatore non abbia intenzione di utilizzare un manager. Con il termine Goal si intende una condizione espressa all'interno dell'architettura di monitoring, con il termine vincolo si implica l'uso dell'AS-RTM.

La struttura di monitoring è stata pensata per essere modulare e offrire la possibilità di creare dei monitor ad hoc che misurano una grandezza dipendente dal contesto dell'applicazione. Per questo motivo l'idea base del design comprende l'uso di un template di Monitor, che racchiude tutte le funzionalità necessarie. Estendendo e specializzando tale template è possibile creare qualsiasi Monitor desiderato, avendo la garanzia che sia pienamente compatibile con il framework ARGO. Nel lavoro di tesi precedente è stata costruita una suite di monitor specializzati: il Time Monitor, per misurare dei tempi; il Throughput Monitor per misurare la quantità di dati elaborata

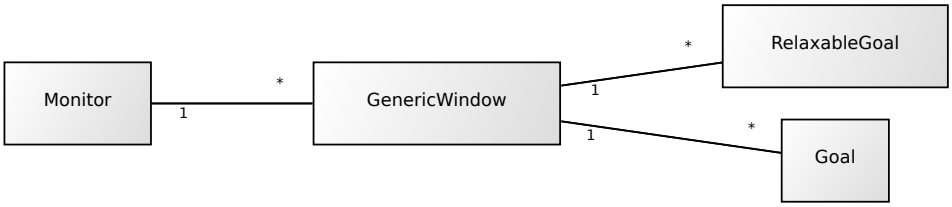



Figura 5.1: Architettura di un Monitor generico

in un determinato periodo; il Memory Monitor per misurare la quantità di memoria utilizzata dall'applicazione.

### 5.1.1 Monitor generico

Il cuore della struttura di monitoring è costituita da un template di Monitor, che contiene tutte le funzionalità necessarie a gestire la misura di una grandezza e l'interazione con l'AS-RTM. Nella Figura 5.1 è rappresentata l'architettura che compone tale template. Ogni Monitor può possedere diverse finestre di osservazione (GenericWindow), ognuna di esse si occupa di una singola misurazione. Se per esempio lo sviluppatore vuole controllare il tempo di esecuzione di due funzioni diverse, dovrà utilizzare un solo monitor (in entrambi i casi si misura un tempo) e due finestre di osservazione.

Ogni finestra permette a sua volta di creare/modificare/rimuovere dei Goal rispetto alla media, alla varianza, al valore massimo o minimo degli ultimi dati misurati. Per memorizzare le osservazioni ogni GenericWindow contiene un buffer circolare in cui verranno inserite le ultime misurazioni registrate in fase di Run-Time. Il dimensionamento di tale buffer è una scelta che influenza la sensibilità e la velocità di risposta della misurazione: avere a disposizione molti dati su cui calcolare la proprietà statistica interessata permette di essere immuni a fluttuazioni anomale delle misure, tuttavia rallenta la velocità di risposta, in quanto prima di registrare un cambiamento occorre che i nuovi dati rimpiazzino quelli vecchi. Per determinare il corretto dimensionamento occorre tenere in considerazione la variabilità della grandezza misurata. La finestra permette inoltre di impostare una maschera per ridurre il numero di dati utilizzati per calcolare la proprietà statistica desiderata.

Da un punto di vista funzionale, l'oggetto GenericWindow è l'attore principale dell'infrastruttura. Esso contiene tutti i metodi per la gestione del buffer, ovvero l'inserzione di nuovi dati, il ridimensionamento, la gestione della maschera per ridurre il numero di dati effettivamente utilizzati e offre la possibilità di estrarne direttamente le proprietà statistiche desiderate.

Da un punto di vista logico la finestra d'osservazione offre un approccio basato sui Goal e uno basto sui dati. Nel primo caso lo sviluppatore è interessato solamente a verificare il raggiungimento di traguardati predeterminati,

nel secondo caso vuole ottenere direttamente le proprietà statistiche dei dati osservati.

Nel caso venga usato un approccio orientato ai Goal, la finestra d'osservazione implementa una serie di metodi per verificare se i Goal sono stati rispettati, fornendo diversi gradi di informazione. La risposta più semplice consiste in un'indicazione binaria, se tale informazione non fosse sufficiente è possibile ottenere anche l'errore relativo e il NAP (Normalised Actual Penality). Quest'ultimo indicatore esprime l'elaborazione mancante al raggiungimento del Goal ed è calcolata con la seguente formula:

$$NAP = \left| \frac{x - G}{x + G} \right|$$

dove x è il valore osservato e G è il valore del goal fissato.

L'oggetto Monitor contiene tutte le funzioni necessarie per la gestione delle finestre d'osservazione, inoltre contiene una serie di metodi che richiamano le funzionalità della GenericWindow per permettere un maggiore livello di trasparenza rispetto alla finestra. La gestione dell'architettura di monitor è pensata per nascondere l'infrastruttura di oggetti appena descritti allo sviluppatore, richiedendo l'interazione solamente con l'oggetto di più alto livello, che nel caso di un monitor generico è il Monitor.

A differenza del lavoro precedente di tesi, l'architettura è stata modificata in modo che un Goal esprime solamente una sola condizione, per esempio "la varianza dei dati misurati deve essere minore o uguale a 1", ed è univocamente identificabile.

Per questo motivo è stato introdotto un metodo di numerazione per ogni Goal, non ambiguo rispetto a tutte le finestre di osservazione di altri monitor. Tale identificatore verrà nominato *ID_goal_globale* ed è un intero senza segno di 64 bit, dove la prima metà è l'identificatore della GenericWindow che lo contiene, la seconda metà è l'identificatore del Goal all'interno della finestra (*ID_goal_locale*). Per esempio il goal 4294967297 identifica il secondo Goal della seconda GenericWindow:

$$4294967297_{10} = \underbrace{\underbrace{000\ldots0001_2}_{ID\_finestra\,(32bit)}\ \underbrace{000...0001_2}_{ID\_goal\_locale\,(32bit)}}_{ID\_goal\_globale\,(64bit)}$$

L'uso di tale identificatore impone un limite al numero massimo di finestre d'osservazione utilizzabili e al numero massimo di Goal dichiarabili all'interno di ogni finestra, tuttavia è necessario per l'interazione con l'AS-RTM. L'identificatore delle GenericWindow è univoco rispetto a finestre che appartengono ad altri monitor, anche quelli personalizzati.

In questo modulo è stato introdotto inoltre il concetto di Goal rilassabile. Se lo sviluppatore decide di utilizzare solamente il modulo di Monitoring, in qualche punto della sua applicazione controllerà se tutti i Goal sono stati

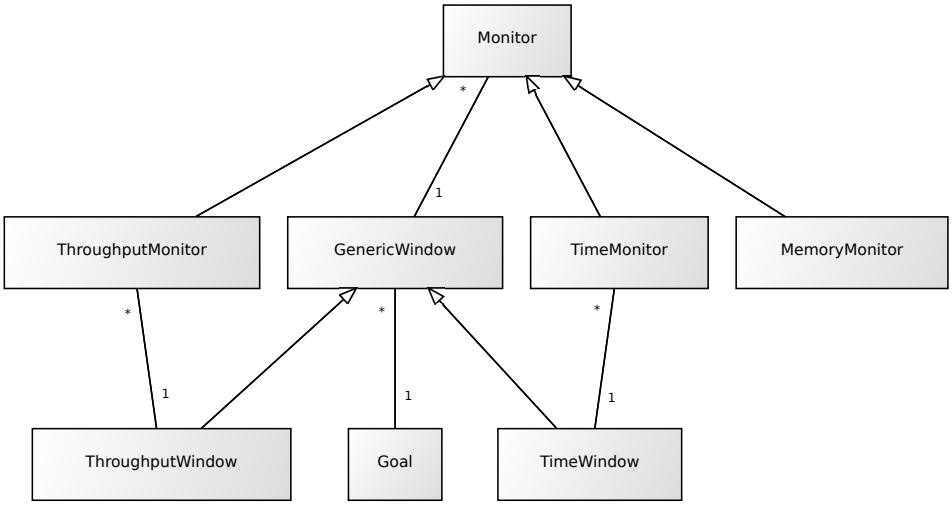



Figura 5.2: Architettura di Monitoring

rispettati. I Goal rilassabili non incidono sull'esito di tale verifica, ma nel caso non fossero rispettati eseguono una funzione di callback definita dallo sviluppatore stesso. Generalmente l'uso di questi goal è significativo se viene accoppiato con l'AS-RTM.

### 5.1.2 Monitor specializzati

La struttura descritta fino ad ora contiene le funzionalità per gestire le misurazioni. L'aspetto rimasto fino ad ora trascurato è l'effettiva misurazione della grandezza. In un monitor generico viene offerta la possibilità di inserire direttamente il valore osservato, un monitor specializzato invece si occupa di rendere più trasparente la misurazione. Considerando che il metodo di misura è dipendente dalla natura della grandezza da osservare, occorre specializzare il monitor generico. Nel precedente lavoro di tesi sono è stata prodotta una suite di monitor che considera la grandezze più comuni.

In Figura 5.2 è presente l'architettura di Monitoring completa, utilizzata come riferimento nella descrizione dei monitor specializzati.

#### 5.1.2.1 Time Monitor

Valutare le tempistiche di elaborazione è forse l'operazione di monitoring più comune. Come è possibile osservare nella Figura 5.2, è stato introdotto l'oggetto TimeMonitor, che permette la misurazione dell'intervallo di tempo. Analogamente al monitor generale, il TimeMonitor permette di essere usato in modalità stand-alone, ovvero senza dover specificare nessuna finestra di misurazione, sia utilizzando l'approccio orientato ai Goal.

La misurazione avviene impostando nel codice dell'applicazione le chiamate alle funzioni di start e stop, automaticamente il monitor calcola la differenza di tempo fra i due attimi con una precisione dell'ordine dei microsecondi. Nel caso si utilizzino delle finestre di osservazione è necessario specificare anche l'ID della finestra quando viene avviato e fermato il monitor.

Come è possibile notare dalla figura, è stato necessario introdurre l'oggetto TimeWindow. Il motivo di tale necessità deriva dal fatto che una normale GenericWindow non ha la possibilità di memorizzare l'attimo in cui è partito il cronometro.

#### 5.1.2.2 Throughput Monitor

Il throughput è un evoluzione concettuale rispetto al tempo, in quanto definisce la quantità di dati elaborati rispetto all'intervallo di tempo prefissato. La struttura di utilizzata è molto simile a quella del TimeMonitor, in quanto occorre sempre osservare un intervallo di tempo, dichiarando quanti dati sono stati elaborati. In particolare l'unità di misura temporale è espressa in millisecondi. Anche questo monitor permette l'esecuzione sia in modalità stand-alone sia utilizzando l'approccio orientato ai goal.

#### 5.1.2.3 Memory Monitor

A differenza del TimeMonitor e del ThroughputMonitor, il Memory Monitor appartiene alla tipologia di monitor orientata alla risorse. L'implementazione del MemoryMonitor deriva direttamente da una specializzazione del monitor generico.

La misurazione che ottiene consiste nella memoria allocata al tempo corrente ottenuta utilizzando le informazioni fornite dal sistema operativo. Anche questo monitor permette di utilizzare una modalità stand-alone oppure avvalersi delle finestre di osservazione.

Il MemoryMonitor permette anche l'utilizzo di una particolare funzione che ottiene dal sistema operativo la quantità massima di memoria virtuale utilizzata. Sebbene tale informazione non è adatta ad essere utilizzata come metrica nei punti di lavoro, può essere importante ai fini dell'attività di profiling, in quanto la memoria nei sistemi embedded può essere una risorsa critica.

## 5.2 Application-Specific Run-Time Manager

La parte più significativa del lavoro di tesi è stata la progettazione e lo sviluppo dell'Application-Specific Run-Time Manager. Lo scopo di questo modulo è quello di fornire all'applicazione i paramatri che appartengono al punto di lavoro migliore, come spiegato in dettaglio nel capitolo precedente.

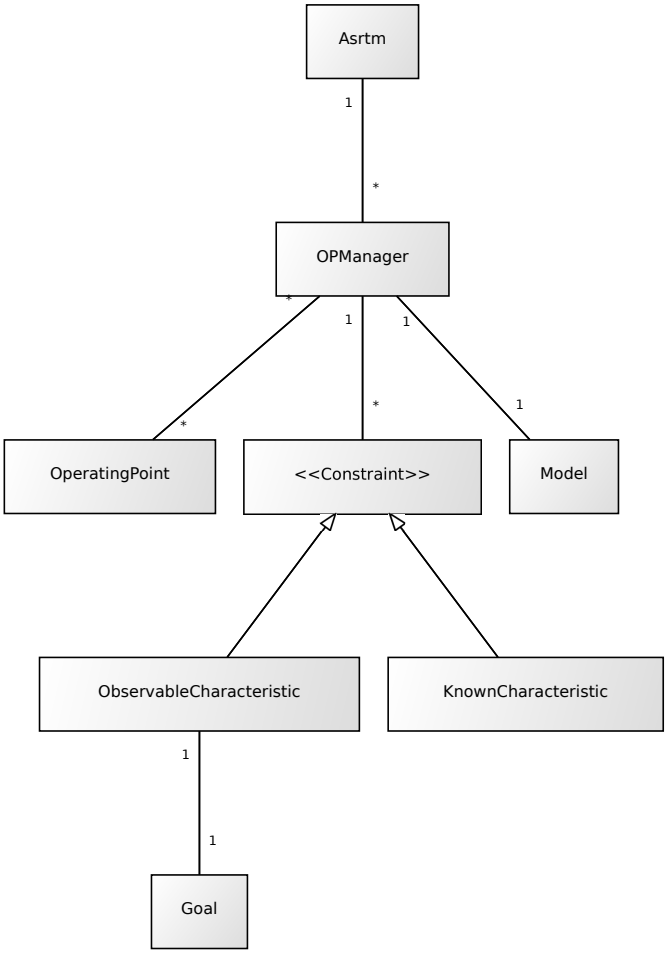



Figura 5.3: Architettura dell'Application-Specific Run-Time Manager

In Figura 5.3 è raffigurata la struttura dell'AS-RTM che costituisce l'implementazione della metodologia esposta, da utilizzare come riferimento durante la descrizione di tutti gli oggetti che compongono il modulo decisionale.

Il significato e utilità della lista di Operating Point è stata ampiamente descritta, prima di descrivere i componenti dell'AS-RTM occorre tuttavia descrivere a livello implementativo come è composta e come viene utilizzata. L'oggetto Operating Point è formato da tre liste distinte. La prima lista rappresenta i parametri applicativi ed è composta da tuple di stringhe e valori numerici. Esse associano il nome del parametro con il valore utilizzato nella configurazione rappresentata dal punto di lavoro. La seconda lista rappresenta le prestazioni dell' Operating Point ed ha la stessa struttura della lista dei parametri, con la differenza che le tuple associano il nome della metrica al valore raggiunto. L'ultima lista contiene gli stati a cui è associato il punto di lavoro. Il framework rappresenta lo stato con il suo nome. Nel momento che viene creato l'oggetto Asrtm, vengono scanditi tutti i punti di lavoro e vengono creati tanti OPManager quanti sono gli stati. Ogni oggetto di quest'ultimo tipo rappresenta uno stato e contiene effettivamente le informazioni che riguardano i punti di lavoro. Al momento della creazione dell'OPManager gli Operating Point vengono numerati seguendo l'ordine con cui appaiano nella lista. Tale numero costituisce l'identificatore univoco del punto di lavoro all'interno dell'OPManager. Tutti gli oggetti contenuti in quest'ultima classe utilizzano tale identificatore per riferirsi ad uno specifico punto di lavoro.

Come è possibile intuire dalla suddivisione dei punti di lavoro, il framework permette allo sviluppatore un approccio gerarchico al problema. Se vengono utilizzati più stati è necessario creare ed interfacciarsi con l'oggetto Asrtm, mentre se viene utilizzato un singolo stato è sufficiente utilizzare l'OPManager. Tale oggetto rappresenta il cuore del modulo decisionale e le sue componenti verranno descritte nelle sezioni successive.

### 5.2.1 Modello

Come accennato nella descrizione della metodologia, il framework utilizza un approccio differenziale per determinare il punto di lavoro migliore. Questa logica si basa sull'idea che la situazione corrente può peggiorare, migliorare o rimanere invariata rispetto all'ultima volta che si è cercato il punto di lavoro migliore. Per poter attuare questa politica l'OPManager memorizza l'ID dell'Operating Point corrente, e il modello della situazione.

Il modello utilizzato dal framework per rappresentare la situazione è composto da due insiemi distinti:

**S1** Il set S1 contiene gli identificatori di tutti gli Operating Point validi. Se tale set è vuoto significa che non ci sono punto di lavoro validi.

**S2** Il set S2 è un insieme temporaneo utilizzato dalla caratteristiche note e dalla caratteristiche osservabile per memorizzare i punti di lavoro ammissibili. La definizione di ammissibile verrà descritta in seguito.

Un modello appartiene ad un singolo OPManager ed è indipendente dagli altri. Se il set S1 contiene almeno un punto di lavoro, il modello viene chiamato possibile, viceversa se il set è vuoto il modello viene detto impossibile. Questi termini verranno utilizzati più avanti nella descrizione.

### 5.2.2 Caratteristica nota

Come descritto nella metodologia, il framework permette la definizione di vincoli sui parametri e su metriche, senza la necessità di osservarle a Run-Time. Lo scopo della caratteristica nota è quello di esprimere tale vincolo. È importante sottolineare che non è possibile esprimere dei vincoli sui parametri utilizzando l'infrastruttura di monitoring. Nell'implementazione effettuata, l'oggetto KnownCharacteristic rappresenta tale funzionalità, di seguito sono elencati gli attributi principali e il loro scopo:

**Informazioni sul vincolo** L'espressione di un vincolo richiede di specificare tre valori: il nome del parametro o della metrica oggetto, il tipo di comparazione e il valore del vincolo. Un esempio di vincolo si può tradurre nel linguaggio naturale in: "il numero di thread deve essere minore o uguale a 2".

**Valore degli OP** Questo attributo è un array della stessa dimensione dei punti di lavoro contenuti nel manager e contiene il valore della metrica e del parametro oggetto del vincolo. L'indice dell'array corrisponde all'identificatore del punto di lavoro.

**OP ordinati** Questo attributo è una lista di identificatori di punti di lavoro. Al momento della creazione dell'oggetto essa viene ordinata con l'obbiettivo di ottenere a bassi valori di indice i punti migliori. Di conseguenza se il vincolo è di minore o uguale gli Operating Point vengono ordinati in modo crescente, viceversa se il vincolo è di maggiore o uguale. Un'importante osservazione è che l'ordinamento di tale lista viene effettuata solamente durante la creazione del vincolo, in seguito non ci sono motivi per riordinare la lista.

**OP limite** Questo attributo è un puntatore all'Operating Point appartenente alla lista di OP ordinata, che ha indice più alto fra quelli che rispettano il vincolo. L'idea è di dividere la lista dei punti di lavoro in due insiemi distinti. Il primo insieme rappresenta i punti di lavoro che rispettano il vincolo, la seconda parte della lista rappresenta gli OP che non lo rispettano. Tale puntatore può slittare in avanti o indietro a seconda se la situazione migliora o peggiora. Se si trova in

coda alla lista significa che tutti i punti di lavoro rispettano il vincolo, se si trovano testa alla lista significa che il primo OP è l'unico valido, oppure che non ci sono punti di lavoro validi.

Come è possibile intuire dalla struttura della caratteristica nota, il concetto differenziale alla base della logica del framework viene replicato anche per i vincoli: data la situazione precedente, la situazione può solo migliorare, peggiorare o rimanere indifferente. In particolare se il valore del vincolo viene rilassato significa che il punto di lavoro a destra di quello limite potrebbe rispettare il vincolo; nel caso l'affermazione risultasse vera, viene spostato il puntatore dell'OP limite a destra, l'Operating Point viene aggiunto al set S2 del modello e si ripete il confronto fino a quando si incontra il primo OP che non soddisfa il vincolo. In questo modo è possibile verificare in modo efficiente la portata del miglioramento della situazione. Nel caso la situazione peggiorasse, si procederebbe in modo duale nella direzione opposta, rimuovendo di volta in volta i punti di lavoro dal set S1 (se sono presenti), notificando l'OPManager nel caso venga rimosso quello corrente.

I Punti di lavoro aggiunti al set S2 del modello si dicono *ammissibili*, ovvero sono tutti quei punti candidati ad essere validi. Ogni caratteristica nota esprime un vincolo, di conseguenza non è a conoscenza delle ulteriori restrizioni richieste. Il massimo che può fare è suggerire all'OPManager i punti di lavoro che rispetto alla situazione precedente, hanno iniziato a rispettare il vincolo.

