Politecnico di Milano
Facoltà di Ingegneria dell'Informazione
Corso di Laurea Specialistica in Ingegneria Informatica

ARGO: UN FRAMEWORK PER IL MONITORING E LA RICONFIGURAZIONE DI APPLICAZIONI IN ARCHITETTURE MULTIPROCESSORE

Relatore: Prof.re Gianluca Palermo
Correlatore: Prof.re Vittorio Zaccaria

Tesi di Laurea di:
Vincenzo Consales
Matricola n. 750581
Andrea Di Gesare
Matricola n. 750483

Anno Accademico 2010-2011

Milano, 23 Aprile 2012

Ai miei genitori e alla mia famiglia.
Andrea

Alla mia famiglia e a tutti i miei amici
che mi hanno aiutato e sopportato in
questa avventura lunga cinque anni.
Vincenzo

Indice

Elenco delle figure

Elenco delle tabelle

Sommario

Questo lavoro di tesi descrive un nuovo framework, ARGO, sviluppato per semplificare il ciclo di sviluppo di un'applicazione introducendo la gestione a run-time della stessa in modo quasi trasparente.

Il framework definito introduce il concetto di goal di un'applicazione ed in base ad esso gestisce a tempo d'esecuzione l'evolversi dello stato dell'applicazione stessa. L'introduzione del concetto di goal permette un controllo preciso dell'applicazione e un miglioramento delle performance globali del sistema a fronte di un'analisi iniziale dell'applicazione.

Il flusso di progettazione delle applicazioni interfacciate con ARGO si divide in due fasi principali. La prima fase si occupa della caratterizzazione dell'applicazione attraverso diverse tecniche a design-time come ad esempio quella della Design Space Exploration (DSE). La seconda fase richiede uno sforzo minimo da parte del programmatore per l'interfacciamento tra l'applicazione e il framework. Essa permette di valutare a run-time, grazie all'Application-Specific Run-Time Manager e alle informazioni ricavate dalla precedente analisi, l'esecuzione dell'applicazione in relazione ai goal definiti.

L'approccio utilizzato permette di disaccoppiare la fase di analisi dalla gestione a run-time minimizzando quindi l'overhead di controllo. Lo sviluppo del framework con l'architettura appena descritta permette al programmatore di definire un goal da rispettare e di disinteressarsi della gestione a run-time dell'applicazione. Questo porta ad ovvi benefici per il ciclo di sviluppo dell'applicazione stessa e per la sua gestione.

Abstract

This thesis describes a new framework, called ARGO, aimed at simplifying applications design flow enabling developers to define with a minimum effort their own applications goals and making achieve them.

ARGO defines the concept of goal of an application and, according to its value, it manages at run-time the execution flow of the application itself. The introduction of goals allows to have an accurate control of the application and an improvement of the global system perfomance after an preliminary analysis of the application has taken place.

Our design flow is mainly composed of two different phases: the former is made at design-time and consists in a detailed characterization of the application through Design Space Exploration (DSE), the latter consists of run-time techniques, based on the information gathered at design-time and from the current environment.

This approach allows to decouple the analysis phase from the run-time management, hence minimising the control overhead. Developers can define their applications goals and then let the framework manage them, without the need of their intervention. This feature leads to a benefit in the process of software developement and run-time management.

Capitolo 1

Introduzione

Nel corso degli anni le architetture dei processori hanno subito una profonda evoluzione. Si è partiti da architetture semplici e con un solo processore sino ad arrivare ad architetture multiprocessore di complessità superiore che sono oggi considerate lo standard per i sistemi di elaborazione. Questo cambiamento è motivato dal limite tecnologico raggiunto dalle soluzioni che ricercano un aumento delle prestazioni ottenuto incrementando le frequenze di elaborazione. I limiti di questo approccio si sono manifestati all'inizio del nuovo millennio quando i consumi di potenza e il calore dissipato prodotti dall'aumento del numero di transistor insieme all'aumento delle frequenze sono diventati insostenibili. Questi limiti hanno spinto i progettisti hardware a ripensare i processori seguendo un approccio differente passando quindi dall'aumento delle frequenze all'aumento del numero di processori per singolo chip. La progettazione e l'impiego di sistemi multiprocessore composti sia da diversi core omogenei sia da core eterogenei è, ad oggi, estesa a tutte le categorie di impiego, dai supercomputer ai sistemi embedded. I sistemi multiprocessore hanno portato notevoli vantaggi a tutte le categorie in cui sono stati impiegati e hanno velocemente rimpiazzato le soluzioni a singolo processore. Il profondo cambiamento delle architetture ha portato ad un conseguente modifica

nell'approccio delle applicazioni nei confronti del proprio modello computazionale. Si è passati da un approccio orientato alle prestazioni ad uno orientato ai goal. Questo cambiamento è stato supportato dagli sviluppatori sempre più attenti all'efficienza e consci dei problemi derivanti da una cattiva gestione della risorse. Mentre il problema è molto chiaro la sua soluzione non è così semplice. Allo stesso tempo non è banale la creazione di applicazioni che sfruttino al pieno le nuove architetture cercando di raggiungere i propri goal.

In questo lavoro di tesi ci occuperemo di sfruttare sia in fase di progettazione sia a run-time alcune strategie per creare un metodo alternativo ed efficace di self-aware computing [1]. Per raggiungere questo obiettivo è stato creato il framework ARGO[1] in cui sono presenti i seguenti concetti chiave: analisi a design-time, monitoring dell'applicazione, Quality of Service (QoS) e run-time decision making. Il nostro framework mira a determinare, in fase di progettazione, la relazione tra i parametri dell'applicazione e le sue metriche, come throughput, consumo di energia e precisione. Le informazioni ricavate sono utilizzate per creare un modello dell'applicazione, caratterizzando i parametri (conosciuti anche come dynamic knobs [2]) attraverso i quali effettuare diverse configurazioni, valutando gli effetti prodotti sulle prestazioni globali dell'applicazione. Una volta che tale modello è stato creato, può essere usato per regolare l'applicazione dinamicamente permettendole di adattare il proprio comportamento in base alle risorse disponibili e ai vincoli correnti.

Per sviluppare il nostro framework abbiamo utilizzato un approccio a due livelli. Il primo si occupa di gestire i parametri e le risorse di sistema mentre il secondo si occupa del supporto alle applicazioni. I componenti associati a questi livelli vengono chiamati rispettivamente System-Level

[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'obiettivo della spedizione, il nostro framework ARGO rappresenta un supporto al raggiungimento dei goal delle applicazioni.

Resource Run-Time Manager (SYS-RTRM) e Application-Specific Run-Time Manager (AS-RTM). Seguendo questo approccio abbiamo chiamato Operating Point le configurazioni dei parametri dell'applicazione che l'AS-RTM può modificare autonomamente senza l'interazione con il SYS-RTRM. Abbiamo definito inoltre i Working Mode come le differenti configurazioni di risorse massime richieste dall'applicazione per funzionare in un determinato Operating Point; il Working Mode è selezionabile dal SYS-RTRM. Gli obiettivi di questo lavoro di tesi sono i seguenti:

- fornire una metodologia efficace per la Design Space Exploration che permettere di ricavare un set ottimale di Operating Point e Working Mode.
- disaccoppiare le attività di run-time management in design-time e run-time, consentendo al run-time manager di operare con overhead minimo.
- fornire un approccio distribuito al run-time management, dove ogni decisione è presa al livello in cui vi è una maggiore conoscenza del problema.
- consentire agli sviluppatori di definire una serie di goal per la loro applicazioni e procedere alla loro realizzazione senza dover monitorare l'esecuzione run-time.

La struttura della tesi è organizzata su nove capitoli, di cui il presente rappresenta il primo. La tesi prevede un capitolo, *Definizioni e background* (capitolo 2), con lo scopo di fornire i concetti base utilizzati in tutto il resto dello scritto. Nel capitolo successivo, chiamato *Stato dell'arte* (capitolo 3), vengono analizzati gli studi precedentemente effettuati a vari livelli: hardware, sistema operativo e applicativo.

Il Capitolo *Metodologia proposta* (4) illustra la metodologia e le strategie utilizzate per lo sviluppo del framework ARGO. Anche questo capitolo presenterà concetti essenziali alla compresione delle tecniche utilizzate

nei capitoli seguenti. Il Capitolo *Architettura di monitoring* (5) introdurrà una della parti fondamentali di questo lavoro di tesi. Descriverà a fondo l'architettura dei monitor implementati appositamente per questa tesi spiegando le scelte effettuate e alcuni aspetti particolari. I due capitoli seguenti, *Tecniche a Design-Time* e *Tecniche a Run-Time* (rispettivamente capitoli 6 e 7), descrivono rispettivamente le strategie utilizzate a Design-Time e a Run-Time coprendo così tutti gli aspetti dell'analisi dell'applicazione e della sua esecuzione. Nel Capitolo *Casi d'uso e valutazioni sperimentali* (8) valuteremo alcuni casi reali di utilizzo del framework. I dati ottenuti da questi test verranno utilizzati per dimostrare come le tecniche sviluppate in ARGO possano essere realmente applicate. Infine nel Capitolo *Conclusioni* (9) termineremo il lavoro di tesi riassumendo i risultati ottenuti e proponendo alcuni sviluppi futuri.

Capitolo 2

Definizioni e background

In questo capitolo verranno introdotti vari argomenti necessari per la piena comprensione dei successivi capitoli e per capire quali sono i concetti e le idee che hanno originato questo lavoro di tesi.

Si parlerà in primo luogo del concetto di self aware, in particolare riguardante il software. Successivamente verranno esaminati i concetti di runtime management e verranno descritte le sue caratteristiche così da avere un'architettura di riferimento per i run-time manager di cui parleremo. Verranno date alcune definizioni riguardanti l'algebra di Pareto e verrà descritto il problema dello zaino. Come ultimo argomento verrà trattato il progetto 2PARMA e il framework BOSP.

2.1 Self aware computing

Da molti anni, nel mondo informatico, è stato introdotto il concetto di self-aware o autonomic computing. La nozione di self-awareness riguarda tutta quelle componenti hardware o software in grado di adattarsi e ottimizzare la propria esecuzione valutando il proprio stato e l'ambiente in cui operano.

Questo concetto è presente da molto tempo, seppure in modo inconscio, in molti film di fantascienza e nell'immaginario collettivo. Pensando ad esempio al famoso film 2001: Odissea nello spazio, in cui il computer HAL 9000 era dotato di intelligenza artificiale ci si rende conto di come un concetto abbastanza immediato come quello di intelligenza artificiale non potrebbe esistere senza quello di autocoscienza o self-aware.

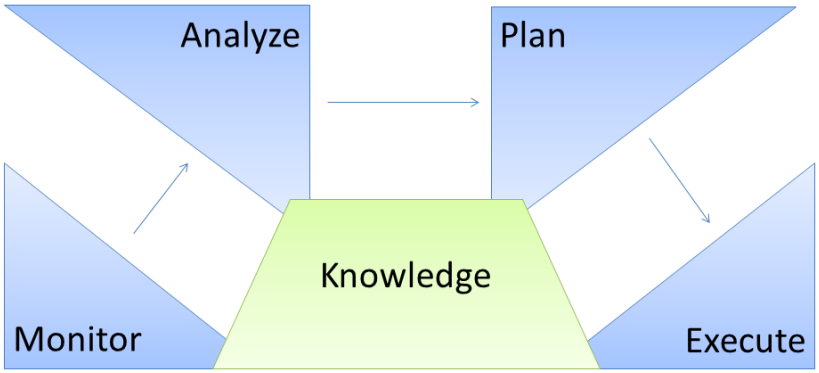


Fig. 2.1: Ciclo MAPE

In figura 2.1 possiamo vedere il ciclo di controllo MAPE (Monitor Analyze Plan Execute) per un'applicazione self-aware descritto da IBM [1]. É composto da quattro diverse fasi:

- Monitora: questa fase fornisce i meccanismi che raccolgono, aggregano e filtrano i dati raccolti dalla risorsa gestita.
- Analizza: la fase di analisi fornisce i modelli e le correlazioni ricavate dai dati osservati nella fase precedente.
- Pianifica: questa fase, situata logicamente dopo la fase di analisi, si occupa di programmare le azioni future sulla base delle analisi effettuate.
- Esegui: dopo aver pianificato le operazione da svolgere in relazione ai dati osservati si passa alla loro esecuzione, modificando quindi i parametri di controllo.