La funzione che si occupa di effettuare tali operazioni si chiama *updateModel* ed è la funzione principale della caratteristiche, tipicamente invocata dall'OPManager quando deve selezionare il punto di lavoro migliore. Quest'ultima procedura verrà descritta in seguito.

### 5.2.3 Caratteristica osservabile

L'oggetto ObservableCharacteristic è complementare rispetto la caratteristica nota: la caratteristica osservabile esprime un vincolo osservato a RunTime. Gli attributi che possiede sono gli stessi della KnownCharacteristic, con l'aggiunta dell'informazione che associa il vincolo al goal del monitor.

L'informazione necessaria consiste nel puntatore alla finestra d'osservazione che contiene il Goal oggetto del vincolo, e l'ID_locale dello stesso. Se nella caratteristica nota l'unico fattore che può indurre un cambiamento nella situazione è la modifica del valore del vincolo, nella caratteristica osservabile entra in gioco la differenza tra il valore contenuto nel punto di lavoro corrente e il valore misurato dai monitor. Questo fatto implica che la situazione attuale differisca rispetto alla situazione precedente per questi due fattori che possono essere modificati contemporaneamente. Per esempio l'ObservableCharacteristic può notare che è stato cambiato il valore del goal e i monitor registrano delle prestazioni differenti rispetto a quelle previste.

Per gestire questa situazione le due differenze vengono sommate algebricamente (in accordo con la funzione di comparazione) in modo da stabilire se la situazione globale è cambiata effettivamente, riportandosi al caso descritto nella sezione precedente.

Nel caso si è osservata una differenza tra il valore misurato a Run-Time e quello contenuto nel punto di lavoro, la metodologia impone di modificare il valore di tutti gli altri Operating Point. Per motivi di efficienza non si effettua tale azione, ma ogni valore degli Operating Point viene pesato con un coefficiente di correzione $c_{osservazione}$ calcolato con la seguente formula:

$$c_{osservazione} = \frac{valore\ osservato}{valore\ atteso}$$

Nell'Algoritmo 5.1 viene riportato il metacodice dell'algoritmo della funzione *updateModel*. Nella prima parte, dal rigo 1 al rigo 15, viene calcolata la differenza totale della situazione corrente rispetto a quella precedente e viene di conseguenza aggiornato $c_{osservazione}$. La parte di codice compresa tra la linea 16 e 23 si occupa di gestire il peggioramento della situazione, ovvero quando esiste la possibilità di rimuovere punti di lavoro dal modello. Nel caso venga rimosso il punto di lavoro corrente viene notificato l'OPManager utilizzando il dato restituito dalla funzione. Nell'ultima parte della funzione viene gestito il miglioramento della situazione, ovvero la possibilità che alcuni punti di lavoro diventino ammissibili. I parametri $v_{osservato}$, $\psi$ e $g_{attuale}$ non sono attributi della classe, tuttavia possono essere recuperati facilmente utilizzando la classe GenericWindow. Se all'algoritmo illustrato si toglie la parte riguardante i dati osservati, la struttura ricalca quello seguito dalla caratteristica nota.

### 5.2.4 Vincolo

Nelle sezioni precedenti si è visto come le caratteristiche note e quelle osservabili rappresentino, rispettivamente, i vincoli statici e dinamici introdotti nella descrizione della metodologia. Dal punto di vista dell'oggetto OPManager i due tipi di vincoli sono identici. Per questo motivo i due oggetti implementano l'interfaccia *vincolo*, che contiene tutte i metodi utilizzati dal manager. Come illustrato nel Class Diagram mostrato in Figura 5.4, i metodi necessari per gestire i vincoli sono:

**isAdmissible** Questo metodo controlla se l'OP ricevuto in ingresso rispetta il singolo vincolo. Il valore restituito contiene la risposta binaria.

**updateModel** Questo metodo è stato descritto nelle sezioni precedenti, il valore restituito indica se l'OP corrente non rispetta il vincolo.

**checkRelaxable** Entrambe le caratteristiche hanno la possibilità di esprimere vincoli rilassabili, la cui funzione è stata descritta nella metodologia proposta. Nel caso il vincolo non sia rispettato, viene eseguita la

**Algorithm 5.1** Algoritmo che implementa la funzione *updateModel*

**Attributi**

- Il vettore dei valori noti degli Operating point $v$
- Il valore osservato a Run-Time $v_{osservato}$
- la funzione di comparazione $\psi$
- Il valore del goal attuale $g_{attuale}$
- Il valore del goal precedente $g_{precedente}$
- Il coefficiente di correzione $c_{osservazione}$

**Input**

- Il puntatore $M$ al Modello
- Il punto di lavoro $o_{corrente}$ corrente

**Output**

- La validità dell'Operating Point corrente

1: **begin**
2: **if** la finestra di osservazione è valida **then**
3: $d_{osservazione} = v_{osservato} - v[o_{corrente}] \cdot c_{osservazione}$
4: $c_{osservazione} = \frac{v_{osservato}}{v[o_{corrente}]}$
5: **else**
6: $d_{osservazione} = 0$
7: **end**
8: **if** $\psi \in \{<; \leq\}$ **then**
9: $d_{osservazione} = -d_{osservazione}$
10: **end**
11: $d_{goal} = g_{precedente} - g_{attuale}$
12: **if** $\psi \in \{>; \geq\}$ **then**
13: $d_{goal} = -d_{goal}$
14: **end**
15: $d_{totale} = d_{osservazione} + d_{goal}$
16: **if** $d_{totale} < 0$ **then**
17: **while** $(v[o_{limite}] \cdot c_{osservazione})\, \bar{\psi}\, g_{attuale}$ **do**
18: $M \rightarrow S1 = M \rightarrow S1 \setminus o_{limite}$
19: **if** $o_{limite} = o_{attuale}$ **then**
20: $o_{corrente}$ non è più valido
21: **end**
22: $o_{limite} = prev(o_{limite})$
23: **end**
24: **else**
25: **while** $(v[next(o_{limite})] \cdot c_{osservazione})\, \psi\, g_{attuale}$ **do**
26: $o_{limite} = next(o_{limite})$
27: $M \rightarrow S2 = M \rightarrow S2 \cup o_{limite}$
28: **end**
29: **end**
30: **return** la validità di $o_{corrente}$

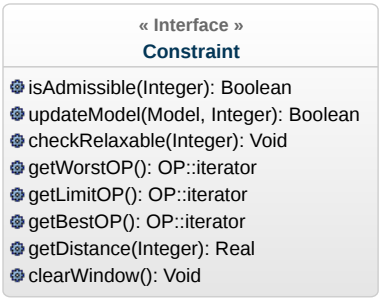



Figura 5.4: Class Diagram dell'interfaccia Constraint

chiamata di callback definita dallo sviluppatore. Nel caso di vincoli statici è necessario specificare il punto di lavoro corrente.

**getWorstOP** Questo metodo restituisce il puntatore dell'ultimo Operating Point contenuto nella lista dei punti di lavoro ordinati. Ovvero il peggior punto di lavoro per quel vincolo.

**getBestOP** Questo metodo restituisce il puntatore del primo OP contenuto nella lista dei punti di lavoro ordinati. Ovvero il miglior punto di lavoro per quel vincolo.

**getLimitOP** Questo metodo restituisce il puntatore del punto di lavoro limite contenuto nella lista dei punti di lavoro ordinati. Ovvero il peggior punto di lavoro valido. Se non ne esistono di validi, restituisce il primo punto di lavoro, ovvero il migliore.

**getDistance** Questo metodo restituisce la distanza che separa l'Operating Point specificato in ingresso dal valore del vincolo. L'utilizzo di questo metodo è circoscritto al caso in cui non esistono punti di lavoro validi.

**clearWindow** Questo metodo svuota la finestra di osservazione della caratteristica osservabile. Nel caso il vincolo sia statico, la funzione non effettua nessuna operazione.

La modifica del valore del vincolo è una funzione che dipende dalla natura del vincolo stesso. Se si tratta di un vincolo statico è il manager l'attore che l'applicazione utilizza per modificarlo, se invece si vuole modificare un vincolo dinamico l'applicazione deve interagire direttamente con il goal oggetto del vincolo. Il motivo di tale scelta è mantenere una coerenza tra l'AS-RTM e la struttura di monitoring.

### 5.2.5 OPManager

Nelle sezioni precedenti sono stati descritti tutti componenti che vengono utilizzati dal manager degli Operating Point, permettendo di introdurre l'ogget-

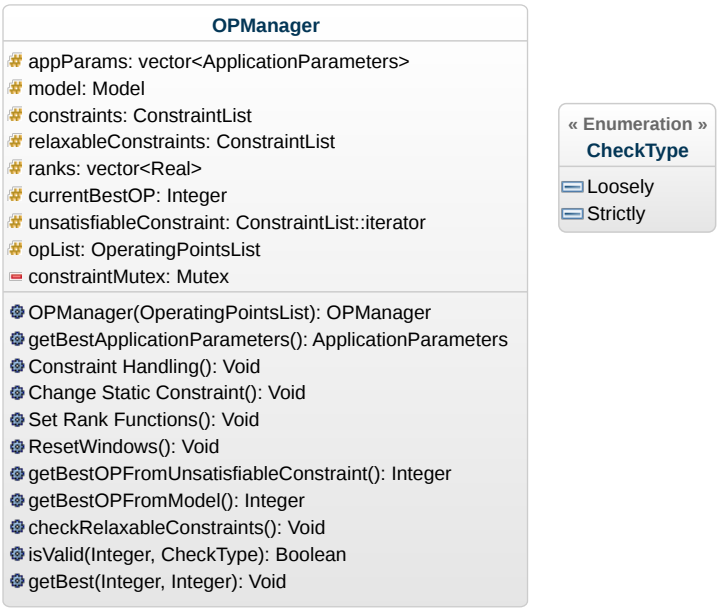



Figura 5.5: Class Diagram dell'OPManager

to più importante del framework. Se la GenericWindow costituisce il cuore dell'architettura di monitoring, l'OPManager lo è per l'AS-RTM. In Figura 5.5 è raffigurato il Class Diagram della classe OPManager, per chiarezza sono stati raggruppati alcuni metodi in base alla loro funzionalità.

Come è possibile osservare dal diagramma, il manager utilizza due liste differenti per i vincoli normali e quelli rilassabili. Il motivo di questa scelta risiede nel metodo utilizzato per esporre allo sviluppatore la gestione dei vincoli. Considerando che la trasparenza dell'integrazione del framework nell'applicazione è sempre stata ritenuta un fattore importante, per gestire l'ordine di priorità dei vincoli lo sviluppatore utilizza un indice numerico. L'OPManager associa al vincolo con indice 0 la massima priorità. Di conseguenza la classe espone allo sviluppatore una serie di funzioni per aggiungere/rimuovere vincoli in base ad una posizione numerica, in aggiunta quelle che indicano il posizionamento in testa o in coda, richiedendo di specificare il minor numero di informazioni possibile e in maniera del tutto trasparente rispetto alle caratteristiche osservabili o note. È infatti responsabilità del manager la creazione di tali oggetti.

Per questa necessità è stata divisa la lista dei vincoli rilassabili, dalla lista dei vincoli normali. La funzione *checkRelaxableConstraints* si occupa di verificare ogni vincolo rilassabile. Nella loro definizione nel capitolo precedente, si è portato l'esempio di un esigenza a Run-Time. Tuttavia per non perdere di generalità, si è fornita allo sviluppatore la possibilità di definire dei vincoli rilassabili su grandezze non osservate dai monitor. Nel diagramma le funzioni che aggiungono/rimuovono entrambi i vincoli sono state raggruppate

nel metodo *Constraint Handling*.

Precedentemente si è affermato che è l'OPManager l'oggetto che permette allo sviluppatore di poter modificare il valore dei vincoli statici. Per effettuare tale operazioni il framework espone una serie di metodi che richiedono in ingresso la posizione del vincolo e il nuovo valore del vincolo, o incremento relativo. Nel diagramma tali funzioni sono state raggruppati nel metodo *Change Static Constraint*.

Nella descrizione della metodologia si è accennato alla funzione di ranking, spiegandone lo scopo. Il framework permette di definire tre tipi di funzione di rank a seconda delle esigenze dello sviluppatore:

**Semplice** Il punteggio di rank viene attribuito ad una sola metrica o parametro. Per esempio se lo sviluppatore decide che il punto di lavoro migliore è quello che minimizza il consumo di energia, il framework selezionerà l'OP che ha il valore di metrica inferiore, fra quelli validi.

**Lineare** La funzione di rank è una combinazione lineare di metriche o parametri, viene utilizzata nel caso lo sviluppatore decida che la qualità di un punto di lavoro sia formata da più metriche. Per definire tale funzione, lo sviluppatore dovrà dichiarare quali metriche sono interessate e un coefficiente che ne determina l'importanza. Per esempio si assuma che si vuole massimizzare la qualità di elaborazione e minimizzare il tempo di esecuzione, favorendo leggermente la qualità rispetto alla velocità. In questo caso la funzione di rank del punto di lavoro $i$ sarà di massimizzazione e avrà la forma:

$$r_i = \underbrace{0.6}_{c_1} \cdot qualità_i + \underbrace{(-0.4)}_{c_2} \cdot velocità_i$$

Dove con $qualità_i$ e $velocità_i$ si indica il valore delle rispettive metriche presenti nel punto. Il vantaggio di usare questa formulazione consiste nella possibilità di utilizzare grandezze discordanti, lo svantaggio risiede nel fatto che le grandezze prese in considerazione devono essere comparabili per avere senso. Per ovviare a questo problema è sufficiente riscrivere i coefficienti $c_1$ e $c_2$ in modo da normalizzare la misura. I coefficienti dell'esempio precedente diventano quindi:

$$c_1 = \frac{0.6}{qualità_{max}} \qquad c_2 = \frac{-0.4}{velocitò_{max}}$$

Dove $qualità_{max}$ e $velocità_{max}$ sono i valori massimi delle rispettive metriche presenti in tutta la lista di Operating Point.

**Geometrica** Il rank è determinato da una funzione geometrica, in cui lo sviluppatore deve dichiarare le metriche interessate e l'esponente che ne determina l'importanza. Per esempio se si vuole minimizzare l'errore

e la velocità di elaborazione, favorendo le prestazioni, la funzione di rank del punto di lavoro $i$-esimo avrà la seguente forma:

$$r_i = errore_i^2 \cdot velocità_i^3$$

Il vantaggio di usare la funzione geometrica è che si possono mettere in relazione anche metriche non comparabili; lo svantaggio risiede nel fatto che non è banale, da un punto di vista numero, unire metriche in contrasto fra di loro utilizzando diversi esponenti. Il risultato dipende se l'intervallo di valori che possono assumere le metriche passa per l'origine, invertendosi di segno.

Nel caso nessuna di queste funzioni sia sufficiente a permettere allo sviluppatore di definire le sue preferenze è possibile definire una metrica calcolata che contiene i valori numerici desiderati e utilizzare una funzione di rank semplice. Nel diagramma le funzioni che si occupano di definire il rank sono state raggruppate nel metodo *Set Rank Functions*.

Durante la descrizione della metodologia è stato definito il concetto di migliore, il quale dipendeva dalla presenza o meno di punti di lavoro validi. Se il set S1 del modello contiene almeno un Operating Point, ottenere il punto di lavoro migliore significa solamente effettuare una ricerca di massimo rispetto al loro punteggio di rank. Il metodo *getBestOPFromModel* effettua questa operazione, restituendo l'identificatore del punto trovato.

Il discorso è più interessante nel caso il modello è impossibile. Prima di descrivere l'algoritmo utilizzato è necessario spiegare come il modello può svuotarsi. Al momento della creazione del manager non sono ancora stati introdotti dei vincoli, di conseguenza tutti i punti di lavoro sono validi. Il modello può diventare impossibile solamente nel momento in cui vengono utilizzati dei vincoli, in quanto la caratteristica ad esso associato può eliminare gli Operating Point appartenenti al set S1. Nel framework la lista dei vincoli viene visitata sempre in ordine, partendo dal vincolo di priorità maggiore, per cui si definisce inizialmente *vincolo insoddisfacibile* (nel diagramma rappresentato da *unsatisfiableConstraint*), il vincolo che invalida l'ultimo punto di lavoro contenuto in S1. Per il framework non è importante se il modello diventa impossibile in seguito alla creazione di un vincolo o durante l'esecuzione dell'applicazione, il fatto significativo è la causa che porta tutti gli Operating Point ad essere invalidi. Esse sono solamente due: nel primo caso un vincolo diventa talmente stringente che la caratteristica ad esso associata svuota il set S1; nel secondo è una combinazione di vincoli, individualmente soddisfacibili, che rende il modello impossibile.

Nel primo caso è evidente che la lista di vincoli viene partizionata in due insiemi ben distinti: il primo contiene tutti i vincoli di priorità maggiore che ammettono almeno un punto di lavoro, mentre il secondo insieme contiene i restanti vincoli con a capo quello che non permette l'esistenza di Operating Point validi: il vincolo insoddisfacibile.

Nel secondo caso purtroppo non è più possibile individuare il vincolo che divide tali insiemi. Il motivo è semplice: si supponga per esempio che lo sviluppatore abbia imposto cinque vincoli e che il modello risulti possibile al tempo $t$. In questa situazione è verosimile che ogni vincolo abbia rimosso qualche punto di lavoro dal modello. Si supponga inoltre che al tempo $t+1$, venga cambiato il valore del terzo vincolo abbastanza da eliminare anche l'ultimo punto di lavoro valido. Secondo la definizione è lui il vincolo insoddisfacibile. Il problema è che non è più vero che divide la lista di vincoli: può esistere un punto di lavoro valido per i primi quattro vincoli (anche nel tempo $t+1$), ma che non sia ammissibile per il quinto, di conseguenza non era presente in S1 al tempo $t$.

Quello che occorre fare è applicare la definizione di migliore nel caso il modello sia impossibile, ovvero un punto di lavoro è migliore rispetto ad un altro se soddisfa il maggior numero di vincoli contigui, iniziando da quello con maggiore priorità. In caso di parità, si prende il punto con rank maggiore. Il metodo *getBest* applica tale metodo, unito al caso in cui il modello è possibile, per determinare il migliore fra due punti. L'unica strategia non ancora definita è decidere quali punti confrontare.

Sfruttando il fatto che le caratteristiche dividono già gli Operating Point in ammissibili e non, viene ridotto di conseguenza il numero di OP da confrontare. Rimane da decidere quale vincolo utilizzare per guidare la ricerca. Per massimizzare la probabilità che la caratteristica selezionata contenga pochi punti di lavoro ammissibili, si procede in questo modo:

1. Si determina il vincolo insoddisfacibile usando la definizione, ovvero il vincolo che svuota il set S1 del modello.

2. Se non esiste un Operating Point ammissibile per il vincolo insoddisfacibile e se non è il vincolo di priorità massima, viene decrementato.

3. Per costruzione si ricade in uno di questi due scenari:

   (a) Esiste almeno un punto di lavoro ammissibile per il vincolo insoddisfacibile da utilizzare per trovare il migliore.

   (b) Il vincolo insoddisfacibile individuato usando la definizione non ammette nessun Operating Point, ed è il vincolo di priorità massima. L'unica azione da effettuare è selezionare il proprio OP migliore, ovvero il primo punto contenuto nella caratteristica.

Il metodo *unsatisfiableConstraint* implementa tale metodologia per individuare il punto di lavoro migliore partendo dal vincolo insoddisfabile. Una conseguenza derivata dall'usare il vincolo insoddisfacibile come guida per confrontare i punti di lavoro è l'introduzione di un nuovo tipo di validità. L'ultimo metodo definito nel caso si trovi nella situazione *a*, scorre tutti i punti ammissibili e confronta solamente quelli che rispettano anche i

vincoli soprastanti, ignorando quelli di priorità inferiore. Questo tipo di ricerca più permissiva viene chiamata *loosely*, mentre la ricerca di un punto valido viene definita *strictly*. Per questa ragione il metodo *isValid* richiede come parametro, oltre l'identificatore del punto di lavoro, anche il metodo di validazione richiesta.

L'unico metodo che rimane da definire è *getBestApplicationsParameters*, il metodo più importante fornito dal framework. Lo scopo di tale metodo è quello di fornire all'applicazione i migliori parametri rispetto alla situazione attuale. L'Algoritmo 5.2 descrive il metacodice del metodo.

La prima operazione che esegue è controllare tutti i vincoli rilassabili, per motivi di spazio tale istruzione non è stata riportata nel metacodice. L'algoritmo si divide in tre parti distinte. Nella prima parte (fino al rigo 10), si cicla su tutti i vincoli, partendo da quello di piorità maggiore, con lo scopo di aggiornare il modello alla situazione corrente (istante $t$). Alla fine di questo processo il set S2 conterrà tutti i punti di lavoro che sono stati aggiunti da almeno una metrica, mentre il set S1 conterrà gli Operating Point validi. In questa parte del codice si verifica anche se il punto di lavoro corrente è ancora ammissibile per i vincoli significativi. Nel caso il modello al tempo $t-1$ era impossibile, i vincoli significativi sono solamente quelli con priorità maggiore rispetto a quello insoddisfacibile. In $validità_{o_{corrente}}$ viene indicata la risposta binaria.

La seconda parte dell'algortimo (fino alla linea 17) ha lo scopo di trovare il miglior punto di lavoro appartenente al set S2 del modello. La natura di tale Operating Point dipende dallo stato del modello. L'identificatore $o_{miglioreValidoAggiunto}$ rappresenta il miglior punto di lavoro valido contenuto in S2. Analogamente $o_{miglioreAmmissibileAggiunto}$ rappresenta il miglior punto di lavoro valido in modo loosely aggiunto. Mentre vengono scanditi gli elementi di S2, se sono validi vengono promossi nel set S1, se non sono validi vengono eliminati. In ogni circostanza, al termine di questa parte dell'algortimo, S2 non deve contenere alcun elemento.

L'ultima parte dell'algoritmo è la vera parte decisionale, in quanto in base alle informazioni raccolte deve selezionare l'azione da intraprendere. Di seguito verranno elencate tutte le situazioni possibili e l'azione intrapresa dal framework.

- Se al tempo $t-1$ il modello era possibile e dopo aver aggiornato il modello, il set S1 rimane non vuoto, possono accadere le seguenti situazioni:
  - L'Operating Point corrente è ancora valido ed è migliore di $o_{miglioreValidoAggiunto}$, allora non occorre modificare $o_{corrente}$
  - Viene aggiunto un punto di lavoro valido migliore di quello corrente, $o_{miglioreValidoAggiunto}$ diventa il nuovo punto di lavoro migliore, indipendentemente dalla validità (strictly) di $o_{corrente}$, in quanto

**Algorithm 5.2** Algoritmo che implementa *getBestApplicationParameters*

**Attributi:**
- Il vincolo $c_{insoddisfacibile}$ non soddisfabile
- Il modello $M$ del sistema
- L'id $o_{corrente}$ dell'operating point corrente
- La lista $C$ dei vincoli

**Output**
-La lista dei parametri applcativi migliore per l'applicazione

1: **begin**
2: **if** $M \rightarrow S1 = \emptyset$ **then**
3: $M^{t-1}$è impossibile
4: **for all** $c \in C$ **do**
5: $validità_{proporsta} = c \rightarrow updateModel(M, o_{corrente})$
6: **if** $c \prec c_{usatisfiable}$ **then**
7: $validità_{o_{corente}} = validità_{o_{corrente}} \wedge validità_{proposta}$
8: **if** $(M \rightarrow S1 \neq \emptyset) \wedge (M^{t-1}\ è\ possibile) \wedge (M^{t}\ è\ impossibile)$ **then**
9: $c_{insoddisfacibile} = c$
10: $M^{t}\ è\ impossibile$
11: **for all** $o \in M \rightarrow S2$ **do**
12: **if** $isValid(o, strictly)$ **then**
13: $o_{miglioreValidoAggiunto} = getBest(o, o_{miglioreValidoAggiunto})$
14: **else**
15: **if** $(M^{t-1}\ è\ impossibile) \wedge (\nexists o_{miglioreValidoAggiunto})$ **then**
16: **if** $isValid(o, loosely)$ **then**
17: $o_{miglioreAmmissibileAggiunto} = getBest(o, o_{miglioreAmmissibileAggiunto})$
18: **if** $(M^{t-1}\ è\ possibile) \wedge (M^{t}\ è\ possibile)$ **then**
19: **if** $\exists o_{miglioreValidoAggiunto}$ **then**
20: $o_{proposto} = getBest(o_{miglioreValidoAggiunto}, o_{corrente})$
21: **if** $(\neg validità_{o_{corrente}}) \wedge (o_{proposto} = o_{corrente})$ **then**
22: $o_{proposto} = getBestOPFromModel()$
23: **else**
24: **if** $\exists o_{miglioreValidoAggiunto}$ **then**
25: $o_{proprosto} = o_{miglioreValidoAggiunto}$
26: **else**
27: $o_{proposto} = o_{corrente}$
28: **if** $(M^{t}\ è\ impossibile) \wedge (\exists o_{miglioreAmmissibileAggiunto})$ **then**
29: $o_{proposto} = getBest(o_{miglioreAmmissibileAggiunto}, o_{corrente})$
30: **if** $(\neg validità_{o_{corrente}}) \wedge (o_{proposto} = o_{corrnte})$ **then**
31: $o_{proposto} = getBestOPFromUnsatisfiableConstraint()$
32: **if** $o_{proposto} \neq o_{corrente}$ **then**
33: $o_{corrente} = o_{proprosto}$
34: $resetWindows()$
35: **return** $params[o_{corrente}]$

per la proprietà transitiva è migliore anche di tutti i punti presenti in S1.

- Se $o_{corrente}$ non è più valido, e $o_{miglioreValidoAggiunto}$ non è migliore dell'Operating Point attuale oppure non esiste, occorre cercare il nuovo punto di lavoro migliore nel set S1.

- Se al tempo $t-1$ il modello era possibile, ma al tempo $t$ non lo è più, l'unica azione da intraprendere è utilizzare il metodo *getBestOPFromUnsatisfiableConstraint* descritto precedentemente.

- Se in entrambi gli istanti di tempo il sistema è impossibile, possono verificarsi le seguenti possibilità:

  - Se viene aggiunto almeno un punto di lavoro valido, $o_{miglioreValidoAggiunto}$ diventa il nuovo Operating Point corrente.
  - Se non esiste nessun $o_{miglioreValidoAggiunto}$, e $o_{miglioreAmmissibileAggiunto}$ è migliore di $o_{corrente}$, esso diventa il nuovo punto di lavoro corrente, a prescindere dalla validità (loosely) di $o_{corrente}$.
  - Se non esiste nessun $o_{miglioreValidoAggiunto}$, e $o_{miglioreAmmissibileAggiunto}$ non è migliore di $o_{corrente}$ oppure non esiste, e $o_{corrente}$ non è più valido (loosely), occorre cercare un nuovo Operating Point dal metodo *getBestOPFromUnsatisfiableConstraint* descritto precedentemente.
  - Se non esiste nessun $o_{miglioreValidoAggiunto}$, e $o_{miglioreAmmissibileAggiunto}$ non è migliore di $o_{corrente}$ oppure non esiste, e $o_{corrente}$ è ancora valido (loosely), non viene cambiato il punto di lavoro corrente.

Ogni volta che viene selezionato un nuovo punto di lavoro, vengono cancellate le osservazioni effettuate dalle finestre contenute nei vincoli dinamici definiti. Come è possibile osservare dalle azioni intraprese, si è cercato di sfruttare al massimo le informazioni ricavate dalle strutture dati. La complessità dell'algoritmo è $\Theta(m \cdot n)$ dove $m$ è il numero di metriche e $n$ è il numero di punti di lavoro, tuttavia generalmente il numero di operazioni effettuate è minore.

### 5.2.6 Asrtm

L'ultimo oggetto da analizzare è l'Application-Specific Run-Time Manager, l'attore di più alto livello del framework. Se lo sviluppatore ha intenzione di utilizzare gli stati è necessario interagire con questo oggetto. Nel momento della sua creazione, il costruttore effettua la cernita dei punti di lavoro creando una lista di Operating Point per ogni stato, usate per creare il respettivo OPManager. Quest'ultimo oggetto ignora infatti qualsiasi informazione riguardante lo stato.

L'Asrtm contiene la lista di manager che rappresentano i singoli stati ed espone i metodi per cambiare stato. Ogni volta che viene effettuata tale operazione, si eliminano i dati dalle finestre di osservazione contenute nei vincoli dinamici. Di conseguenza ogni oggetto OPManager rappresenta uno stato. Per facilitare l'integrazione nell'applicazione, tutte le funzionalità del manager vengono offerte anche dall'Asrtm. In questo modo è necessaria solo l'interazione con quest'ultimo oggetto per effettuare tutte le operazioni necessarie.

# Capitolo 6

# Risultati sperimentali

In questo capitolo verranno descritte le prove sperimentali effettuate per validare il framework sviluppato. Il primo tipo di validazione è di tipo funzionale: è stata scritta un'applicazione che attraverso una suite di test, ha l'obbiettivo di verificare il comportamento corretto dell'AS-RTM descritto nei capitoli precedenti. Il secondo tipo di validazione riguarda le prestazioni fornite da ARGO. Il framework fornisce a un'applicazione la capacità di adattarsi, ma tale caratteristica comporta un costo. In questo capitolo verranno mostrati i vantaggi e gli svantaggi dell'uso del framework. Si mostreranno in primo luogo i risultati ottenuti da un benchmark che esegue le operazioni descritte nel capitolo precedente, operando nel loro caso peggiore, in modo da fornire un limite superiore dell'overhead introdotto. In seguito verranno mostrati degli esempi applicativi in cui è stato integrato il framework, con lo scopo di rappresentare il comportamento del framework nel caso venga applicato in uno scenario reale.

## 6.1 Set-up sperimentale

Il primo passo per validare i risultati del framework è verificare che l'implementazione segua fedelmente la metodologia descritta nei capitoli precedenti. L'applicazione utilizzata per verificare il comportamento di ARGO si basa sul framework di testing sviluppato da google (Google Test [2]). I test effettuati coinvolgono i maggiori attori che hanno un ruolo nel processo decisionale e per ognuno di essi viene controllato oltre il comportamento desiderato, anche la corretta gestione di errori derivati da uno scorretto utilizzo del framework.

L'uso di tale applicazione non è stata solamente utile ai fini dello sviluppo del framework, ma permette di verificarne il funzionamento in una determinata piattaforma. Per questo motivo l'applicazione di testing è fornita assieme al codice di ARGO ed è consigliata la sua esecuzione ogni volta che viene compilato il framework in un nuovo ambiente. Se tutti i test forniscono esito positivo, significa che il framework funzionerà correttamente.

Una volta effettuata tale validazione si può procedere ad analizzare le prestazioni di ARGO. Prima di descrivere gli esperimenti effettuati, in questa sezione viene riassunta la fase di setup eseguita prima di effettuare le prove. La piattaforma utilizzata per eseguire tutte le operazioni è un comune portatile, che utilizza un processore i7-2620M che opera a 2.7Ghz. La versione di gcc utilizzata è la 4.8.1 fornita con il sistema operativo Fedora 19. Per validare le prestazioni del framework viene utilizzata un'applicazione di benchmark, in aggiunta a degli scenari applicativi, dove sono state selezionate delle applicazioni appartenenti alla suite di benchmark PARSEC e integrate nel framework stesso.

L'applicazione di benchmark è inclusa in un ramo dei sorgenti del framework e produce un'eseguibile standalone in grado di effettuare tutti test che verranno descritti nella sezione successiva, fornendo i risultati direttamente in formato csv. Nella valutazione dello scenario applicativo, vengono effettuati due tipi di test per ogni applicazione valutata.

Il primo test avvia quattro istanze dell'applicazione integrata con ARGO, con lo scopo di misurare la capacità di adattamento acquisita e il tempo di overhead del processo decisionale. In questo tipo di esperimento sono state definite una combinazione di vincoli e preferenze che non permettono l'esecuzione simultanea di più istanze senza alterare la configurazione iniziale.

La seconda tipologia di test esegue una singola applicazione in cui viene richiesto, con il passare del tempo, prestazioni (intese come velocità di elaborazione) sempre maggiori. L'idea alla base di questo prova è effettuare uno stress test con l'obbiettivo di osservare come il framework cerca di soddisfare tale richiesta, utilizzando gli strumenti che ha a disposizione. La richiesta è stata dimensionata in modo che verso la fine dell'esecuzione dell'applicazione, il framework non abbia più opzioni per soddisfare tale domanda.

Prima di poter integrare ARGO nelle applicazioni selezionate, è stata fatta un'analisi dei parametri esposti, per capire quali siano i più adatti per essere utilizzati come Dynamic Knobs, ovvero per influenzare l'elaborazione. Il passo successivo è stato decidere quali metriche siano necessarie per definire la prestazione di una configurazione di parametri. Infine si è svolta un'attività di profiling atta ad individuare la lista di punti di lavoro necessaria al funzionamento del framework. I dettagli implementativi rimanenti sono dipendenti dall'applicazione utilizzata e verranno forniti nelle sezioni successive.

## 6.2 Prestazioni e analisi di Scalabilità

Una volta effettuata la validazione funzionale del framework, rimane da misurare le sue prestazioni. Il primo problema affrontato è individuare il limite superiore dell'overhead introdotto dal framework. Questo valore è utile per comprendere i limiti di applicabilità di ARGO in applicazioni critiche. Per ef-

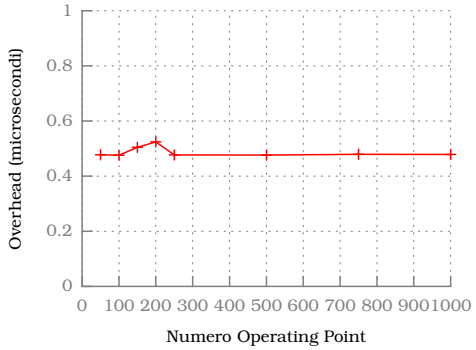



Figura 6.1: Ritardo minimo del framework

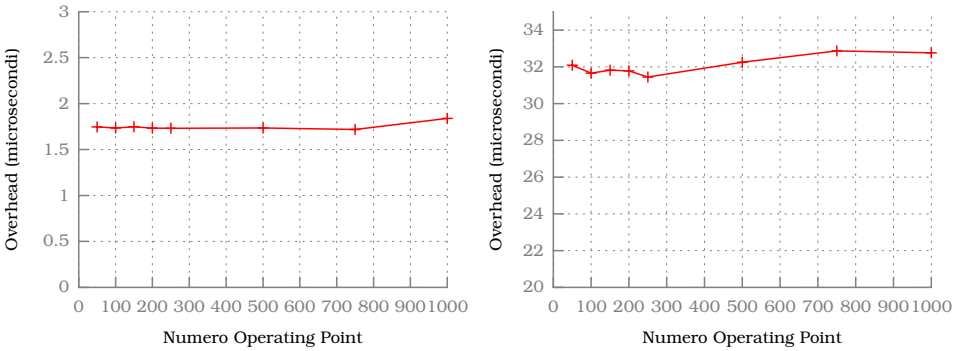
fettuare queste misurazioni è stata scritta un'applicazione di benchmark che esegue le operazioni di più alto livello nella loro condizione peggiore, variando il numero di punti operativi. In questo modo è possibile determinare empiricamente la complessità delle operazioni.

Prima di elencare i test effettuati e mostrane i risultati è necessario descrivere come vengono effettuati. Tutti i test vengono eseguiti su dei manager distinti che contengono diversi numeri di punti di lavoro. Tutti i risultati dei test non sono frutto di una singola esecuzione, ma vengono ripetuti più volte e ne viene calcolata la media. In questo modo è possibile ottenere dei risultati più consistenti. Ogni test quindi effettua l'operazione oggetto di valutazione, ne viene misurato il tempo di esecuzione e infine viene eseguita l'operazione inversa per riportare il manager nello stato precedente. I risultati mostrati in seguito sono stati ottenuti con il seguente procedimento: ogni test viene ripetuto cento volte e sono stati utilizzati sette manager, con 50, 100, 150, 200, 250, 500, 750 e 1000 Operating Point.

L'applicazione di benchmark viene fornita assieme al codice del framework ed è stata pensata per essere personalizzata facilmente riguardo il numero di OP da valutare e il numero di volte in cui ripetere l'esperimento. Gli scenari misurati dal benchmark sono i seguenti:

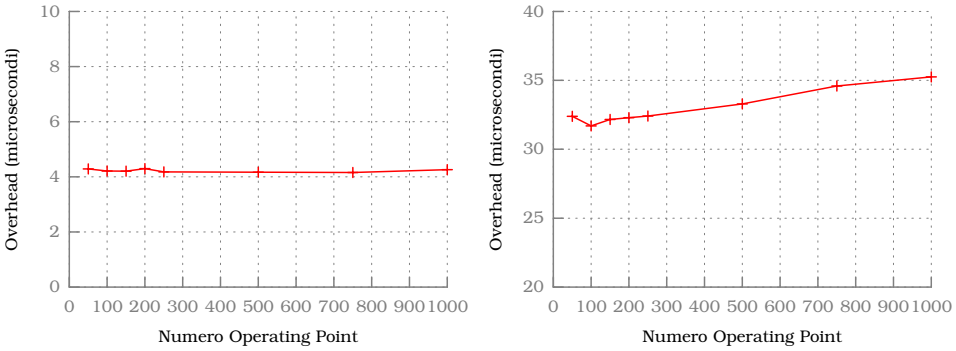
### 6.2.1 Ritardo minimo

Lo scenario descritto in questa sezione misura il tempo che impiega il framework a decidere il punto di lavoro migliore nel caso non ci siano cambiamenti. Per questa regione viene considerato il ritardo minimo introdotto. In Figura 6.1 vengono mostrati i risultati ottenuti. Come è possibile osservare. il ritardo minimo è praticamente trascurabile, di conseguenza possiamo affermare che se non presenti cambiamenti nella situazione, il framework non introduce ritardi, indipendentemente dal numero di punti di lavoro.




(a) Rimozione di 10 OP senza finestra

(b) Rimozione di 10 OP con finestra




(c) Aggiunta di 10 OP senza finestra

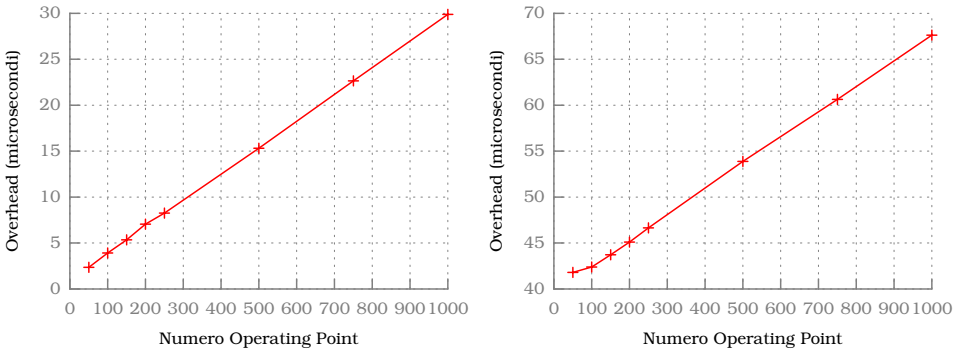
(d) Aggiunta di 10 OP con finestra

Figura 6.2: Aggiunta/Rimozione di 10 OP

### 6.2.2 Aggiunta/rimozione di dieci Operating Point

Lo scopo di questi test è verificare il tempo impiegato dal framework nell'aggiungere o rimuovere dieci Operating Point sub ottimi dal set S1 del modello. Il significato dell'aggettivo sub ottimo dipende dall'operazione considerata: nel caso di rimozione di Operating Point indica il fatto di non rimuovere quello corrente; nel caso di aggiunta di Operating Point indica che tutti i punti aggiunti abbiano punteggio di rank inferiore rispetto a quello attuale. Tale scenario è suddiviso in due casi: nel primo si utilizza una caratteristica nota per effettuare tale cambiamento, nel secondo caso una caratteristica osservabile dove sono presenti cento osservazioni generate casualmente.

Come è possibile notare dalla Figura 6.2, l'overhead dell'operazione è determinata in modo prevalente dalla presenza della finestra di osservazione. Di conseguenza per migliorare le prestazioni occorre essere parsimoniosi con il numero di caratteristiche osservabili utilizzate nei vincoli. Si può inoltre osservare che la complessità dell'operazione non è in relazione con il numero di punti di lavoro totali, ma è proporzionale al numero di Operating Point coinvolti nel cambiamento.