In aggiunta alle quattro fasi appena descritte, l'immagine presenta il blocco *Knowledge*. Esso consiste in un insieme di strutture e di dati come registri, dizionari e database che consentono l'accesso alla base di conoscenza necessaria alle fasi appena presentate. Tornando al concetto di self-aware computing possiamo valutare la complessità insita in tutti quei dispositivi facenti parte di questa categoria. Il numero di ambienti nel quale un dispositivo di questo tipo può operare sono infiniti come del resto anche le metriche di valutazione per l'ottimizzazione del proprio comportamento. Le analisi che si possono effettuare riguardano obiettivi predeterminati (chiamati anche goal): performance computazionali, energetiche o di altro tipo. Per quanto riguarda gli ambienti operativi possiamo pensare ad un ambiente software più o meno complesso in cui diverse applicazioni devono rispettare dei vincoli oppure ad un ambiente reale in cui un dispositivo hardware deve variare il proprio comportamento in relazione a degli stimoli esterni.

Citando la classificazione di Klein et al. [3] vediamo quali caratteristiche deve avere un sistema con modello computazionale self-aware, sia esso software o hardware:

- Adattabilità: capacità di modificare il comportamento del sistema osservato. Tipicamente il comportamento del sistema viene modificato a seguito di un input di un utente. Invece, facendo riferimento al concetto di *adattamento automatico* abbiamo che il sistema decide autonomamente la modifica del proprio comportamento.

- Awareness: questo concetto è fortemente legato all'adattabilità ed è un prerequisito dell'adattamento automatico. Rappresenta la capacità di osservare l'ambiente circostante ed il proprio comportamento.

- Monitoring: rappresenta la capacità di controllare lo svolgimento della propria esecuzione identificando gli eventuali errori.

- Dinamicità: esprime la capacità di un sistema di adattarsi durante l'esecuzione, quindi di modificare il proprio comportamento interno senza modificare i valori osservabili all'esterno.

- Autonomia: questa proprietà rappresenta la capacità di essere autonomi rispetto all'utente, quindi la capacità di decisioni complesse senza l'intervento umano.

- Robustezza: la robustezza rappresenta l'affidabilità di un sistema a fronte di disturbi o errori, quindi la capacità di continuare a funzionare correttamente anche dopo un evento imprevisto.

Da questo breve elenco di caratteristiche predominanti è ancora più chiaro come in un sistema self-aware le varie componenti debbano collaborare al fine di raggiungere i goal sui propri parametri in autonomia.

Questo lavoro di tesi si propone di esplorare solamente questi concetti a livello software. Partendo da questa esigenza è stata effettuata un'approfondita analisi delle soluzioni già presenti nello stato dell'arte ed una successiva implementazione ad hoc per adempiere alle esigenze incontrate durante la fase di sviluppo.

2.2 Run-time management

Nella sezione 2.1 si è visto che può esistere una relazione tra l'applicazione e l'ambiente in cui viene eseguita. Nel caso sussista questa relazione, l'applicazione deve valutare i parametri dell'ambiente in cui si trova ad operare. I parametri secondo cui l'applicazione viene eseguita sono vari e possono cambiare secondo le impostazioni del framework a cui è demandata la loro gestione. A titolo esemplificativo si può pensare ad un'applicazione eseguita inizialmente avendo a disposizione una certa quantità di memoria che a seguito di alcuni eventi viene dimezzata. Un altro caso possibile riguarda la diminuzione della frequenza della CPU a seguito di un allarme termico.

Il componente che si occupa dei cambiamenti di stato come quelli appena illustrati è chiamato Run-Time Manager (RTM). Questo componente si occupa di cambiare le condizioni di esecuzione a run-time per rispettare le politiche imposte dal sistema e dall'utente. Queste politiche possono essere diverse e variano dalla condivisione di risorse tra più applicazioni alla massimizzazione del throughput passando per il risparmio energetico.

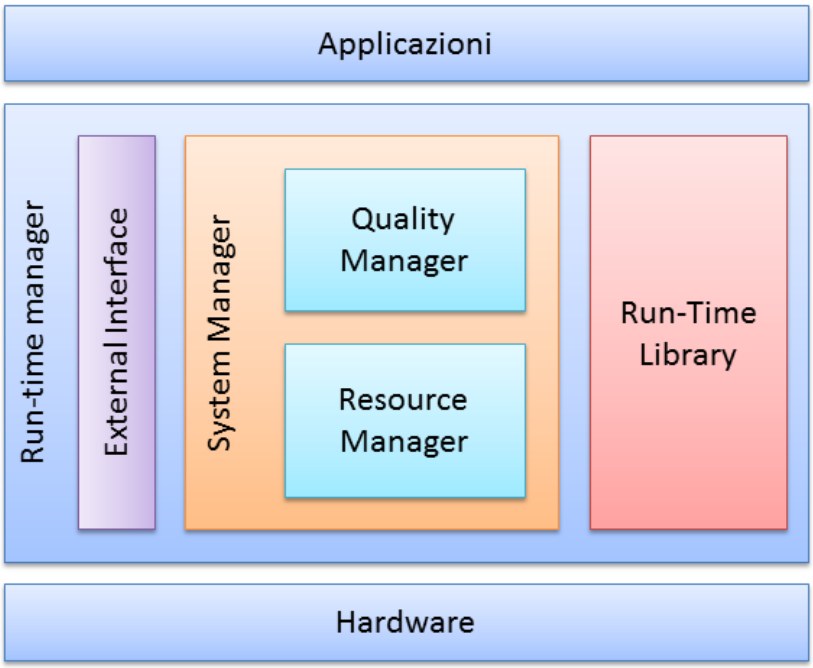


Fig. 2.2: Struttura del run-time manager

Secondo la classificazione di Nollet et al. [4] il run-time manager, la cui struttura è riportata in Figura 2.2, si troverà logicamente tra il livello applicativo e il livello hardware in quanto necessita si conoscenze estese di entrambi gli ambienti.