(a) Senza utilizzare una finestra (b) Utilizzando una finestra

Figura 6.3: Rimozione punto di lavoro corrente

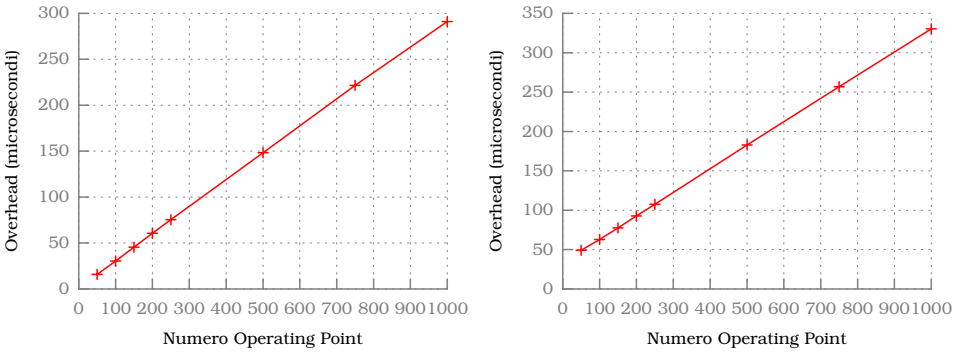
Questo risultato è molto buono, in quanto evidenzia come l'overhead introdotto dal framework nel caso di cambiamenti che non coinvolgono il punto di lavoro corrente è trascurabile. Dalla figura è possibile notare come il tempo di aggiunta di Operating Point validi sia maggiore di quello impiegato nella rimozione, tale effetto è conseguenza del fatto che i punti aggiunti da un vincolo devono essere giudicati ammissibili anche dai restanti vincoli prima di essere considerati validi.

### 6.2.3 Rimozione punto di lavoro corrente

In questo test viene rimosso il punto di lavoro corrente, ovvero quello migliore. Il caso peggiore si ha quando tutti i punti di lavoro sono validi, perché questa situazione obbliga il framework a valutare tutti i restanti punti nella ricerca del nuovo punto di lavoro migliore. Anche in questo caso vengono separati i casi in cui viene o non viene utilizzata una finestra di osservazione.

Nella Figura 6.3 è mostrato l'overhead introdotto dal framework. Come è possibile osservare esso ha una complessità lineare rispetto al numero di Operating Point contenuti nel modello, che nel caso peggiore equivale al numero totale dei punti di lavoro. L'unico modo per migliorare tale risultato è mantenere ordinato il modello rispetto al punteggio di rank, tuttavia in questa implementazione del framework si è preferito evitare questo approccio per due ragioni: nel caso generale i punti di lavoro validi sono un sottoinsieme di quello totale e se si dovesse ridefinire la funzione di rank in fase di Run-Time occorrerebbe riordinare tale punti, teoricamente con una complessità maggiore di quella lineare.

Non è stato effettuato l'esperimento duale rispetto alla rimozione del punto di lavoro migliore, ovvero l'aggiunta di un nuovo Operating Point migliore, in quanto tale operazione non prevede azioni differenti rispetto alla normale aggiunta di punti di lavoro misurata nelle sezioni precedenti.




(a) Senza utilizzare una finestra (b) Utilizzando una finestra

Figura 6.4: Passaggio al modello impossibile

### 6.2.4 Passaggio al modello impossibile

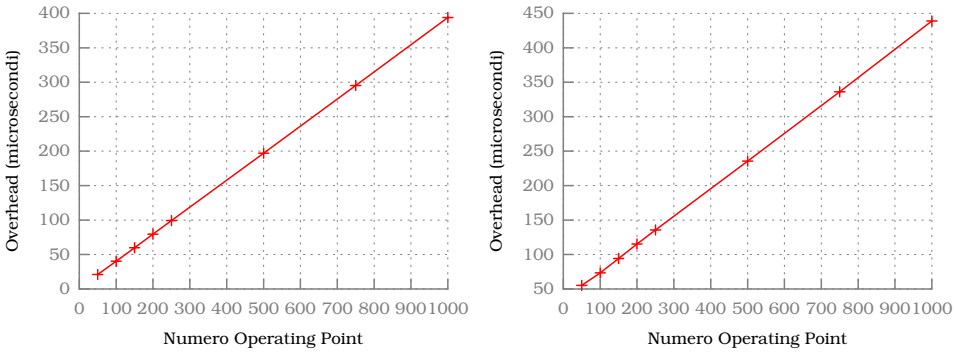
In questo esperimento si vuole replicare lo scenario in cui partendo da un modello possibile (istante $t$), viene rimosso l'ultimo punto di lavoro valido (istante $t+1$). Per emulare il caso peggiore occorre utilizzare due metriche. Nell'istante $t$ tutti i punti di lavoro rispettano entrambi i vincoli. Nell'istante $t+1$ il vincolo di priorità inferiore elimina tutti i punti dal modello. Come reazione il framework dovrà cercare fra tutti i punti di lavoro ammissibili per il vincolo soprastante, quello che più si avvicina al valore del vincolo inferiore. In questo modo è obbligato a considerare tutti gli Operating Point due volte. Ancora una volta vengono considerati i casi con e senza finestra d'osservazione.

In Figura 6.4 sono mostrati i risultati ottenuti. Come è possibile osservare anche in questo caso la complessità è lineare rispetto ai punti di lavoro ammissibili per il vincolo insoddisfacibile, che nel caso peggiore coincidono con il numero di Operating Point totali.

### 6.2.5 Passaggio al modello possibile

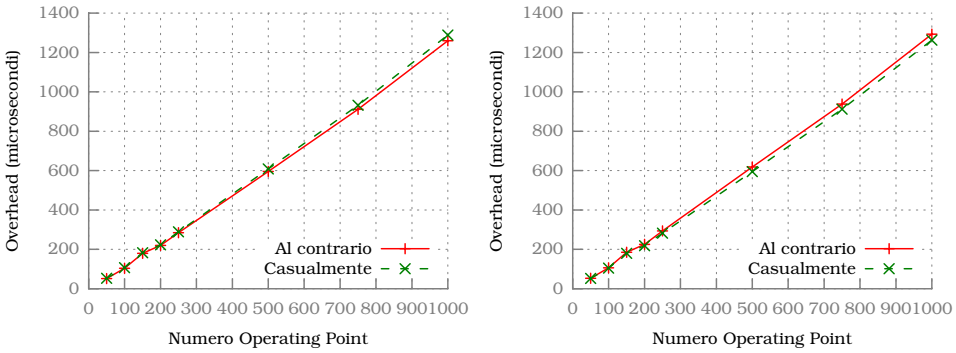
In questa sezione si mostrano i risultati dello scenario duale rispetto al precedente, ovvero partendo da un modello impossibile (istante $t$), si passa ad avere almeno un punto di lavoro valido (istante $t+1$). Il caso peggiore si verifica quando tutti i punti di lavoro passano ad essere validi. In questo caso infatti oltre a diventare ammissibili per un vincolo, il framework deve controllare che essi rispettino anche i restanti vincoli prima di essere giudicati validi.

In Figura 6.5 sono mostrati i risultati ottenuti. In questo caso è possibile osservare che la complessità dell'operazione è lineare rispetto al numero di Operating Point divenuti validi, che nel caso peggiore equivale al numero totale dei punti di lavoro, moltiplicati per il numero di vincoli.




(a) Senza utilizzare una finestra (b) Utilizzando una finestra

Figura 6.5: Passaggio al modello possibile




(a) Risultati con un vincolo statico (b) Risultati con un vincolo dinamico

Figura 6.6: Creazione di un vincolo

### 6.2.6 Aggiunta di vincoli

Nell'ultimo scenario considerato dal benchmark, viene osservata la creazione di un vincolo generico. Considerando che l'operazione effettuata più dispendiosa è quella di ordinare i punti di lavoro secondo il tipo di vincolo, come descritto nel capitolo precedente, per determinare il caso peggiore occorre studiare l'algoritmo di ordinamento. Il framework utilizza il procedimento fornito dalle librerie standard di C++, che può variare a seconda dell'implementazione della libreria. Per essere generici vengono considerati due casi possibili: il primo dove gli Operating Point vengono creati in ordine inverso rispetto a quello desiderato, nel secondo caso vengono generati casualmente. Inoltre vengono considerati il caso in cui il vincolo sia statico e il caso in cui il vincolo sia dinamico.

In Figura 6.6 vengono mostrati i risultati raggiunti. Sorprendentemente la complessità sembra essere lineare, tuttavia da specifica dovrebbe avere

complessità $\Theta(n \cdot \lg(n))$. In figura è possibile osservare come l'overhead nel caso di vincolo dinamico e statico sia molto simile, tale risultato era prevedibile in quanto nel momento della creazione del vincolo non vengono considerati i dati osservati della finestra, essi vengono presi in considerazione quando il framework viene chiamato a decidere il miglior punto di lavoro, rientrando negli scenari considerati precedentemente. Paragonando quest'ultima operazione con le altre, aggiungere un vincolo introduce un overhead tre volte maggiore rispetto alla seconda operazione più dispendiosa nel caso peggiore. Per queste ragioni è consigliato rimuovere/aggiungere vincoli solamente in fase di inizializzazione dell'applicazione, evitando se possibile di farlo durante l'elaborazione.

## 6.3 Scenari applicativi

Nella sezione precedente si è descritto uno strumento per ottenere il limite superiore dell'overhead introdotto dal framework, mostrando i risultati ottenuti. Lo scopo di questa sezione è cercare di mostrare i vantaggi e il prezzo nell'integrare il framework in una situazione generica, fornendo i risultati raggiunti con tre applicazioni reali appartenenti alla suite PARSEC. Per riuscire a rappresentare un caso d'uso generico, le applicazioni sono state selezionate in modo da mostrare un aspetto interessate dell'integrazione con il framework.

### 6.3.1 Swaptions

L'obbiettivo di questa applicazione è di calcolare il valore di un portfolio di swaptions finanziarie, basandosi sul framework Heath-Jarrow-Morton [6]. La particolarità del modello utilizzato è che non può essere risolto analiticamente utilizzano delle equazioni differenziali, quindi l'applicazione utilizza una simulazione Monte Carlo per stabilire il valore di tutto il portfolio. Come parametri prevede la possibilità di specificare il numero di thread da utilizzare, il numero di simulazioni da effettuare e un seed da utilizzare per la generazione pseudo-casuale delle caratteristiche delle swaptions nel portfolio. I parametri applicativi adatti ad influenzare la fase di Run-Time sono solamente il numero di thread impiegati e il numero di simulazioni, in quanto il seed non influenza le prestazioni.

Prima di integrare il framework ARGO è stato necessario modificare il comportamento dell'elaborazione. Invece di calcolare un numero fisso di swaptions che compone il portfolio e terminare l'esecuzione, viene simulato uno stream di azioni finanziarie che deve essere valutato dall'applicazione. Per ottenere tale scopo si è trasformato il numero di swaptions che compone il portfolio nel numero di elementi processati contemporaneamente (ognuno affidato ad un thread), ed è stato creato un ciclo esterno che simula la durata dello stream da elaborare.

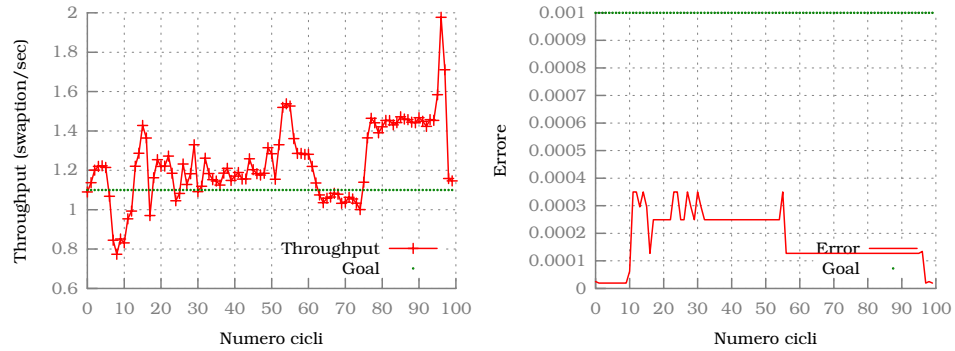
Utilizzando questo approccio si ricalcola più volte il valore delle swaptions, tuttavia si è effettuata tale modifica in quanto non è importante il valore del risultato ottenuto, ma mostrare la capacità di adattamento offerta dal framework e l'overhead introdotto dallo stesso. La conseguenza interessante derivata dall'utilizzare questo approccio è che all'inizio di ogni ciclo vengono creati i thread e distrutti alla fine. Considerando che la creazione di un thread è un'operazione onerosa, nel test sull'adattabilità viene riscontrato un comportamento interessante. Questa applicazione è stata scelta per evidenziare come il metodo di attuazione del cambiamento dei parametri applicativi influisca sul comportamento dell'applicazione.

#### 6.3.1.1 Adattabilità rispetto alle prestazioni

In questa prova si cerca di mostrare come il framework permetta all'applicazione di rispettare i goal stabiliti senza avere il bisogno di coordinarsi con altri attori. Per ottenere questo scopo si avvia un'istanza dell'applicazione ogni venti secondi e si tiene traccia delle grandezze rilevanti. Per caratterizzare le prestazioni dell'applicazione vengono misurati il throughput dell'applicazione e l'errore commesso nell'elaborazione. L'errore è calcolato come differenza rispetto al risultato ottenuto con i parametri che permetto la miglior qualità possibile. Per misurare le prestazioni del framework si misura il tempo impiegato a decidere la configurazione di parametri migliore e il tempo di esecuzione vero e proprio.

All'applicazione viene imposto un vincolo sull'errore massimo che può commettere e sul throughput che fornisce. Viene utilizzata una funzione di rank geometrica che cerca di minimizzare in ugual misura il numero di thread utilizzati e l'errore commesso. Per evidenziare le azioni che il framework intraprende viene riportato anche il numero di thread proposto. Le politiche appena espresse non vengono modificate in fase di Run-Time, i cambiamenti ottenuti sono frutto del cambiamento dell'ambiente di esecuzione che fa da contesto all'applicazione stessa.

In Figura 6.7 sono presenti i risultati ottenuti dal framework dalla prima applicazione eseguita. Come è possibile osservare il vincolo sul throughput viene rispettato per la maggior parte del tempo, tuttavia nell'intorno del decimo istante di tempo rimane insoddisfatto per molti istanti di tempo. In Figura 6.7c è possibile vedere come l'applicazione cerchi di compensare tale mancanza aggiungendo thread e diminuendo il numero di punti utilizzati per effettuare la simulazione (l'errore aumenta), ovvero effettuando tutte le azioni che ha a disposizione. Come è possibile vedere dalle immagini l'andamento delle caratteristiche osservate è abbastanza movimentato, tale effetto si genera in quanto sono state lanciate diverse istanze dell'applicazione che si autoconfigurano in maniera indipendente, influenzandosi a vicenda. La causa di tale influenza è derivata dal fatto che l'attuazione delle modifiche decise



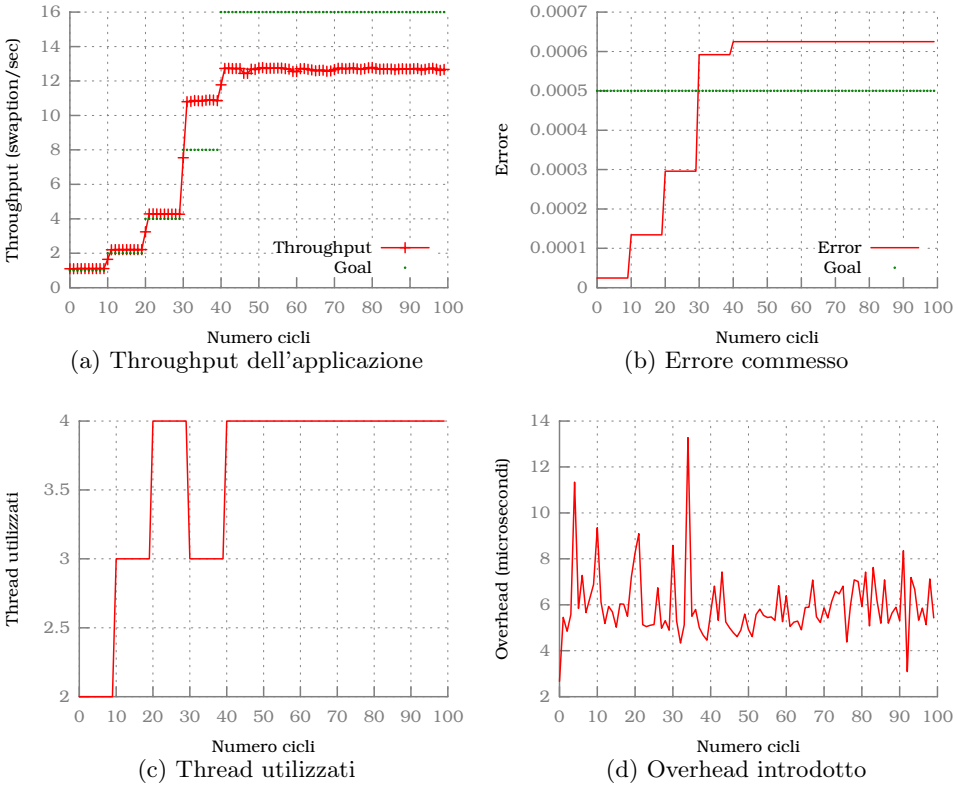

(a) Throughput dell'applicazione

(b) Errore commesso

(c) Thread utilizzati

(d) Overhead del framework

Figura 6.7: Adattabilità in swaptions




(a) Throughput dell'applicazione
(b) Errore commesso
(c) Thread utilizzati
(d) Overhead introdotto

Figura 6.8: Stress test per swaptions

dall'AS-RTM sono molto onerose, provocando oscillazioni nelle prestazioni del sistema, che si riflettono in oscillazioni decisionali.

Per questo motivo è importante nella fase di progettazione dell'applicazione prevedere delle tecniche di attuazione dei cambiamenti che non comportino operazioni molto onerose computazionalmente.

#### 6.3.1.2 Stress test

In questa prova si considera una sola istanza dell'applicazione a cui viene aumentato periodicamente il vincolo che esprime il throughput minimo che deve raggiungere, mantenendo invariato il vincolo sull'errore commesso. In questo modo è possibile osservare le azioni intraprese dal framework. Le politiche decisionali sono le stesse dell'esperimento precedente, si è modificato solamente i valori dei vincoli.

In Figura 6.8 sono raffigurati i risultati ottenuti dal framework. A differenza del caso precedente tali risultati sono molto più stabili e lineari, in quanto l'attuazione dei cambiamenti non è sufficiente ad alterare la capacità computazionale del sistema. Come è possibile notare al trentesimo istante

il modello diventa impossibile, in quanto non viene più rispettato il vincolo sull'errore. Dopo tale istante, il framework rinuncia a soddisfare tale vincolo per cercare di inseguire quello di priorità maggiore. Tuttavia è possibile osservare come il punto di lavoro selezionato cerca di restare sempre vicino al valore del vincolo sull'errore, rispettando correttamente la metodologia proposta.

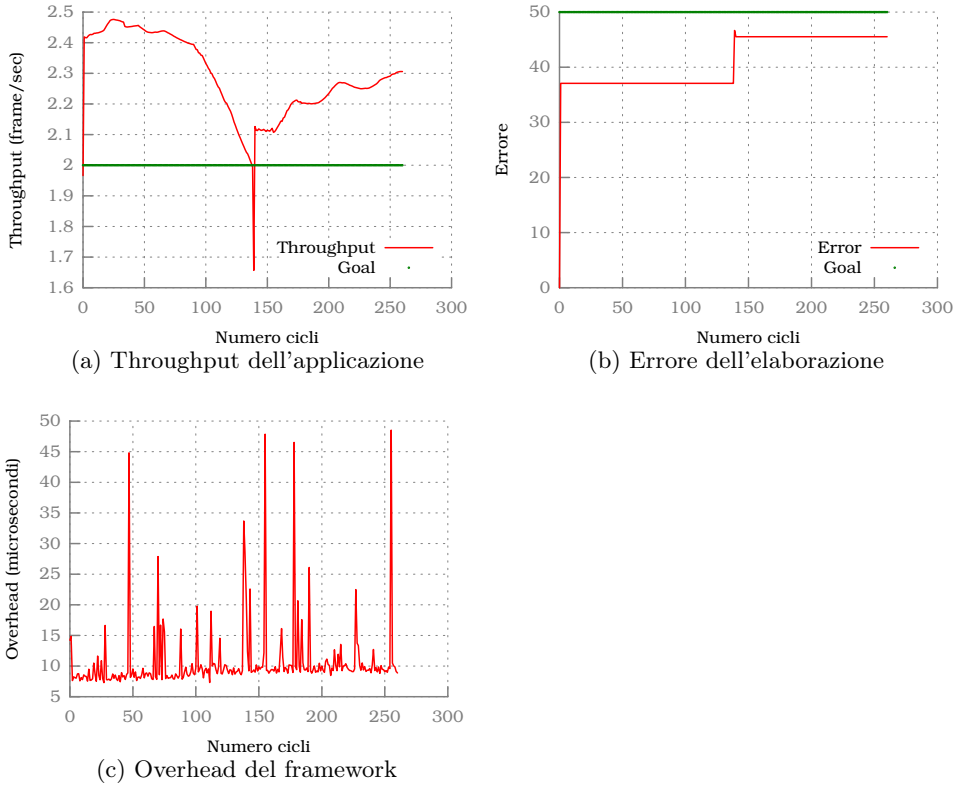
### 6.3.2 Bodytrack

Questa applicazione ha lo scopo di individuare la posa di un corpo umano in tre dimensioni, utilizzando le informazioni di quattro telecamere. Per ottenere tale risultato utilizza un filtro particellare per individuare i contorni del corpo umano evidenziati dalla differenza rispetto l'immagine di background, basandosi sulla cinematica di un modello tridimensionale del corpo umano, composto da dieci segmenti. L'output dell'applicazione è la posizione e l'angolo delle giunture di tale modello scritte in un file di testo e disegnate direttamente sulle immagini delle telecamere.