Il run-time manager è composto da un System Manager contenente un Quality Manager e un Resource Manager. Il primo si occuperà di interagire con l'applicazione per rispettare i criteri di qualità richiesti dall'utente mentre il secondo si occuperà della gestione efficiente di tutte le risorse disponibili. In aggiunta al system manager è anche presente una

Run-Time Library che si occuperà dell'interfacciamento tra l'applicazione e le altre componenti.

2.2.1 Quality Manager

Il compito del quality manager è quello di garantire un'elaborazione ottimale mediando tra le risorse disponibili nel sistema e le esigenze dell'applicazione.

Tipicamente questo componente offre due specifiche funzioni: la prima è la gestione della *Quality of Experience* (QoE) e la seconda è la selezione della modalità di funzionamento o del punto operativo. La prima funzionalità consiste nella gestione e nella classificazione dei profili di qualità per ogni applicazione. La fase di selezione della modalità o del punto operativo si occupa di scegliere il corretto livello di qualità per ogni applicazione cercando di massimizzare l'obiettivo generale imposto dal sistema.

Il concetto di qualità riguardo alle risorse di sistema dipende dalle politiche imposte e può coincidere con l'esecuzione contemporanea del massimo numero di applicazioni o con l'esecuzione prioritaria di alcune di esse. Definire il concetto di qualità delle applicazioni è più complesso in quanto in primo luogo bisogna definire quali sono i parametri che alterano la qualità di un'esecuzione e successivamente bisogna effettuare uno studio per esplorare quali sono le combinazioni che massimizzano gli obiettivi a vari livelli qualitativi, rispettando i diversi vincoli di risorse. Questo concetto verrà ripreso nel corso di questa tesi.

2.2.2 Resource Manager

Il resource manager è una componente che interviene successivamente al quality manager. Una volta selezionato il punto operativo, questa componente si occuperà di rilasciare le risorse all'applicazione garantendo la coerenza delle risorse rilasciate ad ogni applicazione rispetto alla di-

sponibilità totale. Questa componente, per una corretta assegnazione delle risorse, deve risolvere il problema MMKP definito precedentemente. Poichè queste scelte devono essere effettuate a run-time le strategia di soluzione devono rispettare un compromesso tra velocità di esecuzione e accuratezza della soluzione. Per avere una soluzione che rispetti i tempi di una esecuzione real-time è necessario affidarsi ad algoritmi di tipo euristici, quindi non esatti, per risolvere il problema MMKP.

2.2.3 Run-time library

Oltre alle componenti appena descritte troviamo una libreria che si occupa di gestire la comunicazione tra il livello hardware e quello applicativo. La run-time library ha due compiti principali. Il primo è quello di implementare delle primitive API adatte ad astrarre il livello hardware mentre il secondo è quello di creare un'interfaccia con i vari livelli del system manager.
Le primitive messe a disposizione da questa libreria sono di tre tipologie. Primitive per il *quality management*: si occupano si collegare il quality manager dell'applicazione con il quality manager dell'intero sistema, così da permettere all'applicazione di dialogare con il manager e rinegoziare la propria posizione.

Primitive per il *data management*: queste primitive vengono utilizzate tipicamente per le gestione della memoria in un sistema o per il passaggio di messaggi. Primitive per il *task management*: sono le primitive che si occupano della gestione dei processi, della loro creazione e distruzione.

2.3 Pareto ottimalità

In informatica sono molto frequenti problemi decisionali multi-obiettivo in cui va ottimizzato l'uso contemporaneo di più risorse. Noi abbiamo affrontato un problema di questo tipo per la valutazione della qualità del servizio (QoS) e di altre metriche. In generale, l'ottimizzazione di più

risorse, quindi la selezione di una configurazione ottima, richiede una design-space exploration multidimensionale [5]. I concetti e le tecniche di design-space exploration utilizzate in questo lavoro di tesi verranno approfondite nel Capitolo 6 mentre in questa sezione verranno illustrati concetti legati all'analisi di Pareto. Questo tipo di analisi vengono utilizzate per alleggerire il carico computazionale necessario a prendere decisioni a run-time. Nei problemi di ottimizzazione o decisione con criteri multipli bisogna tenere conto di obiettivi dipendenti o in conflitto. La dipendenza è data correlazione tra più obiettivi come ad esempio tempo di esecuzione e throughput. La conflittualità invece indica l'eventualità che, per raggiungere un obiettivo siano richieste specifiche contrastanti tra loro.

Vilfredo Pareto diede una definizione di *massima ottimalità* in un sistema economico in cui definì questa proprietà come l'impossibilità di trovare una soluzione differente che migliorasse anche solo un obiettivo. Se una soluzione è migliore in un obiettivo rispetto ad un'altra e non peggiora nessun altro degli obiettivi essa è definita *dominante*.

I punti di pareto descrivono le diverse caratteristiche delle possibili configurazioni di un sistema, come: larghezza di banda utilizzata, tempo necessario per il completamento, consumo di energia. Possiamo chiamare queste caratteristiche di una configurazione quantità o metriche. Le quantità possono essere ordinate , parzialmente o totalmente, in termini di migliore e peggiore. Si noti che ordinare parzialmente significa che non tutte le quantità potrebbero essere confrontabili contrariamente al caso di un ordinamento totale. Geilen et al. danno alcune formulazioni matematiche riguardo l'algebra di Pareto [6]. Di seguito elencheremo quelle necessarie alla comprensione degli argomenti trattati.