L'algoritmo prevede la possibilità di variare diversi parametri, tuttavia quelli più adatti ad influenzare il comportamento dell'applicazione sono tre: il numero di thread che compongono un gruppo di lavoro, il numero di particelle utilizzate e il numero di livelli da considerare nell'elaborazione. A differenza dell'applicazione swaptions, l'attuazione del cambiamento del numero di thread risulta più semplice in quanto è presente il concetto di gruppo di lavoro. Purtroppo nel codice originale la dimensione di quest'ultimo oggetto non era previsto che variasse, è stato quindi necessario modificare alcune parti dell'applicazione per implementare tale capacità. Il cambiamento dei restanti punti di lavoro è più immediato.

Le metriche considerate per caratterizzare la prestazione dell'algoritmo sono: il numero di frame elaborati in un secondo (fps) e l'errore di elaborazione. Quest'ultima qualità è calcolata in modo analogo all'applicazione precedente, ovvero come differenza rispetto al risultato ottenuto con la configurazione che garantisce il miglior risultato. A differenza dell'applicazione precedente l'errore commesso con una configurazione di scandente qualità è parecchio distante da quella utilizzata come riferimento.

La caratteristica che contraddistingue questa applicazione è la natura dell'influenza del cambiamento dei parametri rispetto alla qualità dell'elaborazione. Nell'applicazione precedente utilizzare pochi punti nella simulazione comportava immediati risultati inferiori, rendendo ogni elaborazione indipendente. In questo caso invece la qualità dell'elaborazione precedente influisce sulla qualità dell'elaborazione attuale. In particolare diminuendo il numero di particelle, o livelli, utilizzati, si avvia una progressiva diminuzione della qualità. Questo particolare rende l'individuazione della qualità dell'elaborazione ottenuta in una configurazione dipendente dal procedimento utiliz-




(a) Throughput dell'applicazione

(b) Errore dell'elaborazione

(c) Overhead del framework

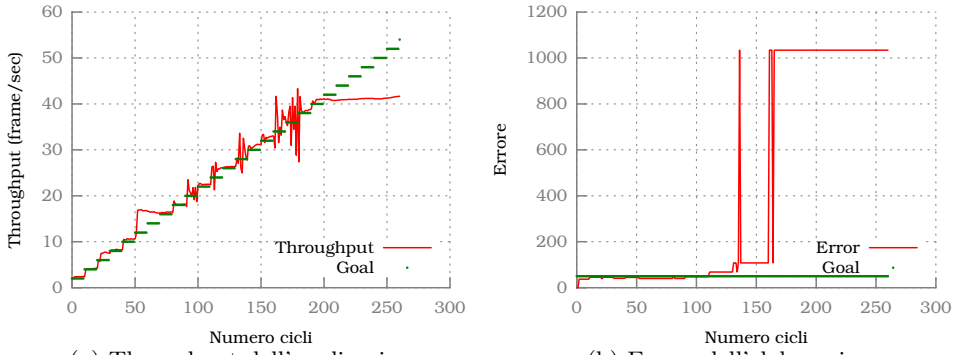
Figura 6.9: Adattabilità in bodytrack

zato. Negli esperimenti si è utilizzata la media dei risultati ottenuti rispetto ai frame elaborati per evindenziare il comportamento che ne consegue.

Questa caratteristica permette al framework di trovare un equilibrio composto da più punti di lavoro, dove si passa da una configurazione che aumenta la qualità, ma più computazionalmente onerosa, ad una più veloce ma che diminuisce la qualità. Nei test effettuati si è stabilito un vincolo sull'fps minimo da raggiungere e un vincolo sull'errore massimo tollerato. La funzione di rank utilizzata è di tipo geometrico e cerca di minimizzare il numero di thread e massimizzare il numero di particelle utilizzate.

#### 6.3.2.1 Adattabilità rispetto alle prestazioni

Per questo esperimento viene avviata un'istanza di bodytrack ogni 25 secondi, senza cambiare le politiche decisionali descritte precedentemente. In Figura 6.9 sono illustrati i risultati ottenuti. Come è possibile osservare l'adattabilità permette all'applicazione di operare sempre i punti di lavoro validi, tranne in un punto dove si è richiesto al sistema il picco di risosrse computazionali. Tale punto coincide con l'esecuzione dell'ultima istanza del-




(a) Throughput dell'applicazione

(b) Errore dell'elaborazione

(c) Overhead del framework

Figura 6.10: Stress test per bodytrack

Nelle

l'applicazione. I risultati di quest'esperimento sono molto più stabili rispetto all'applicazione precedente, il motivo risiede nel fatto che le politiche decisionali definite favoriscono il cambiamento di parametri applicativi puri, evitando di richiedere la creazione di nuovi thread.

#### 6.3.2.2 Stress test

In questa prova si considera una sola istanza dell'applicazione a cui viene aumentato periodicamente il vincolo che esprime il throughput minimo che deve raggiungere, mantenendo invariato il vincolo sull'errore commesso. In questo modo è possibile osservare le azioni intraprese dal framework. Le politiche decisionali non sono cambiate rispetto all'esperimento precedente.

In Figura 6.10 sono mostrati i risultati ottenuti nello stress test da bodytrack. Dai risultati è evidente il fenomeno descritto precedentemente: nell'intorno del centosettantesimo frame è possibile notare come le prestazioni contenute nel punto di lavoro differiscano da quelle osservate dai monitor. Il fenomeno oscillatorio è infatti originato dalla propagazione lineare dell'errore di osservazione, che risulta rispettivamente pessimistico e successivamente

ottimistico. Le politiche decisionali impostante favoriscono il cambiamento dell'unico paramatro applicativo non coinvolto nella funzione di rank o soggetto di vincoli: il numero di livelli da considerare per l'elaborazione. Quest'ultimo parametro è la variabile che incide maggiormente nella qualità del risultato finale, ed è stata lasciata liebera per evindenziare il comportamento del framework in presenza di parametri che rende invalida l'ipotesi alla base della propagazione lineare dell'errore descritta nella metodologia.

Dopo l'elaborazione del duecentesimo frame il punto di lavoro correttamente selezionato dal framework è quello che offre le prestazioni migliori, che coincide la configurazione di parametri che comporta la peggiore qualità di elaborazione.

Una soluzione per evitare tale comportamento è non selezionare tale variabile come parametro applicativo modificabile, ed effettuare un'analisi sul profiling dell'applicazione per individuare un suo valore ottimale in fase di Design-Time. Un'altra soluzione è utilizzare questa caratteristica per trasformare l'equilibrio cercato da un punto ad un anello. Si supponga che l'applicazione sia inserita in un gioco di tiro al bersaglio, dove il sistema deve colpire con delle freccette un manichino in movimento. In questo contesto non è necessario che l'applicazione individui in modo preciso la posa del manichino in ogni istante di tempo, basta farlo solo in prossimità del tiro, mentre nel tempo restante è sufficiente tenere traccia della posizione del manichino. In questo modo è possibile stabilizzarsi in un insieme di più punti di lavoro.

### 6.3.3 x264

L'ultimo esempio applicativo appartiene al mondo dell'elaborazione video, l'applicazione x264 è un encoder H.264/AVC (Advanced Video Coding). È basato sullo standard ITU_T H.264 che descrive la compressione con perdita di uno stream video[49]. Migliora le prestazioni degli altri encoder standard, con nuove funzionalità come la variable block-size motion compensation (VBSMC) o come il context-adaptive binary arithmetic coding (CABAC). Questa applicazione utilizza diverse tecniche per identificare e rimuovere ridondanze nei dati, quella più importante è la motion compensation che sfrutta la ridondanza temporale rispetto a due frame diversi.

Questa applicazione espone moltissimi parametri che influenzano l'elaborazione video, nelle due prove sono stati selezionati due parametri che influenzano la compensazione di movimento e un ultimo parametro che indica quanti reference frame utilizzare per definire i frame futuri. A differenza delle altre due applicazioni si è scelto di non variare il numero di thread creati, ma di utilizzare solamente parametri applicativi. I valori numerici che possono assumere tali parametri sono paragonabili, quindi è possibile raggruppare in un unico grafico i cambiamenti effettuati dal framework. Le metriche che definiscono la prestazione dell'algoritmo sono il tempo medio

di elaborazione di un frame, la dimensione del file prodotto e il valore di PSNR (Peak Signal to Noise Ratio). Quest'ultimo parametro viene espresso in decibel ed esprime il rapporto tra il rumore e l'informazione presente nel frame. Maggiore è il valore del PSNR, maggiore è la "somiglianza" con l'immagine originale, nel senso che si avvicina maggiormente ad essa da un punto di vista percettivo umano.

In entrambi gli esperimenti legati a questa applicazione, viene utilizzata una funzione di rank geometrico che punta a minimizzare la dimensione del file prodotto e a massimizzare il valore di PSNR, imponendo un vincolo sulla velocità di elaborazione di un frame. In questo modo si sono messi in relazione tutti gli aspetti principali dell'elaborazione video.

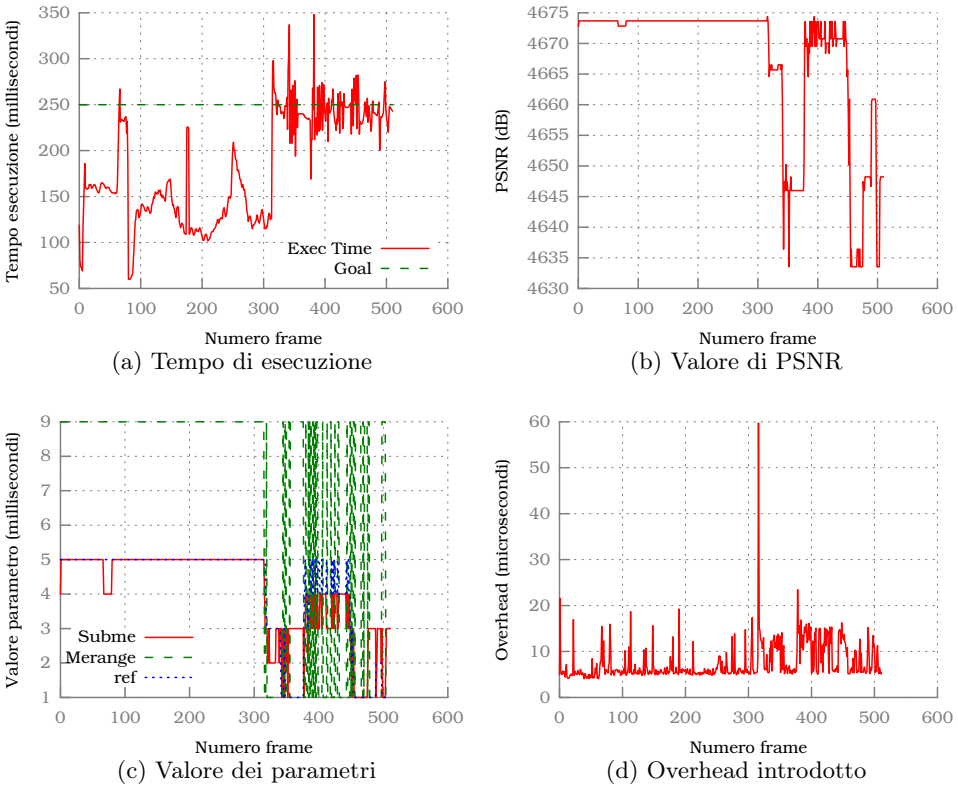
La caratteristica di questa applicazione è che l'elaborazione di un framework può richiedere diverso tempo rispetto a quello successivo. A differenza delle altre applicazioni dov'era era possibile effettuare delle previsioni accurate sulle prestazioni raggiunte, in questa applicazione un frame può impiegare fino al triplo del tempo rispetto ad un altro, a parità di configurazione. Per questo motivo è stato selezionato questo encoder video.

#### 6.3.3.1 Adattabilità rispetto alle prestazioni

In questa prova viene eseguita un'istanza dell'applicazione ogni 5 secondi, senza cambiare le politiche decisionali descritte precedentemente. In Figura 6.11 sono raffigurati i risultati ottenuti. La natura del filmato utilizzato come input permette all'encoder di elaborare velocemente i frame centrali. Per questo motivo è possibile notare come solo verso la fine dell'elaborazione il framework inizia ad effettuare cambiamenti per rispettare il vincolo sul tempo di esecuzione. L'effetto che si genera è dovuto al fatto che codificare un frame non impiega lo stesso tempo per tutti i frame, quindi si genera un meccanismo di oscillazione che porta il framework a continuare a cambiare decisione in base al framework che analizza. Nonostante questo effetto è possibile osservare che il tempo impiegato ad elaborare un fotogramma è vicino al valore vincolo.

#### 6.3.3.2 Stress test

In quest'ultimo esperimento viene modificata la definizione di rank per accentuare l'imprevedibilità dell'elaborazione, in particolare è stata definita una funzione di ranking semplice che massimizza il tempo di elaborazione. In questo modo si viene a creare un conflitto diretto con il vincolo creato, spingendo il framework a lavorare sul limite del vincolo. In Figura 6.12 vengono mostrati i risultati ottenuti. Come nel caso precedente, i frame centrali permettono un'elaborazione veloce, tuttavia è possibile notare come nei frame finali il framework tenta di seguire il valore del vincolo. L'effetto oscillatorio spiegato precedentemente perturba il risultato ottenuto.



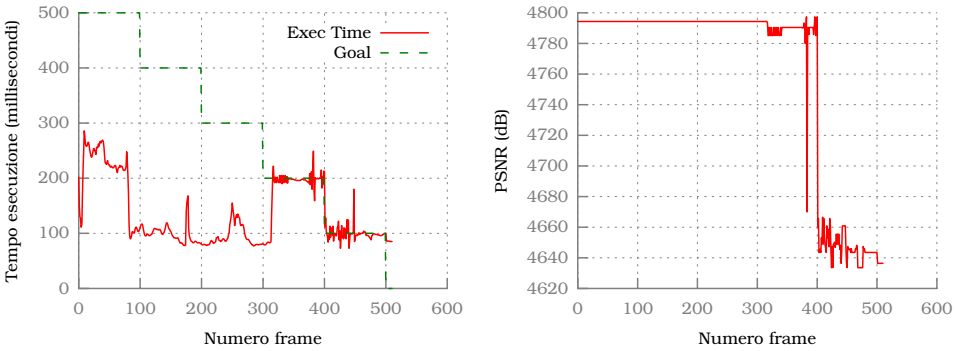

(a) Tempo di esecuzione

(b) Valore di PSNR

(c) Valore dei parametri

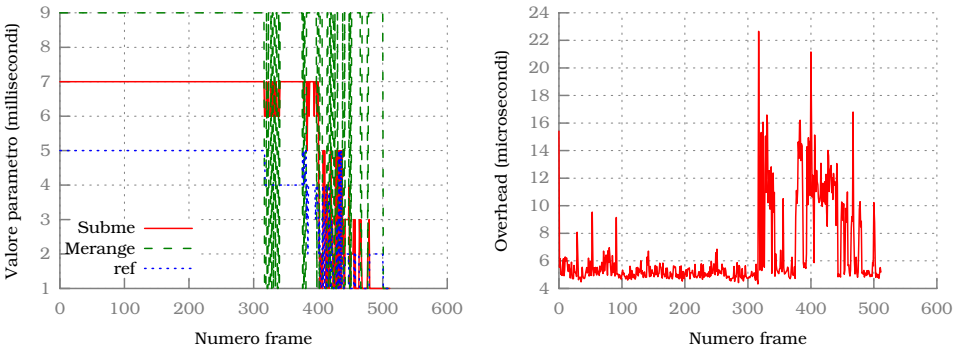
(d) Overhead introdotto

Figura 6.11: Adattabilità in x264




(a) Tempo di esecuzione

(b) Valore di PSNR




(c) Valore dei parametri

(d) Overhead introdotto

Figura 6.12: Stress test per x264

## 6.4 Analisi dei risultati

In questo capitolo sono stati riportati gli esiti delle sperimentazioni effettuate per validare il framework sviluppato. Il benchmark sviluppato ha fornito ottimi risultati riguardo la complessità sperimentale delle maggiori operazioni effettuate, evidenziando un andamento lineare rispetto al numero di Operating Point totali, nel caso peggiore. La piattaforma utilizzata non è propriamente un sistema embedded, quindi il valore numerico dei risultati ottenuti hanno importanza relativa, tuttavia l'overhead introdotto che è stato osservato sia dal benchmark, sia nei casi applicativi, evidenzia come il costo computazione del framework sia praticamente trascurabile rispetto all'elaborazione dell'applicazione.

I casi applicativi hanno fornito l'occasione per mostrare le tre cause per cui possono esserci dei fenomeni oscillatori nelle prestazioni dell'applicazione. Il primo motivo consiste nell'utilizzare diversi programmi che hanno una pessima strategia di attuazione dei cambiamenti decisi dal framework, richiedendo in maniera quasi-periodica ingenti risorse al sistema innescando un fenomeno oscillatorio che influenzerà le altre applicazioni. La seconda causa è la presenza di parametri o metriche che dipendono, oltre che dalla configurazione attuale, anche dallo stato precedente; fornendo la possibilità di sfruttare tale caratteristica, per iniziare ad utilizzare un equilibrio composto da diversi punti di lavoro. L'ultima causa di oscillazione è la scarsa predicibilità del tempo di elaborazione. Alcune applicazioni, come swaptions, permettono di rispettare stime accurate, tuttavia altre sono molto dipendenti dai dati in ingresso.

# Capitolo 7

# Conclusioni e sviluppi futuri

In questo lavoro di tesi sono state affrontate alcune problematiche riguardo alla crescente domanda di self-awareness delle applicazioni. Il contesto che ha fatto da contorno al problema affrontato è il mondo dei sistemi embedded, che impone forti requisiti sulle caratteristiche di esecuzione del programma e parsimonia nell'utilizzo di risorse. Per un'applicazione avere la capacità di riconfigurarsi è un fattore chiave per aumentare la sua efficienza e affrontare con successo l'evoluzione della situazione.

Come descritto nei capitoli precedenti sono state utilizzate svariate tecniche per permettere l'adattabilità in fase di Run-Time che operano a livelli differenti. La metodologia implementata dal framework ARGO si colloca al livello di astrazione più alto, integrandosi completamente nell'applicazione. Tale approccio permette una forte versatilità nel suo impiego andando a soddisfare anche esigenze esterne al mondo dei sistemi embedded. La capacità di far fronte a fluttuazioni della mole di lavoro da svolgere oppure a cambiamenti dell'ambiente di esecuzione permette l'uso del framework anche in sistemi più grandi che offrono servizi più complessi.

La struttura modulare del framework permette una forte personalizzazione nell'integrazione e permette di utilizzare qualsiasi grandezza dell'applicazione per guidare il processo di riconfigurazione. Molti esempi e scenari utilizzati nei capitoli precedenti erano incentrati sulla velocità e qualità di elaborazione, tuttavia il framework non impone nessun vincolo sulla natura di metriche e parametri, permettendone un uso generico, a differenza di molte soluzioni descritte nello stato dell'arte.

Si è passati infatti da un approccio orientato alle prestazioni, intese come velocità di esecuzione, ad un approccio orientato ai Goal permettendo al progettista di utilizzare il framework con il più alto grado di libertà possibile, offrendo la promessa di fare il possibile per portare il funzionamento dell'applicazione verso il comportamento desiderato. Qualora non fosse possibile raggiungere tale risultato variando i parametri applicativi, viene offerta la possibilità di utilizzare ulteriori strumenti come l'uso di vincoli rilassabili.

Quest'ultimo concetto permette infatti al progettista di espandere il controllo sull'esecuzione dell'applicazione ad altri attori esterni.

A differenza di altre soluzioni che permettono di esprimere un solo tipo di comportamento ideale, l'utilizzo del concetto di stato permette allo sviluppatore di poter cambiare in fase di Run-Time gli obbiettivi e preferenze di esecuzione, permettendo di fatto di definire diverse modalità di funzionamento.

## 7.1 Obbiettivi raggiunti

Questo lavoro di tesi ci ha permesso di affrontare il problema della gestione di applicazioni a Run-Time. Il primo obbiettivo è stato rendere ARGO indipendentemente dal framework BOSP, descritto precedentemente. In questo modo si è reso il lavoro completamente indipendente dall'architettura hardware, aumentando il numero di scenari in cui può essere applicato. Il secondo obbiettivo è stato quello di ridisegnare L'Application-Specific Run-Time Manager in modo da permettere allo sviluppatore di poter definire il comportamento ideale dell'applicazione in ogni sua caratteristica, mantenendo un alto livello di generalità e trasparenza nell'integrazione.

Di conseguenza la soluzione ottenuta, si è concretizzata nella metodologia descritta nei capitoli precedenti. L'implementazione ha richiesto alcune modifiche dell'ottima struttura di monitoring sviluppata nel lavoro di tesi precedente, per integrarsi nel nuovo modulo decisionale descritto nel Capitolo 5. La stratificazione dell'architettura permette allo sviluppatore di utilizzare esattamente la frazione del framework che ritiene necessaria, in linea con l'esigenza di ottimizzare le risorse tipica del mondo dei sistemi embedded. In particolare se il codice di elaborazione è molto semplice, l'utilizzo della sola infrastruttura di monitoring può essere sufficiente, in quanto fornisce tutti gli strumenti necessari per verificare il raggiungimento dei Goal dichiarati o le proprietà statistiche richieste. In questo caso è lo sviluppatore a occuparsi dell'adattamento dell'applicazione. Al contrario, se si utilizza l'AS-RTM è il framework a garantire ogni volta che è chiamato a decidere, i parametri che meglio si adattano alla situazione attuale.

Da un punto di vista funzionale, si è raggiunto l'obbiettivo di fornire all'applicazione gli strumenti necessari per raggiungere il più alto livello di adattabilità consentito da una gestione che non prevede il coordinamento fra attori diversi. Utilizzando il framework sviluppato, ogni istanza è in grado di autogestirsi senza la necessità di conoscere altre informazioni sull'ambiente in cui viene eseguita al di fuori di quelle contenute nella lista di punti di lavoro.

Da un punto di vista sperimentale, i test effettuati hanno confermato la validità della soluzione proposta evidenziando il corretto funzionamento di ARGO con la gestione dell'applicazione a Run-Time. La fase decisionale,

eseguita in un tempo trascurabile rispetto al tempo di esecuzione totale, presenta un bassissimo overhead non appesantendo l'esecuzione.

Affrontare questi aspetti ha fornito un'ottima conoscenza delle problematiche che saranno sempre più attuali data l'esigenza di sfruttare adeguatamente le architetture multiprocessori.

## 7.2 Sviluppi futuri

In questa sezione verranno discussi gli sviluppi futuri pensati durante lo svolgimento della tesi, ma non implementati nella soluzione finale. Tali proposte si possono dividere in tre categorie principali.

L'aspetto più importante da migliorare è la gestione dei vincoli in caso non esiste un punto di lavoro valido, perché il significato di vincolo diventa più relativo. Nel caso in cui ci sia almeno un punto di lavoro valido il concetto di vincolo deve essere prioritario rispetto al valore di rank. Nel caso precedente invece non esiste alcun punto di lavoro che rispetta tutti i vincoli, per cui indipendentemente da quale configurazione viene selezionata non è possibile portare il funzionamento dell'applicazione all'interno dell'intervallo desiderato. In questa situazione si potrebbe dare meno peso ai vincoli in favore del punteggio di rank. L'idea alla base di questa affermazione deriva dalla seguente considerazione: "se non posso rispettare i vincoli, posso almeno usare una configurazione che giudico buona".

Nell'implementazione attuale si da precedenza ai vincoli, anche nel caso sopra citato. In particolare si effettua il possibile per avvicinarsi alla zona valida, indipendentemente dal punteggio di rank di tale punto (trascurando il caso in cui due punti soddisfino gli stessi vincoli). Il metodo che si vuole discutere è meno rigido sul concetto di vincolo, ridefinendo in parte il metodo per giudicare se un punto di lavoro sia meglio di un altro, cercando di determinare se la differenza del punteggio di rank giustifichi la scelta di un Operating Point più lontano dalla zona di validità. Una possibile strategia che si può adottare è la seguente. Si supponga di voler stabilire se l'$i$-esimo Operating Point sia migliore del $j$-esimo punto di lavoro. Il primo passo è stabilire la distanza del punto dalla zona di validità. Per ottenere tale scopo si definisce $d_k$ la distanza (in percentuale) del generico punto di lavoro $k$ dalla zona di validità descritta dai vincoli come:

$$d_k = \frac{\sum_{\gamma}^{\Gamma} d_k^{\gamma}}{\|\Gamma\|}$$

dove $\Gamma$ è la lista dei vincoli e $d_k^{\gamma}$ è la distanza dal valore del $\gamma$-esimo vincolo appartenente a $\Gamma$, del $k$-esimo Operating Point. In particolare quest'ultima distanza è definita come:

$$d_k^{\gamma} = \begin{cases} \left|\frac{v_k - v_g}{v_g}\right| & \textit{se } k \textit{ non è ammissibile per } \gamma \\ 0 & \textit{altrimenti} \end{cases}$$

dove $v_k$ è il valore che permettere di raggiungere il punto di lavoro e $v_g$ è il valore del vincolo. Possiamo ora definire il fattore di miglioramento $\mu_k$ come:

$$\mu_k = \frac{r_k}{d_k}$$

dove $r_k$ è il punteggio di rank del $k$-esimo Operating Point. Tale fattore mette in relazione la bontà del punto di lavoro con la distanza dalla zona di validità. In questo modo la discriminazione degli Operating Point non avviene più rispetto al numero di vincoli contigui rispettati, ma si tiene in considerazione anche il punteggio di rank. Utilizzando tale strategia non si considera più il vincolo come una barriera che divide i punti di lavoro in ammissibili o meno, ma assume un significato più sfumato. In questo modo si permette ad un Operating Point che non soddisfa nessun vincolo singolarmente, ma che è molto vicino alla zona di validità e ha alto punteggio di rank, di essere selezionato come configurazione migliore.

Il secondo aspetto da migliorare è la trasparenza dell'integrazione nell'applicazione. Attualmente l'uso del framework è stato reso il meno intrusivo possibile, tuttavia la fase di configurazione risulta ancora un po' opaca. Si potrebbe progettare un sistema di inizializzazione automatica, basato su un file di configurazione. Il problema principale di questo approccio consiste nel trovare la strategia per esporre i monitor all'applicazione, effettuando meno assunzioni possibili.

L'ultimo aspetto che si può migliorare è l'ottimizzazione del codice e delle strutture dati in generale. In particolare se l'applicazione/rimozione di un vincolo può essere un'operazione dispendiosa, la ricerca di un nuovo punto migliore valido deve essere molto reattiva. Attualmente la complessità è lineare rispetto agli Operating Point contenuti nel modello. Per migliorare la situazione si potrebbe pensare ad una soluzione più efficiente nell'organizzare i punti di lavoro validi.

# Appendice A

# Esempio applicativo

Si supponga di avere un'applicazione di editing video che venga eseguita su un dispositivo embedded alimentato da batterie. L'algoritmo di tale applicazione è parametrico, in particolare si basa su due parametri: il numero di volte in cui applicata un'elaborazione sul frame (chiamato *numero_iterazioni*) e il numero di frame elaborati simultaneamente (chiamato *numero_thread*). Per chiarezza viene riportata la struttra generica del codice che compone l'applicazione originale:

```
1  ...
2
3  void fai_il_lavoro(int numero_iterazioni, int numero_thread) {
4    // codice dell'applicazione
5  }
6
7  ...
8
9  int main() {
10
11   // ciclo principale
12   while(1) {
13     fai_il_lavoro(1, 3);
14   }
15
16   return 0;
17 }
```

Prima di utilizzare il framwork occorre decidere quali grandezze siano significative per caratterizzare i parametri utilizzati. Per questa applicazione si supponga di essere interessati a tre metriche: il numero di frame elaborati in un secondo (chiamata *fps*), la potenza dissipata dal dispositivo (chiamata *potenza*) e la qualità dell'elaborazione (chiamata *qualità*).

Occorre inoltre decidere i criteri secondo i quali un punto di lavoro sia valido, la definizione di bontà di un punto di lavoro e quali metriche osservare

durante l'evoluzione dell'esecuzione. Per questa applicazione si assuma che un punto di lavoro è valido se:

- riesce ad elaborare almeno 25 frame al secondo.
- consumi non più di 20 Watt.

Si supponga inoltre che la definizione di bontà di un operating point sia indicata dalla qualità dell'elelaborazione.

Mentre la qualità e la potenza dissipata sono metriche difficilmente misurabili in fase di esecuzione, tenere traccia del numero di frame elaborati in un secondo può decisamente permettere all'applicazione di adattarsi; per queste ragioni si supponga di voler creare un monitor che osservi tale grandezza.

## Inizializzazione

Di seguito verranno elencate le operazioni necessarie per utilizzare correttamente il framework nell'applicazione appena presentata.

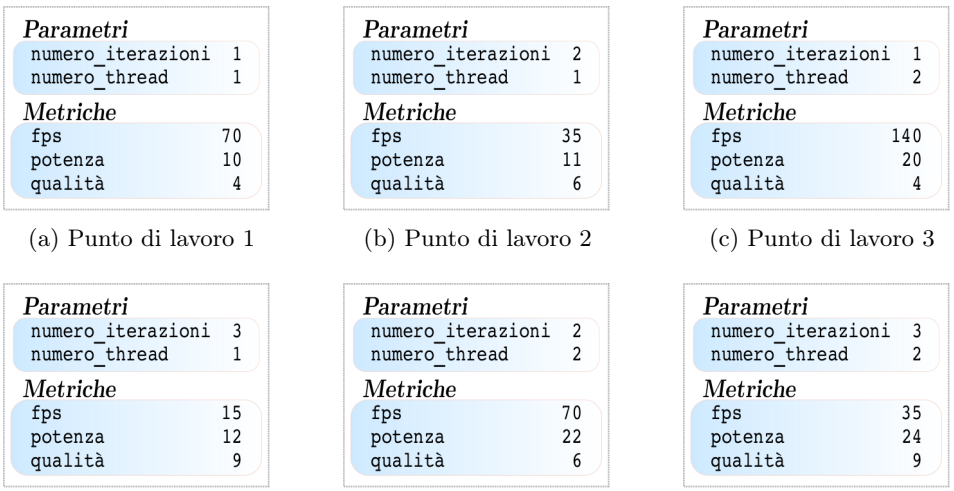
### Definizione della lista di Operating Point

La prima operazione da effettuare prima dell'integrazione del framework consiste nell' ottenere i valori che assumono tutte le metriche per ogni punto di lavoro; tale compito viene tipicamente effettuato a Design Time utilizzando un simulatore dell'architettura e svolgendo un'attività di profiling. Ipotizzando di aver effettuato tali operazioni, si supponga di aver ottenuto un curva di Pareto formata dai punti di lavoro rappresentati in Figura A.1.

La metrica qualità si supponga essere espressa in una scala da uno a dieci, dove il punteggio migliore ha valore dieci; mentre la metrica potenza indica la quanità di energia utilizzata dal dispositivo. Supponendo che esista una funzione chiamata *get_op_list()* che restituisce la lista degli operating point, il codice dell'applicazione diventa:

```
1  ...
2
3  void fai_il_lavoro(int numero_iterazioni, int numero_thread) {
4    // codice dell'applicazione
5  }
6
7  ...
8
9  int main() {
10
11   argo::asrtm::OperatingPointsList OPList = get_op_list();
12
13   // ciclo principale
```

```
14    while(1) {
15      fai_il_lavoro(1, 3);
16    }
17
18    return 0;
19  }
```




(a) Punto di lavoro 1 (b) Punto di lavoro 2 (c) Punto di lavoro 3

(d) Punto di lavoro 4 (e) Punto di lavoro 5 (f) Punto di lavoro 6

Figura A.1: Punti di lavoro ottenuti

### Inizializzazione dell'Operating Point Manager

Una volta ottenuta la lista dei punti di lavoro, si può direttamente creare l'Operating Point manager. In questo modo i parametri dell'applicazione non vengono più gestiti staticamente, ma è compito dell'OPManager fornire all'applicazione i parametri da utilizzare. Il nuovo codice dell'applicazione diventa quindi:

```
1  ...
2
3  void fai_il_lavoro(int numero_iterazioni, int numero_thread) {
4    // codice dell'applicazione
5  }
6
7  ...
8
9  int main() {
10
11   argo::asrtm::OperatingPointsList OPList = get_op_list();
12
13   argo::asrtm::OPManager manager(OPList);
```

```
14
15    argo::asrtm::ApplicationParameters parametri;
16
17    // ciclo principale
18    while(1) {
19      // ottieni i parametri
20      parametri = opManager.getBestApplicationParameters();
21
22      fai_il_lavoro(parametri["numero_iterazioni"],
23                    parametri["numero_thread"]);
24    }
25
26    return 0;
27  }
```

Il manager appena creato non sa qual'è la definizione di migliore e nemmeno quando un punto di lavoro è valido, quindi l'unica decisione che può prendere è quella di restituire sempre il primo punto di lavoro.

Quando si crea l'oggetto OPManager esso inizializza la situazione iniziale; in particolare viene creato il modello, si salva l'array dei parametri e inizializza le liste dei vincoli, come è possibile osservare dalla Figura A.2.

Prima di iniziare a spiegare in dettaglio il significato di ogni struttura al suo interno, occorre precisare che l'oggetto OPManager identifica i punti di lavoro in base alla loro posizione nella lista dei punti di lavoro partendo a contare da zero; quindi l'operating point 1 è rappresentato in Figura A.1b.

Come prima operazione il costruttore si salva la lista dei paramtri per ogni punto e siccome non ha a disposizione nessun'altra informazione assume che ogni punto di lavoro è valido e non ci sono preferenze tra un punto rispetto agli altri; per queste ragioni i parametri migliori sono quelli appartenenti al punto di lavoro 0, nella figura l'array dei parametri e il puntatore ai parametri correnti sono indicati in verde.

Come seconda operazione si crea il modello della situazione. Il modello è formato da due liste di identificatori di punti di lavoro; la prima lista "S1" contiene tutti i punti validi che rispettano tutti i vincoli (quindi per ora contiene tutti gli elementi della lista), mentre la seconda lista "S2" contiene tutti gli identificatori dei punti di lavoro eventualmente aggiunti durante l'esecuzione del programma, per il momento quindi è vuota. Nella figura sono indicati in giallo.

Come ultima operazione inizializza l'array del rank per ogni punto di lavoro. Il rank di un punto di lavoro identifica quanto buono sia un punto di lavoro, quindi siccome per ora non abbiamo alcuna informazione a riguardo viene inizializzato a zero. Nella figura viene indicato in grigio.

Nell'oggetto OPManager sono presenti due ulteriori liste, quella dei vincoli e qualla dei vincoli rilassabili. La lista dei vincoli contiene tutte le proprietà che le metriche di un punto di lavoro deve possedere per essere valido, quindi essa influenzerà la decisione riguardo la scelta dei parametri migliori.

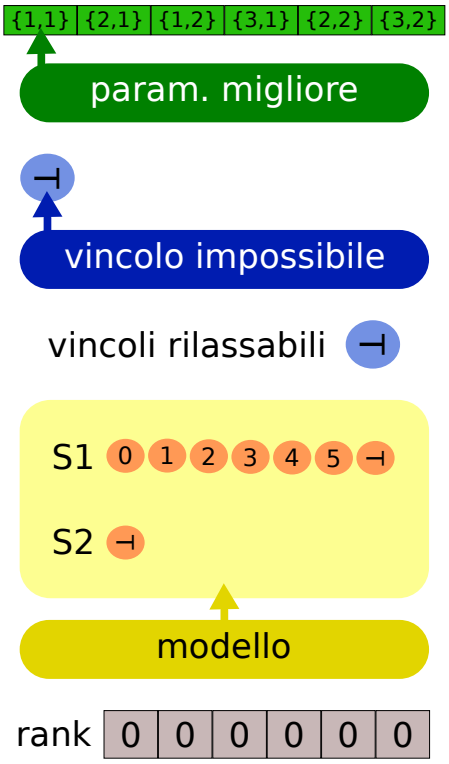



Figura A.2: Situazione iniziale dell'OPManager

La lista dei vincoli rilassabili invece è la lista di vincoli che non influenzano la scelta dei parametri, ma in caso non vengano rispettati chiameranno una funzione di callback che lo sviluppatore dell'applicazione ha deciso. Siccome per il momento non sono presenti informazioni riguardo entrambe le liste, esse sono vuote. Nella figura vengono indicate in blu.

### Applicazione del monitor

Il prossimo passo consiste nell'inserire il monitor che osserva quanti frame vengono elaborati in un secondo. Questa aggiunta per il momento non interagisce con l'oggetto OPManager, ma definisce solamente un monitor con il goal che avevamo deciso, ovvero almeno 25 frame elaborati al secondo. Il codice dell'applicazione risulta quindi essere:

```
1  ...
2
3  void fai_il_lavoro(int numero_iterazioni, int numero_thread) {
4    // codice dell'applicazione
5    ...
6    monitor.start(win_id);
7    // codice che elabora un frame
```

```
8     monitor.stop(win_id, 1); // uno perchè elabora un singolo frame
9     ...
10  }
11
12  ...
13
14  // dichiarazione del monitor
15  argo::asrtm::ThroughputMonitor monitor;
16  uint16_t win_id;
17  uint32_t goal_id;
18
19  int main() {
20
21    // dichiarazione dell'op manager
22    argo::asrtm::OperatingPointsList OPList = get_op_list();
23    argo::asrtm::OPManager manager(OPList);
24    argo::asrtm::ApplicationParameters parametri;
25
26    // dichiarazione del goal
27    win_id = monitor.newWindow("fps");
28    goal_id = monitor.newGoal(win_id,
29                              argo::DataFunction::Average,
30                              argo::ComparisonFunction::GreaterOrEqual,
31                              25);
32
33    // ciclo principale
34    while(1) {
35      // ottieni i parametri
36      parametri = opManager.getBestApplicationParameters();
37
38      fai_il_lavoro(parametri["numero_iterazioni"],
39                    parametri["numero_thread"]);
40    }
41
42    return 0;
43  }
```

Il framework permette l'uso di macro che rende il processo di integrazione più trasparente. Siccome lo scopo di questa appendice è quello di mostrare l'integrazione del framework nell'applicazione e il suo comportamento dinamico, sono state utilizzate le istruzioni normali. Come è possibile notare dal codice, quando è stata creata la finestra non è stata introdotta nessuna informazione riguardo la sua dimensione. In questo caso viene utilizzata la dimensione di default, ovvero di dieci elementi. Nel caso si dovesse specificare la dimensione, occore passare al metodo *newWindow* oltre al nome della finestra, anche la sua dimensione.

### Definizione di "migliore"

Il prossimo passo consiste nel fornire all'applicazione la definizione di "migliore". Questo è possibile definendo una funzione di rank che associa ad ogni punto di lavoro un valore che indica la sua bontà. Come descritto nel Capitolo 5 esistono tre funzioni di ranking da massimizzare o minimizzare.