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 di riferimento sia di minimizzazione e quindi che valori piccoli siano preferibili a valori grandi. La variante*

irriflessiva di $\preceq_Q$ è denotata con $\prec_Q$

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

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

Definizione 4 (Dominanza) *Se $\bar{c}_1, \bar{c}_2 \in \mathcal{S}$, allora $c_1 \preceq c_2$ se e solo se per ogni quantità Q_k di $\mathcal{S}$, $\bar{c}_1(Q_k) \preceq_{Q_k} c_2(Q_k)$. Se $\bar{c}_1 \preceq \bar{c}_2$, allora $\bar{c}_1$ è detto dominante rispetto a $\bar{c}_2$.*

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

Definizione 6 (Pareto equivalente) *Due insiemi di configurazioni $\mathcal{C}_1$ e $\mathcal{C}_2$ da uno spazio di configurazione $\mathcal{S}$ sono Pareto equivalenti se e solo se si dominano l'un l'altro: $C_1 \preceq C_2$ e $C_2 \preceq C_1$. Questa proprietà è denotata come $\mathcal{C}_1 \equiv \mathcal{C}_2$.*

2.4 Problema dello zaino

Nel paragrafo precedente abbiamo già introdotto il problema della scelta multi-dimensionale in cui si deve ottimizzare l'esecuzione di più applicazioni dividendo tra esse le risorse di sistema assegnate. Questo tipo di

procedimento richiede di selezionare esattamente una configurazione per ogni applicazione, massimizzando le risorse assegnate senza superarne i limiti. Un problema così definito è chiamato problema dello zaino. Essendo nel caso multidimensionale avremo un Multi-dimensional Multiple-choice Knapsack Problem (MMKP) [5,7]. Una ottimizzazione MMKP richiede una potenziale soluzione ogni volta che avviene un cambiamento nell'insieme della applicazioni in esecuzione, ad esempio ogni volta che un'applicazione viene eseguita o fermata.

Definiamo ora matematicamente il problema MMKP. Sia N il numero di applicazioni e N_i il numero di configurazioni dell'applicazione i $(0 \leq i < N)$. Sia R il numero di risorse e R_k il numero di risorse disponibili per il tipo k $(0 \leq k < R)$. Supponiamo che la configurazione j dell'applicazione i abbia valore v_{ij} e richieda le risorse r_{ijk} $(0 \leq k < R)$, con le variabili x_{ij} che indicano se il punto j dell'insieme di configurazione i è stato selezionato o meno. Il problema MMKP si risolve come descritto in [5,7]:

- Bisogna massimizzare:

$$\sum_{0 \leq n < N} \sum_{0 \leq j \leq N_i} x_{ik} v_{ij}$$

- Soddisfando i seguenti vincoli:

$$x_{ij} \in \{0,1\},\ 0 \leq i < N,\ 0 \leq j < N_i$$
$$\sum_{0 \leq j \leq N_i} x_{ij} = 1,\ 0 \leq i < N$$
$$\sum_{0 \leq i < N} \sum_{0 \leq j < N_i} x_{ij} r_{ijk} \leq R_k,\ 0 \leq k < R.$$

Il problema MMKP appena definito è un problema NP difficile. Ogni generico algoritmo che risolve il problema in modo esatto, ha una complessità esponenziale che aumenta con il numero di insiemi di configurazioni [8]. Per questo motivo sono state sviluppate tecniche euristiche, come quella

proposta in [7], che raggiungono soluzioni vicine a quella ottimale con tempi di esecuzione minori.

2.5 Il 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 many-core 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 e utilizzato in questo lavoro di tesi è chiamato *Barbeque* [9]; fa parte del progetto open-source che ha come acronimo Barbeque Open Source Project (BOSP). 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 livello di qualità 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 le due componenti. Questa comunicazione è necessaria a garantire una efficace gestione delle risorse e dei vincoli di qualità forniti dall'applicazione. Nonostante questo sia il framework di resource management utilizzato in questo lavoro di tesi, non verranno fornite altre informazioni a riguardo in quanto il modello di programmazione introdotto non è oggetto di questo tesi.

2.6 Conclusioni

In questo capitolo abbiamo fornito alcune nozioni utili per una piena comprensione dei capitoli successivi. Abbiamo introdotto alcuni concetti come self-aware e run-time management per poter meglio comprendere le architetture e i componenti trattati in dettaglio nel corso della tesi. Successivamente sono stati illustrati alcuni concetti e problemi matematici come l'algebra di Pareto e il problema dello zaino. A conclusione del capitolo è stato trattato il progetto 2PARMA. Nel seguente capitolo andremo ad analizzare lo stato dell'arte per gli argomenti di questo lavoro di tesi.

Capitolo 3

Stato dell'arte

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. Questi sistemi spesso svolgono le loro funzionalità a diversi livelli:

- 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.
- 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.

3.1 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 [10] 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). La Figura 3.1 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 [11] che l'utilizzo di metriche altamente dipendenti dall'architettura e dallo

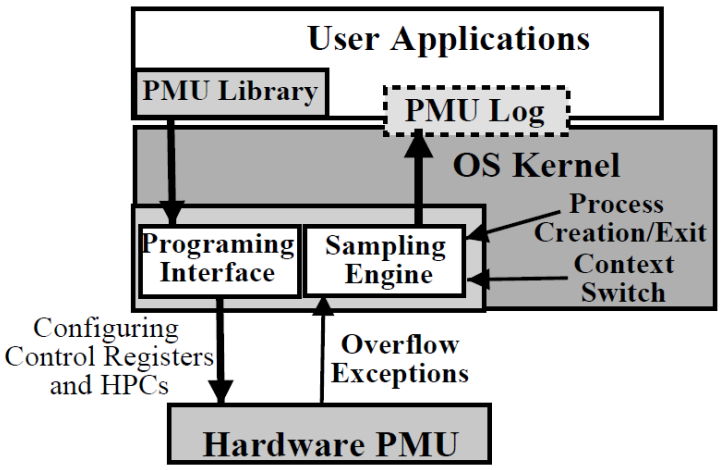


Fig. 3.1: Schema architetturale del lavoro di Azimi et al. [10]

scheduling fornito dal sistema operativo, come il numero di instruzioni 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. [12] invece, 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 sottomoduli. Essi possono essere attivati e disattivati in base ai dati forniti dal monitoring dei contatori hardware presenti nel sistema. La loro proposta (Figura 3.2) consiste in un processore superscalare con diversi componenti adatattivi: 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

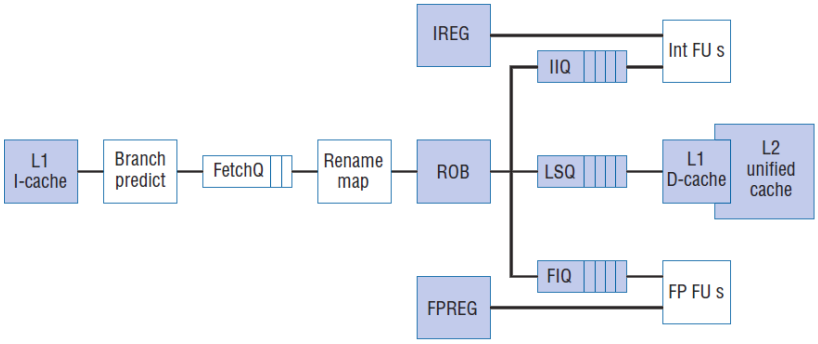


Fig. 3.2: Sistema adattivo proposto in [12]. Gli elementi ombreggiati sono stati ridisegnati per un supporto adattativo dell'architettura

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%.

Bitirgen et al. [13] 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 ottimizare le perfomance sono tutte basate direttamente o indirettamente sul numero di istruzioni per ciclo, con tutte le limitazioni del caso. La Figura 3.3 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 dal 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 de-

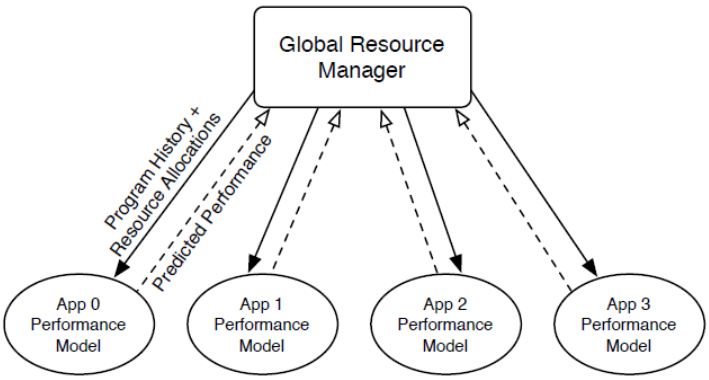


Fig. 3.3: Interazione tra il resource manager globale e ognuno dei modelli di performance applicativi (reti neurali) [13]

terminato numero di volte , con numero di risorse disponibili diverse, il resource manager prende la decisione che comporta il più alto incremento delle prestazioni.

Anche Choi e Yeung [14] propongono una soluzione simile. L'approccio è sempre basato sul machine learning ma in questo caso la soluzione proposta 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 di prestazioni. Anche in questo caso le metriche usate per la valutazione dei goal sono derivate dall'IPC. La Figura 3.4 rappresenta lo schema dell'architettura

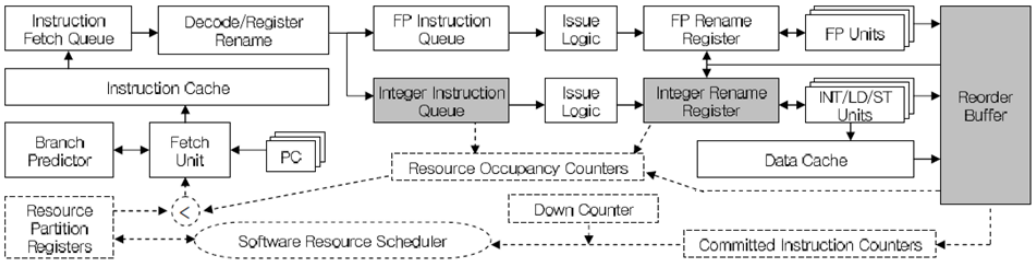


Fig. 3.4: Diagramma a blocchi del modello di processore SMT proposto da [14]

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 [15] descrive un sistema di monitoring specifico per sistemi operativi accelerati via hardware. La Figura 3.5 fornisce un'idea visiva dell'architettura proposta. 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 di appresta ad un uso veramente specifico come quello dei sistemi operativi real-time accelerati via hardware.

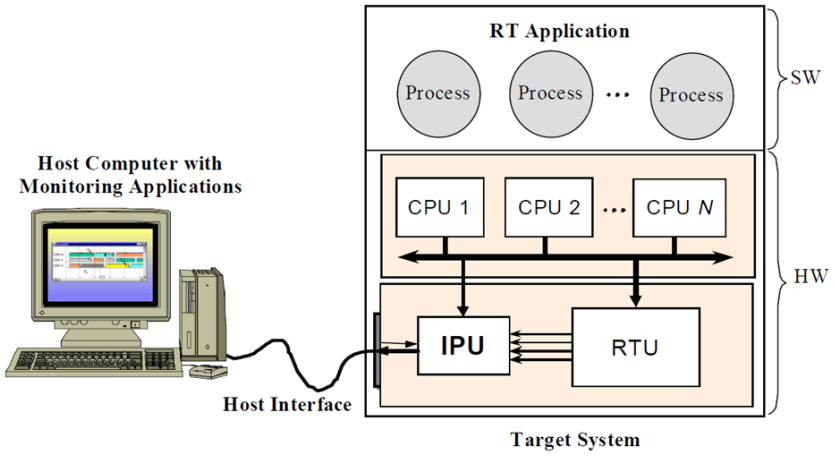


Fig. 3.5: Modello architetturale di MAMon [15]

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. [16] 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. [17]. 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 [18]. Per garantire queste caratteristiche, soprattutto in sistemi complessi come quelli multiprocessore, una delle funzionalità più importanti è quella di monitoring. In [17], gli autori si occupano di implementare queste funzionalità direttamente in hardware. Facendo riferimento alla Figura