La definizione della preferenza non è relegata ad essere utilizzata nell'inizializzazione del framework, può infatti essere cambiata durante l'esecuzione dell'applicazione. In questo esempio siamo quindi interessati a una funzione di rank semplice basata sulla metrica qualità da massimizzare, quindi il codice diventa:

```
1  ...
2
3  void fai_il_lavoro(int numero_iterazioni, int numero_thread) {
4    // codice dell'applicazione
5    ...
6    monitor.start(win_id);
7    // codice che elabora un frame
8    monitor.stop(win_id, 1); // uno perchè elabora un singolo frame
9    ...
10 }
11
12 ...
13
14 // dichiarazione del monitor
15 argo::asrtm::ThroughputMonitor monitor;
16 uint16_t win_id;
17 uint32_t goal_id;
18
19 int main() {
20
21   // dichiarazione dell'op manager
22   argo::asrtm::OperatingPointsList OPList = get_op_list();
23   argo::asrtm::OPManager manager(OPList);
24   argo::asrtm::ApplicationParameters parametri;
25
26   // dichiarazione del goal
27   win_id = monitor.newWindow("fps");
28   goal_id = monitor.newGoal(win_id,
29                             argo::DataFunction::Average,
30                             argo::ComparisonFunction::GreaterOrEqual,
31                             25);
32
33   // dichiarazione del rank
34   manager.setSimpleRank("qualità",
35                         argo::RankObjective::Maximize);
36
37   // ciclo principale
```

```
38    while(1) {
39      // ottieni i parametri
40      parametri = opManager.getBestApplicationParameters();
41
42      fai_il_lavoro(parametri["numero_iterazioni"],
43                    parametri["numero_thread"]);
44    }
45
46    return 0;
47  }
```

Quando viene aggiunto il rank al manager degli operating point, la scelta dei parametri migliori cambia, in quanto ora il manger degli operating point ha le informazioni riguardo a come lo sviluppatore giudica "buono" un punto di lavoro. Tali cambiamenti vengono rappresentati in Figura A.3.

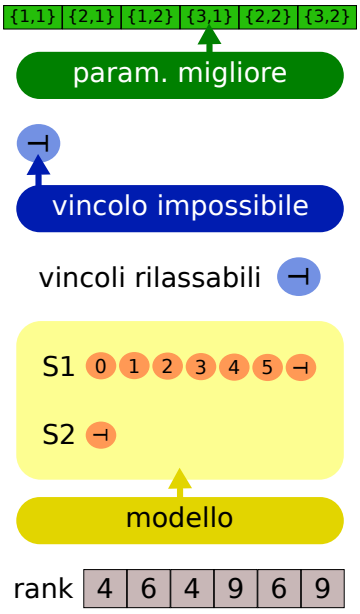



Figura A.3: Nuova struttura dell'OPManager

### Aggiunta dei vincoli

Per completare l'inizializzazione della libreria rimane solamente da aggiungere i vincoli creati all'OPManager. Il codice dell'applicazione risulta quindi essere:

```
1  ...
2
3  void fai_il_lavoro(int numero_iterazioni, int numero_thread) {
4    // codice dell'applicazione
5    ...
```

```
6     monitor.start(win_id);
7     // codice che elabora un frame
8     monitor.stop(win_id, 1); // uno perchè elabora un singolo frame
9     ...
10  }
11
12  ...
13
14  // dichiarazione del monitor
15  argo::asrtm::ThroughputMonitor monitor;
16  uint16_t win_id;
17  uint32_t goal_id;
18
19  int main() {
20
21    // dichiarazione dell'op manager
22    argo::asrtm::OperatingPointsList OPList = get_op_list();
23    argo::asrtm::OPManager manager(OPList);
24    argo::asrtm::ApplicationParameters parametri;
25
26    // dichiarazione del goal
27    win_id = monitor.newWindow("fps");
28    goal_id = monitor.newGoal(win_id,
29                              argo::DataFunction::Average,
30                              argo::ComparisonFunction::GreaterOrEqual,
31                              25);
32
33    // dichiarazione del rank
34    manager.setSimpleRank("qualità",
35                          argo::RankObjective::Maximize);
36
37    // dichiarazione del vincolo sulla potenza
38    manager.addStaticConstraintOnBottom("potenza",
39                                        argo::ComparisonFunction::LessOrEqual,
40                                        20);
41
42
43    // dichiarazione del vincolo sulla velocità
44    manager.addDynamicConstraintOnTop(monitor.getWindow(win_id), goal_id);
45
46    // ciclo principale
47    while(1) {
48      // ottieni i parametri
49      parametri = opManager.getBestApplicationParameters();
50
51      fai_il_lavoro(parametri["numero_iterazioni"],
52                    parametri["numero_thread"]);
53    }
54    return 0;
```

```
55 }
```

In seguito all'aggiunta dei vincoli, l'OPManager crea la carattersitica nota e quella osservabile per rappresentare, risppetivamente il vincolo sulla velocità e quello sulla potenza. Durante la creazione del vincolo dinamico non è stato stato specificato il numero di dati che deve possedere la finestra per essere considerato valido, per cui viene utilizzato il valore di default, ovvero uno. I componenti più significativi che compongono tale oggetto sono raffigurati in Figura A.4.

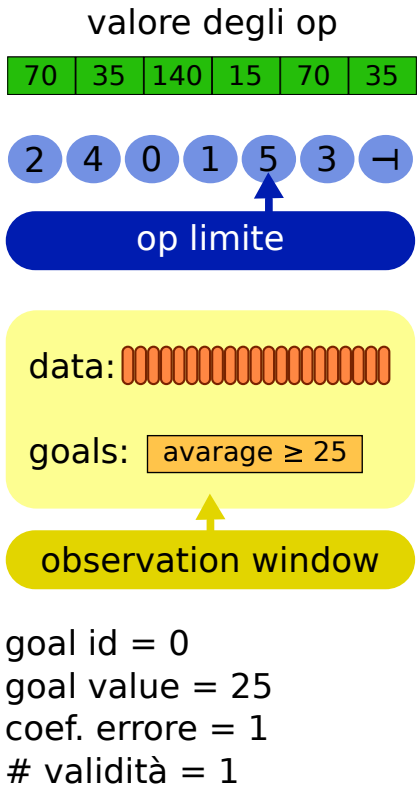



Figura A.4: Caratteristica osservabile della metrica "*fps*"

Nel momento in cui viene creata la caratteristica osservabile, oltre a fornirgli la lista di punti di lavoro, occorre associarli la finestra di osservazione del monitor e l'id del goal che deve rappresentare. In automatico cerca di associare il nome della finestra al nome della metrica presente nella lista di punti di lavoro, ma se tale assunzione non è valida, occorre specificare al metodo anche il nome della metrica nei punti di lavoro su cui si basa tale goal. L'insieme dei parametri da utilizzare con il metodo di inserzione è definito da:

1. Il puntatore alla finestra di osservazione che contiene il goal desiderato
2. L'identificatore di tale goal

3. [OPZIONALE] Il nome della metrica presente nella lista di Operating Point che fa riferimento alla proprietà statistica utilizzata nel goal.
4. [OPZIONALE] Il numero di validità, ovvero il numero minimo di dati che deve contenere la finestra di osservazione.

Ci sono svariate ragioni per cui il nome della metrica nella lista di OP sia differente rispetto a quello della finetra. Il più importante è rappresentato nel caso in cui si voglia monitorare più di una proprietà statistica di una grandezza, si supponga media e varianza. In questo caso occorre utilizzare due metriche distinte per le due misurazioni, mentre la finestra rimane una sola. Di conseguenza è necessario che almeno una metrica abbia il nome differente da quello della finetra.

Il primo passo nella costruzione dell'oggetto ObservableCharacteristic è quello di costruirsi un array che contiene il valore della metrica che ha ottenuto ciascun punto di lavoro, nell'immagine è colorato in verde.

Il secondo passo è quello di costruirsi una lista di operating point e in seguito ordinarla in modo che a valori di indici bassi corrispondano punti buoni e viceversa per indici alti; quindi se il tipo di comparazione è di "maggiore" o "maggiore o uguale" la lista viene ordinata in modo decrescente (rispetto ai valori della metrica); nella figura è colorata in blu.

Il terzo passo è quello di individuare il punto di lavoro limite. Tale punto viene definito come l'operating point valido con indice maggiore, in altre parole è il punto di lavoro che separa quelli che rispettano il vincolo da quelli che non lo rispettano.

Per l'oggetto ObservableCharacteristic, i parametri della finestra di osservazione che sono significativi sono il buffer circolare contenente i dati e il goal all'interno dell'array dei goal all'interno della finestra di osservazione. Il parametro "*coefficiente d'errore*" viene utilizzato per tenere traccia del discostamento del valore osservato dal monitor rispetto a quello calcolato in simulazione. In questo modo tutti i valori degli operating point vengono scalati in base alla nuova informazione portata dal monitor senza dover effettivamente modificare tutti i valori. Il parametro "*goal value*" serve a tenere traccia dell'ultimo valore noto del goal nel caso venga modificato quello nel monitor. Il parametro "*# validità*" serve ad indicare quanti elementi devono essere presenti nel buffer per essere considerato valido.

Nel caso di questa applicazione l'oggetto che si viene a creare viene rappresentato in Figura A.4, si può notare come nella finestra compaia solo il goal specificato e che i punti di lavoro vengono ordinati in modo decrescente.

Le caratteristiche note sono la controparte delle caratteristiche osservabili nel caso in cui non venga utilizzato un monitor. La struttura dell'oggetto è quindi molto simile a quella descritta precedentemente, a meno della gestione del goal che avviene internamente, al posto di delegarla al monitor. Il parametro differenza viene infatti utilizzato per tenere traccia dei cambiamenti del valore del goal. Nel caso di questa applicazione la loro struttura

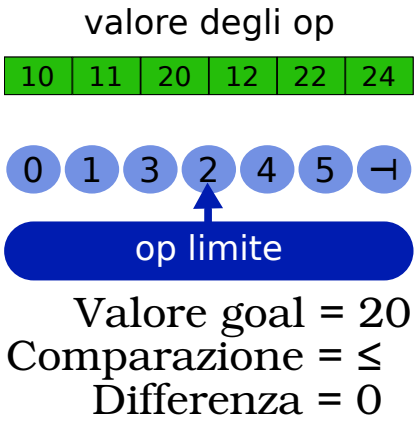



Figura A.5: Caratteristiche nota della metrica “potenza”

principale è raffigurata in Figura A.5.

Quando viene aggiunto il vincolo sull’fps il punto di lavoro 3 viene eliminato da S1 in quanto non valido; a questo punto viene cercato un nuovo operating point tra quelli rimanenti e viene selezionato il punto di lavoro 5.

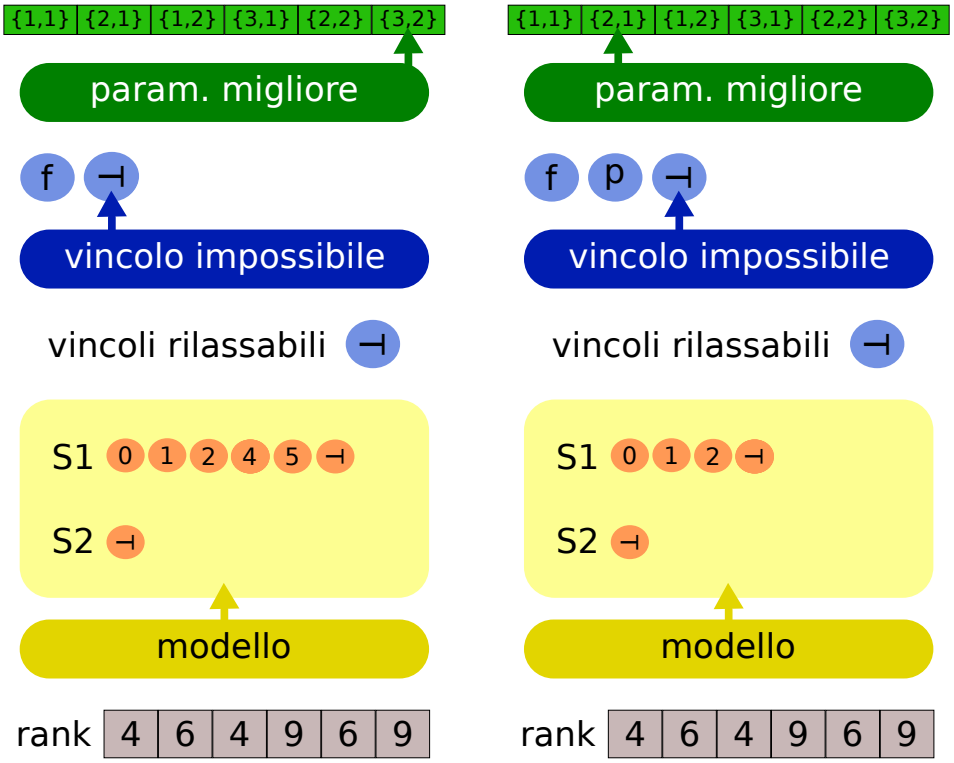
Quando viene aggiunto anche il vincolo sulla potenza, vengono eliminati anche gli operating point 4 e 5; quindi viene ancora una volta cercato un punto di lavoro in S1, selezionado l’operating point 1. I seguenti cambiamenti vengo evidenziati in Figura A.6.

## Comportamento dinamico

Dopo aver definito la lista dei vincoli che delineano i requisiti che un punto di lavoro deve avere per essere considerato valido e aver definito i criteri che ne stabiliscono la bontà, il manager degli operating point selezionerà i migliori parametri per l’applicazione. Tali parametri non devono cambiare ad eccezione che si verifichi almeno una di queste situazioni:

- Viene definita una nuova funzione di rank.
- Vengono aggiunti/rimossi dei vincoli.
- Cambiano i valori dei vincoli.
- Il monitor osserva un valore differente rispetto a quello ottenuto a simulazione.

Le situazioni di cui non si è ancora parlato sono le ultime due, in quanto le prime si sono già verificate nell’inizializzazione del framework quando si è aggiunto il rank e i vincoli decisi in fase di analisi; per questo motivo di seguito verranno riportati due casi che generano le ultime due situazioni.




(a) Aggiunta del vincolo su "fps" (b) Aggiunta del vincolo su "potenza"

Figura A.6: Aggiunta dei vincoli

### Differenza di osservazione

Si supponga che il simulatore sia ideale e i valori di fps individuati siano perfettamente aderenti con quelli osservati dal monitor. Ad un certo punto viene eseguita un'ulteriore istanza dell'applicazione, quindi i valori osservati dal monitor inizieranno a calare (della metà se non intervenisse l'OPManager). In un istante di tempo successivo la seconda istanza dell'aplicazione viene chiusa e quindi i valori di fps osservati dal monitor inizieranno a salire fino ai loro valori nominali.

Iniziamo a descrivere cosa succede quando viene avviata la nuova istanza dell'applicazione. Il cambiamento di fps è istantaneo, ma siccome la finestra di osservazione contiene altri valori, il cambiamento osservato dal manager è graduale (opera sulla media); quindi fintanto che tale valore non scende sotto i 25 fps la caratteristica osservabile ( a parte aggiornare il coefficiente d'errore) non cambia il modello.

Quando il valore di fps scende sotto la soglia del vincolo, la caratteristica osservabile rimuove l'op 1, il manager di conseguenza seleziona l'op 0 (siccome è indifferente, per costruzione mantiene quello scelto inizialmente)

e succesivamente svuota la finestra di osservazione. Nelle figure sottostanti vengono visualizzati 3 istanti di tempo che indicano l'evelversi della situazione; in particolare per t=0 la media dei valori vale 30, per t=1 vale 25 e per t=2 vale 20.

Nella Figura A.7 è presente l'evoluzione della caratteristica osservabile nel tempo, mentre nella Figura A.8 è presente l'evoluzione del manager dei punti di lavoro.

La lista di parametri ora selezionata non cambia fino a quando la seconda istanza dell'applicazione viene terminata e il valore osservato dal monitor ritorna a salire. Quando il valore osservato supera lo soglia dei 25 fps, la caratteristica aggiunge a S2 il punto di lavoro 1, al che l'OPManager verifica che sia valido anche per i restanti vicoli, lo aggiunge ad S1 e siccome il punto appena aggiunto è migliore dell'attuale, lo utilizza per ricavare i migliori parametri per l'applicazione.

In Figura A.9 è presente l'evoluzione della situazione quando il monitor rileva che la metrica fps raggiunge i 25 fps. Dopo tale cambiamento l'unico attributo che varia è il coefficiente d'errore che si assesterà di nuovo a 1.

### Modello impossibile

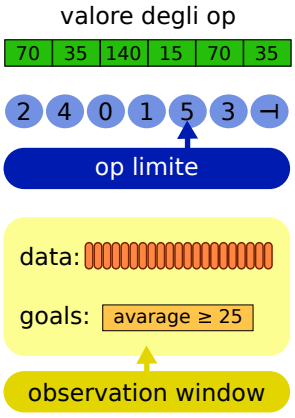
In questo caso si supponga che durante l'esecuzione dell'applicazione si sia modificato il valore del vincolo sulla potenza, in particolare un punto di lavoro è valido se il suo consumo di potenza è minore di 9 watt. Quando il manager degli Operating Point chiede alla caratteristica nota di aggiornare il modello, essa toglierà da S1 tutti i punti di lavoro e sposterà il puntatore dell'OP limite al primo elemento della lista (anche se tale punto di lavoro non è valido).

Dopo che tutti i vincoli hanno aggiornato il modello, l'OP manager dichiarerà inizialmente il vincolo sulla potenza *vincolo impossibile*; siccome nessun punto di lavoro può rispettare tale vincolo, l'OP manager indicherà il vincolo sull'fps come nuovo *vincolo impossibile*. A questo punto il manager sceglierà il migliore punto di lavoro fra quelli che rispettano il solo vincolo sull'fps, essendo il vincolo di più alta priorità. Di conseguenza verranno scelti i parametri appartenenti all'Operating Point 0, in quanto esso rispetta il vincolo sugli fps e ha il minor consumo di potenza.

In Figura A.10 è presente l'evoluzione del vincolo sulla potenza prima e dopo l'aggiornamento del modello; in Figura A.11 è presente l'evoluzione del manager degli Operating Point dopo che ha aggiornato il modello e dopo che ha terminato la funzione getBestApplicationParameters().

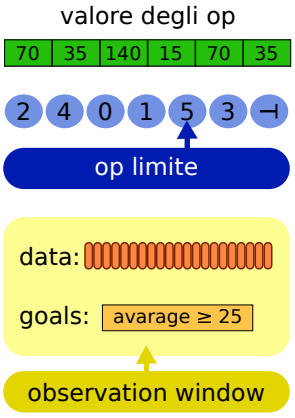
### Modello possibile

In quest'ultimo caso viene trattato la situazione opposta rispetto all'esempio precedente, ovvero il modello ritorna a contenere almeno un punto di



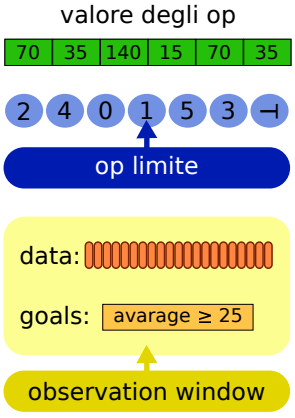

goal id = 0
goal value = 25
coef. errore = 0.8571
# validità = 1

(a) t=0



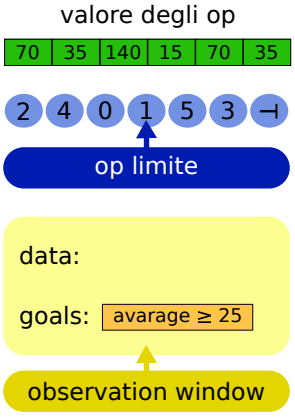

goal id = 0
goal value = 25
coef. errore = 0.7142
# validità = 1

(b) t=1




goal id = 0
goal value = 25
coef. errore = 0.5714
# validità = 1

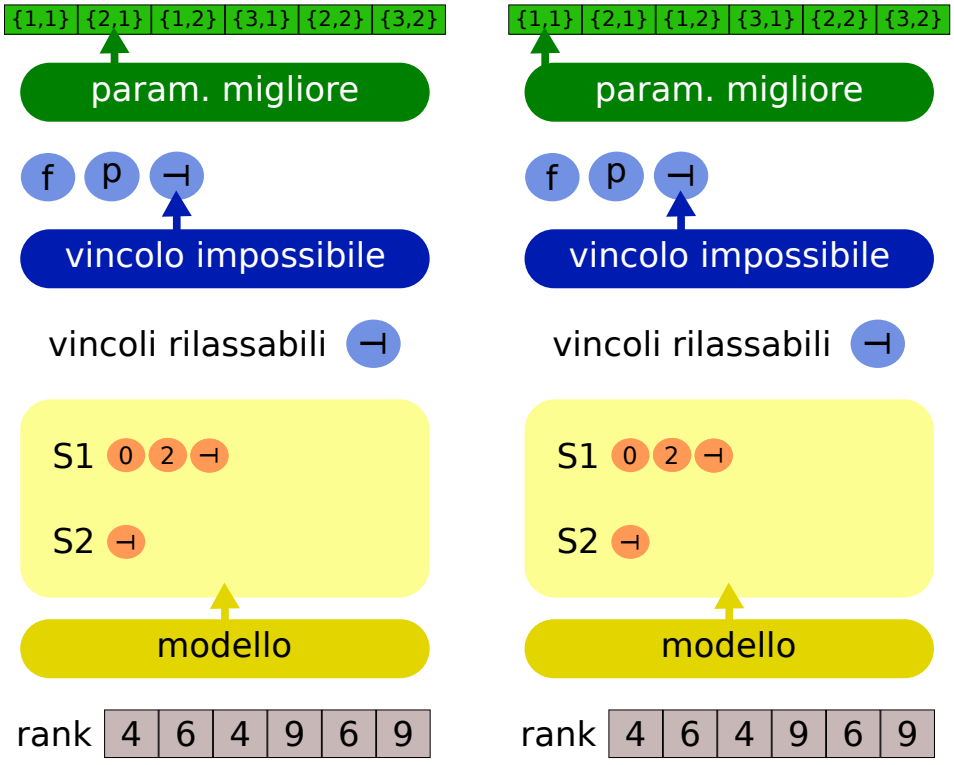
(c) t=2 dopo la chiamata updateModel(...)




goal id = 0
goal value = 25
coef. errore = 0.5714
# validità = 1

(d) t=2 dopo la chiamata getBestApplicationParameters()

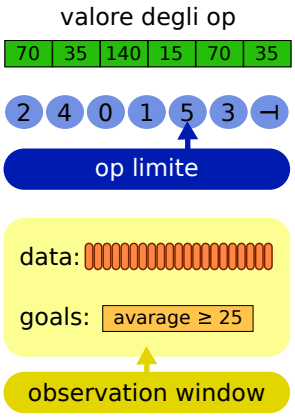
Figura A.7: Evoluzione della caratteristica osservabile




(a) t=2 dopo la chiamata updateModel(...)