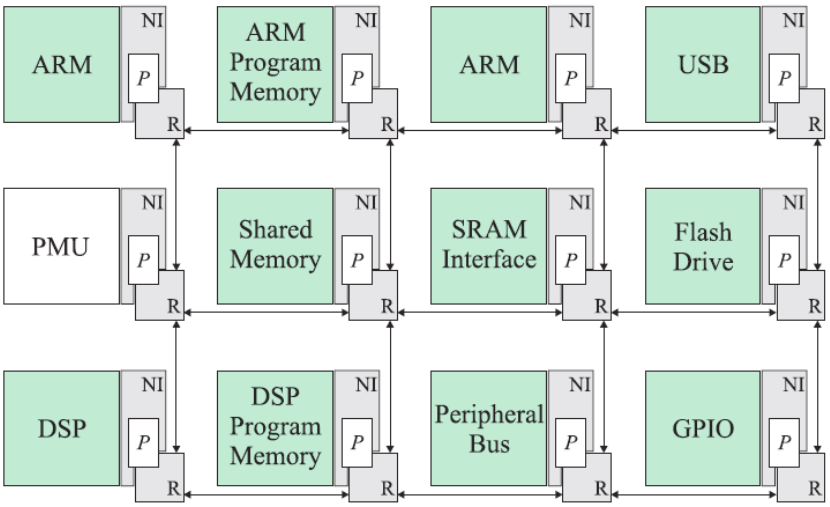


Fig. 3.6: Panoramica dell'architettura di monitoring per NoC [17]

3.6, è possibile notare la struttura eterogenea del sistema, nel quale alle

interfacce di rete (NI nell'immagine) sono stati 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 varie sonde, ottenerne i loro valori misurati ed effettuare le varie 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 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 arriva 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. Questo capitolo 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 [19]. 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 3.7 rappresenta una panoramica del sistema proposto. Com'è possi-

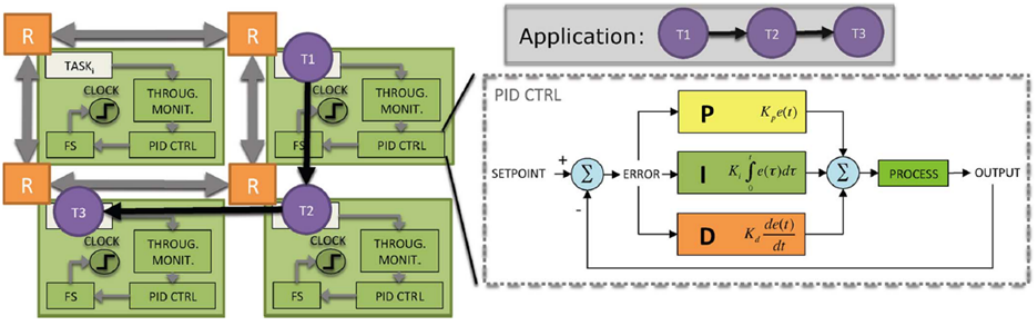


Fig. 3.7: Vista architetturale del sistema proposto con controllori PID [19]

bile 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 [20], un progetto di ricerca che estende il sistema operativo NetBSD [21] per permettere una collaborazione maggiore con le applicazioni. Le decisioni riguardo l'allocazione 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 [22, 23]. 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

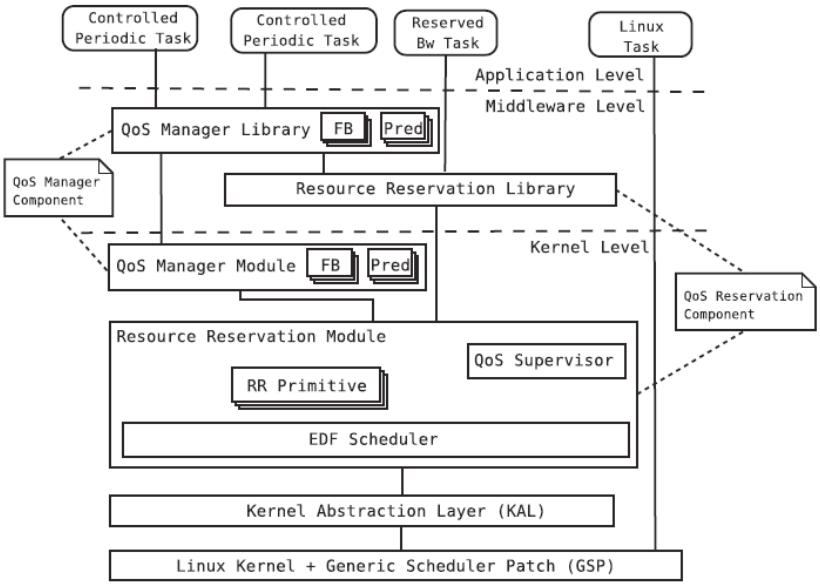


Fig. 3.8: Architettura di AQuoSA [22]

AQuoSA è visibile in Figura 3.8 ed è composta da diverse componenti tra le quali:

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

- 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
- 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 vari 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.

3.3 Livello applicativo

Come già descritto nella Sezione 3.2, è possibile astrarre le funzionalità di resource e run-time management sino al livello applicativo. In questo caso non sarà sempre possibile modificare parametri specifici del sistema come quelli relativi allo scheduling ma ci si dovrà affidare a tecniche diverse per raggiungere i risultati sperati. In questo ambito sono state sviluppate sia soluzioni diverse da quelle usate nelle sezioni precedenti sia altre tecniche più generali già usate nei precedenti livelli d'astrazione. Nel primo caso si parla ad esempio di compilatori specializzati o tecniche di Design Space Exploration, mentre nel secondo di teoria del controllo o di machine learning. Questa sezione descriverà alcuni lavori di ricerca svolti a livello applicativo sia riguardo la pura gestione delle risorse e delle applicazioni sia riguardo ad altre tecniche affiancate alle soluzioni standard.