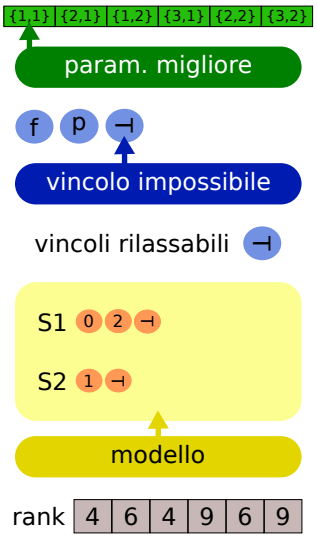
(b) t=2 dopo la chiamata getBestApplicationParameters()

Figura A.8: Evoluzione del manager degli Operating Point

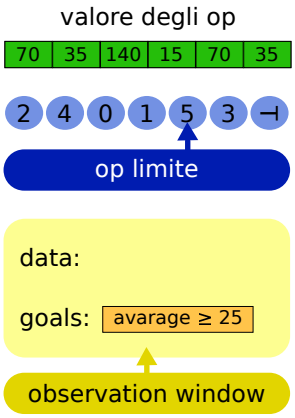



goal id = 0
goal value = 25
coef. errore = 0.71428
# validità = 1

(a) Caratteristica Osservabile dopo la chiamata updateModel(...)

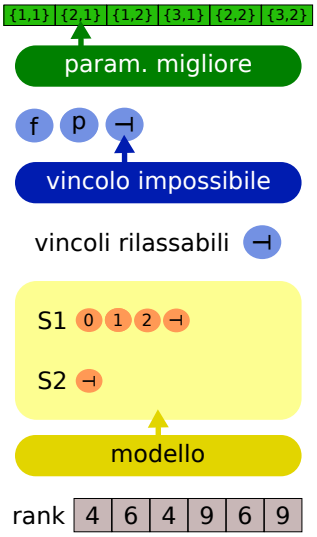



(b) OPManager dopo la chiamata updateModel(...)



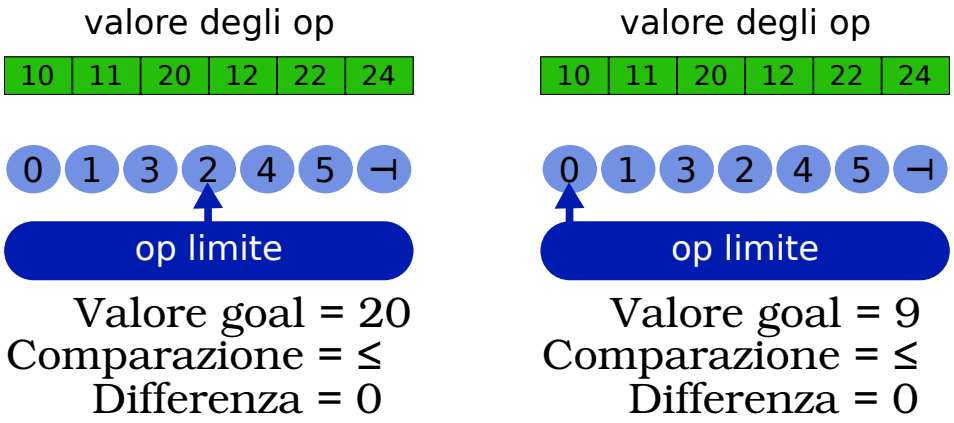

goal id = 0
goal value = 25
coef. errore = 0.71428
# validità = 1

(c) Caratteristica osservabile dopo la chiamata getBestApplicationParameters()



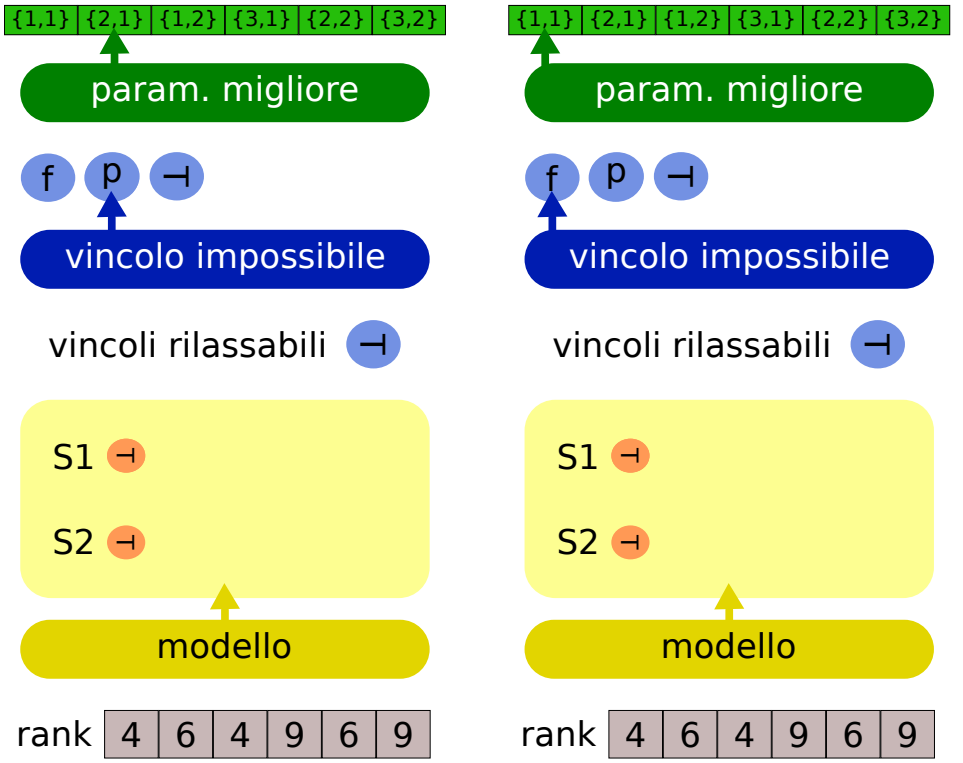

(d) OPManager dopo la chiamata getBestApplicationParameters()

Figura A.9: Evoluzione degli oggetti



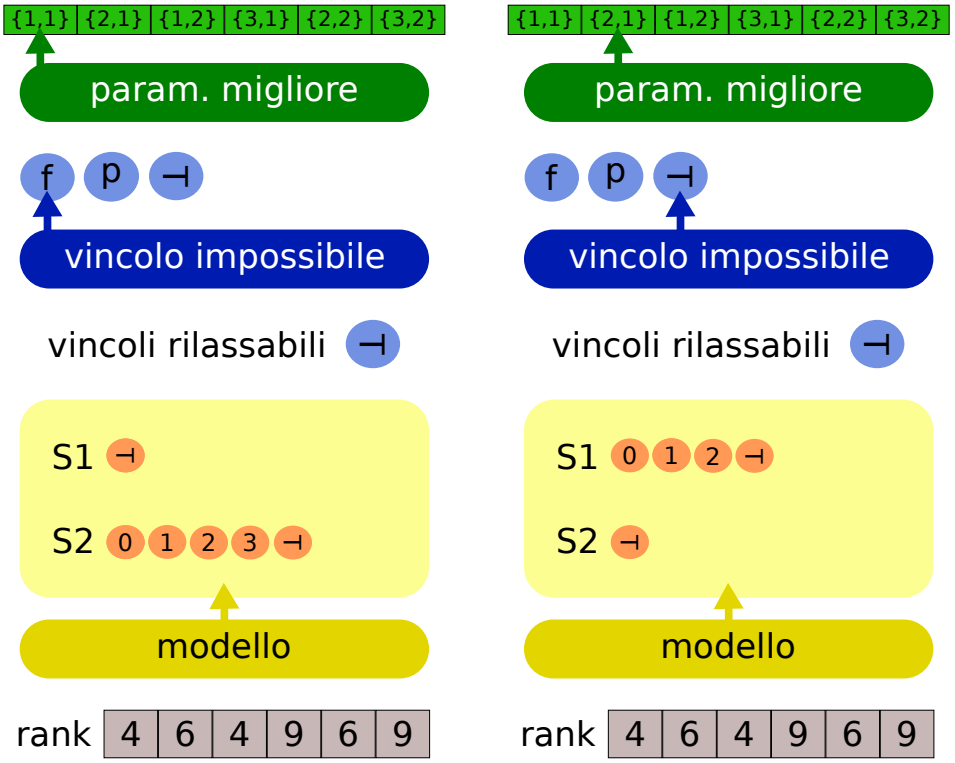

(a) prima dell'update (b) dopo l'update

Figura A.10: Evoluzione del vincolo sulla potenza




(a) dopo aver aggiornato il modello (b) dopo aver eseguito la funzione getBestApplicationParameters()

Figura A.11: Evoluzione dell'OP manager




(a) op manager dopo aver aggiornato il modello

(b) dopo aver eseguito la funzione getBestApplicationParameters()

Figura A.12: Evoluzione dell'OP manager

lavoro valido. Per ottenere questo è sufficiente, partendo dalla situazione precedente, fissare di nuovo il valore del vincolo sulla potenza pari a 20.

In questo caso la caratteristica nota sulla potenza aggiunge ad S2 i punti di lavoro 0, 1, 2 e 3. Quando il manager degli op finisce di interagire con i vincoli, analizza gli Operating Point contenuti in S2 e si accorge che gli op 0, 1 e 2 sono totalmente validi, quindi li aggiunge a S1, seleziona l'Operating Point 1 come migliore e infine posiziona il puntatore del vincolo impossibile alla fine della lista dei vincoli. In Figura A.12 è rappresentata tale evoluzione.

# Bibliography


[1] Bosp: Barbeque open source project. http://bosp.dei.polimi.it.

[2] Google c++ testing framework. http://code.google.com/p/googletest.

[3] Netbsd operating system. http://www.netbsd.org.

[4] A.R. Alameldeen and D.A. Wood. Ipc considered harmful for multiprocessor workloads. *IEEE Micro*.

[5] Calton Pu Jonathan Walpole Ashvin Goel, David Steere. Swift: a feedback control and dynamic reconfiguration toolkit. Technical report, 1998.

[6] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton, 2011.

[7] Dushyanth Narayanan James Eric Tilton Jason Flinn Kevin R. Walker Brian D. Noble, M. Satyanarayanan.

[8] G. Mariani G. Palermo C. Silvano V. Zaccaria C. Ykman-Couvreur, P. Avasare. Linking run-time resource management of embedded multicore platforms with automated design-time exploration. *Computers & Digital Techniques*, 2011.

[9] Th. Marescaux E. Brockmeyer Fr. Catthoor H. Corporaal Ch. Ykman-Couvreur, V. Nollet. Design-time application mapping and platform exploration for mp-soc customised run-time management. *IET Computers & Digital Techniques*, 2007.

[10] Fr. Catthoor H. Corporaal Ch. Ykman-Couvreurl, V. Nollet. Fast multidimension multi-choice knapsack heuristic for mp-soc run-time management. *International Symposium on System-on-Chip*, 2006.

[11] Christian Leuxner Wassiou Sitou Bernd Spanfelner Cornel Klein, Reiner Schmid. A survey of context adaptation in autonomic computing. *Autonomic and Autonomous Systems*, 2008.

[12] Gianluca Palermo Vittorio Zaccaria Fabrizio Castro Marcos Martinez Sara Bocchio Roberto Zafalon Prabhat Avasare Geert Vanmeerbeeck Chantal Ykman-Couvreur Maryse Wouters Carlos Kavka Luka Onesti Alessandro Turco Umberto Bondik Giovanni Marianik Hector Posadas Eugenio Villar Chris Wu Fan Dongrui Zhang Hao Tang Shibin Cristina Silvano, William Fornaciari. Multicube: Multi-objective design space exploration of multi-core architectures. *2010 IEEE Computer Society Annual Symposium on VLSI*, 2010.

[13] S.G. Dropsbo S. Dwarkadas E.G. Friedman M.C. Huang V. Kursun G. Magklis M.L. Scott G. Semeraro P. Bose A. Buyuktosunoglu P.W. Cook S.E. Schuster D.H. Albonesi, R. Balasubramonian. Dynamically tuning processor resources with adaptive processing. *Computer*, 2003.

[14] Lothar Thiele Eckart Zitzler. *An evolutionary algorithm for multiobjective optimization: The strength pareto approach.* TIK-Report, 1998.

[15] Luciano Ost Florent Bruguier Gilles Sassatelli Pascal Benoit Lionel Torres Michel Robert G. Almeida, Rèmi Busseuil. Pi and pid regulation approaches for performance-constrained adaptive multiprocessor system-on-chip. In *IEEE Embedded Systems Letters*, 2011.

[16] Vittorio Zaccaria Gianluca Palermo, Cristina Silvano. Respir: A response surface-based pareto iterative refinement for application-specific design space exploration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2009.

[17] Cristina Silvano Vittorio Zaccaria Giovanni Mariani, Gianluca Palermo. Arte: An application-specific run-time management framework for multi-core systems. *IEEE 9th Symposium on Application Specific Processors (SASP)*, 2011.

[18] W.J. Gutjahr. Aco algorithms with guaranteed convergence to the optimal solution. *Information Processing Letters*, 2002.

[19] Twan Basten Marc Geilen Sander Stuijk Rob Hoes Hamid Shojaei, AmirHossein Ghamarian. A parameterized compositional multi-dimensional multiple-choice knapsack heuristic for cmp run-time management. In *DAC '09 Proceedings of the 46th Annual Design Automation Conference*, 2009.

[20] Steve Heath. *Embedded Systems Design.* Newnes, 2003.

[21] Marco D. Santambrogio Alberto Leva Anant Agarwal Henry Hoffmann, Martina Maggio. Technical report.

[22] Marco D. Santambrogio Alberto Leva Anant Agarwal Henry Hoffmann, Martina Maggio. Seec: A framework for self-aware computing. Technical report, 2010.

[23] Michael Carbi Sasa Misailovic Anant Agarwa Martin Rinard Henry Hoffmann, Stelios Sidiroglou. Dynamic knobs for responsive power-aware computing. In *ASPLOS XVI Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, 2011.

[24] Michael Carbin Sasa Misailovic Anant Agarwal Martin Rinard Henry Hoffmann, Stelios Sidiroglou. Dynamic knobs for responsive power-aware computing. In *ACM SIGARCH Computer Architecture News*, 2011.

[25] IBM. An architectural blueprint for autonomic computing. Technical report, 2006.

[26] Khaled Ghédira Inès Alaya, Christine Solnon. Ant colony optimization for multi-objective optimization problems. *19th IEEE International Conference on Tools with Artificial Intelligence*, 2007.

[27] Sameer Agarwal Kalyanmoy Deb, Amrit Pratap and T. Meyariva. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE Transactions on Evolutionary Computation*, 2002.

[28] K. et al. Keutzer. System-level design: orthogonalization of concerns and platform-based design. *Computer-Aided Design of Integrated Circuits and Systems , IEEE Transactions on*, 2000.

[29] G. De Micheli L. Benini. Networks on chips: a new soc paradigm. 2002.

[30] L. Marzario G. Lipari L. Palopoli, T. Cucinotta. Aquosa - adaptive quality of service architecture. *Software: Practice and Experience*, 2009.

[31] Cristina Silvano Leandro Fiorin, Gianluca Palermo. A monitoring system for nocs. In *roceedings of the Third International Workshop on Network on Chip Architectures*, 2010.

[32] D.S.Johnson M. R. Garey. Computers and intractability: A guide to the theory on np-completeness (a series of books in the mathematical sciences). *W. H. Freeman and Company*, 1979.

[33] J B MacQueen.

[34] Bart Theelen Ralph Otten Marc Geilen, Twan Basten. An algebra of pareto points. *Fundamenta Informaticae*, 2007.

[35] Marco D Santambrogio Anant Agarwal Alberto Leva Martina Maggio, Henry Hoffmann. Decision making in autonomic computing systems: Comparison of approaches and techniques. In *Proceedings of the 8th ACM international conference on Autonomic computing - ICAC '11*, 2011.

[36] Liu Sanyang Meng Hongyun. Ispea: improvement for the strength pareto evolutionary algorithm for multiobjective optimization with immunity. In *Fifth International Conference on Computational Intelligence and Multimedia Applications*, 2003.

[37] B. Jacob P. Kohout, B. Ganesh. Hardware support for real-time operating systems. In *First IEEE/ACM/IFIP International Conference on Hardware/ Software Codesign and Systems Synthesis*, 2003.

[38] Kang G. Shin Padmanabhan Pillai. Real-time dynamic voltage scaling for low-power embedded operating systems. In *SOSP '01 Proceedings of the eighteenth ACM symposium on Operating systems principles*, 2001.

[39] William Fornaciari Patrick Bellasi, Giuseppe Massari. A rtrm proposal for multi/many-core platforms and reconfigurable applications. In *Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC) , 2012 7th International Workshop*, 2012.

[40] Engin Ipek Ramazan Bitirgen and Jose F. Martinez. Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach.

[41] Michael Van Biesbrouck Timothy Sherwood Brad Calder rez Perelman, Greg Hamerly.

[42] Robert W. Wisniewski Reza Azimi, Michael Stumm. Online performance analysis by statistical sampling of microprocessor performance counters. In *Proceedings of the 19th annual international conference on Supercomputing*, 2005.

[43] Nikolaus Huber Ralf Reussner Samuel Kounev, Fabian Brosig. Towards self-aware performance and resource management in modern service-oriented systems. *Services Computing (SCC)*, 2010.

[44] Henry Hoffmann Martin Rinard Sasa Misailovic, Stelios Sidiroglou. Quality of service profiling. 2010.

[45] Donald Yeung Seungryul Choi. Learning-based smt processor resource distribution via hill-climbing. *CM SIGARCH Computer Architecture News*, 2006.

[46] Mohammed El Shobaki. On-chip monitoring of single- and multiprocessor hardware real-time operating systems. In *IEEE 8th International Conference On Real-Time Computing Systems and Applications*, 2002.

[47] Lothar Thiele Eckart Zitzler Stefan Bleuler, Martin Brack.

[48] R. W. Brodersen T. D. Burd. Energy efficient cmos microprocessor design. In *Proceedings of the 28th Annual Hawaii International Conference on System Sciences*, 1995.

[49] G. Bjontegaard A. Luthra T. Wiegand, G. J. Sullivan. Overview of the h.264/avc video coding standard. *EEE Transactions on Circuits and Systems for Video Technology*, 2003.

[50] Luigi Palopoli Giuseppe Lipari Tommaso Cucinotta, Luca Abeni. A robust mechanism for adaptive scheduling of multimedia applications. *ACM Transactions on Embedded Computing Systems*, 2011.

[51] F. Castro C. Silvano G. Mariani V. Zaccaria, G. Palermo. Multicube explorer: An open source framework for design space exploration of chip multi-processors. *23rd International Conference on Architecture of Computing Systems (ARCS)*, 2010.

[52] Andrea Di Gesare Vincenzo Consales. Argo: un framework per il monitoring e la riconfigurazione di applicazioni in architetture multiprocessore. Master's thesis, Politecnico di Milano, 2011.

[53] Trishul M. Chilimbi Woongki Baek. Green: a framework for supporting energy-conscious programming using controlled approximation.