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 loop [24]. 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 3.9, il flusso di sviluppo all'interno di Green si svolge in più fasi. Prima di tutto lo sviluppatore deve fornire sia le

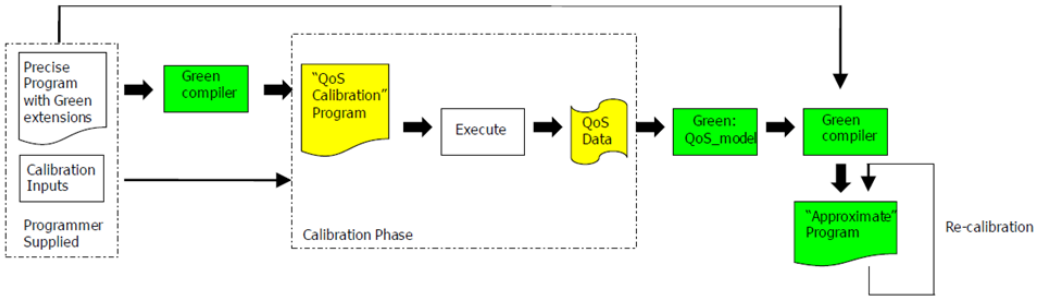


Fig. 3.9: Schema del sistema Green [24]

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. [25] 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 [26] è un'architettura basata sulla teoria del controllo specializzata su servizi internet. La Figura 3.10 delinea la metodologia di sviluppo per applicazioni che sfruttano ControlWare. La

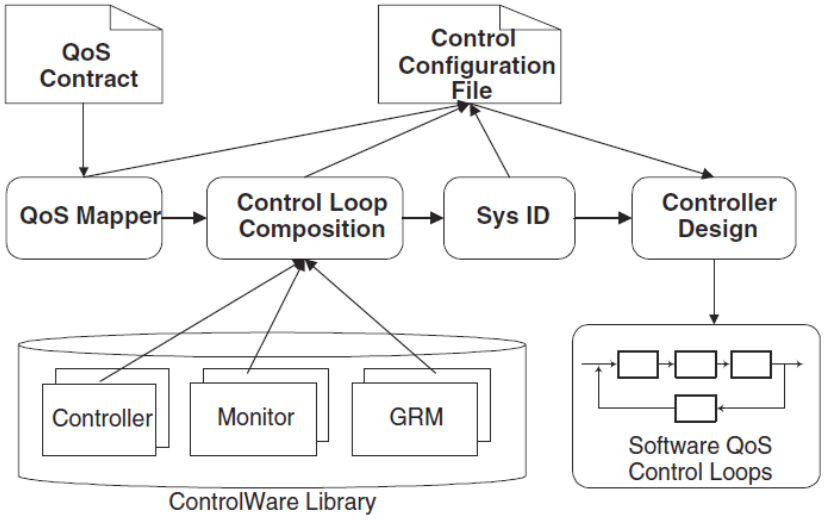


Fig. 3.10: Metodologia di sviluppo di ControlWare [26]

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 obiettivo 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 [27]) atto a risolvere tale problema. La fase successiva consiste nella configurazione dei vari monitor e degli attuatori forniti da 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 verrano 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 quella proposta in [7, 28, 29]. 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. La Figura 3.11 rappresenta il flusso che collega la fase di design a quella d'esecuzione. L'input del sistema è un insieme di diverse versioni della stessa applica-

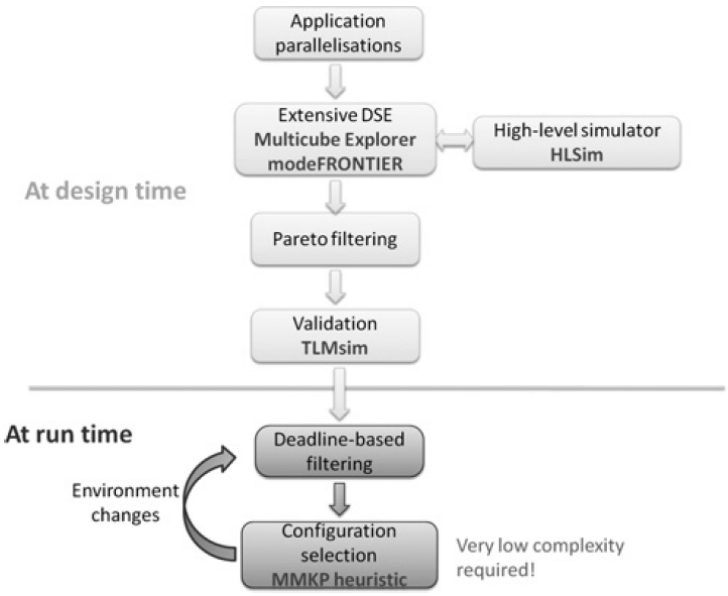


Fig. 3.11: Flusso di sviluppo tra fase di design e d'esecuzione [29]

zione sviluppata con livelli di parallelizzazione differenti. Queste versioni della stessa applicazione vengono usate per analizzare il comportamento dell'applicazione alle diverse frequenze dei processori. Questa fase può essere svolta da diversi motori di Design Space Exploration come ad esempio Multicube [30, 31], illustrato più avanti in questa sezione. 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 classificazione 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. [32], 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 3.12 rappresenta

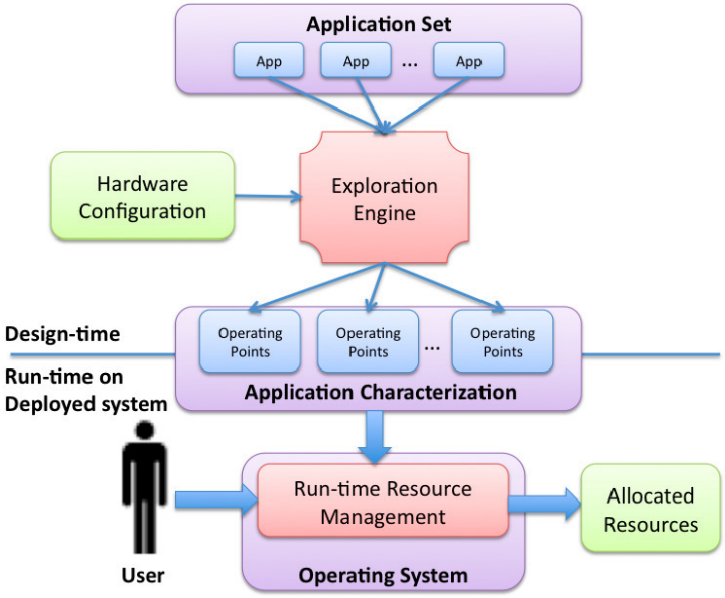


Fig. 3.12: Flusso di progettazione nel framework ARTE [32]

il flusso di progettazione di un'applicazione all'interno del framework. La strategia utilizzata è simile in alcuni aspetti a quella usata nel nostro

lavoro di tesi. 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 motore di Design Space Exploration usato da Mariani et al. è quello fornito dal framework Multicube [30,31]. 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 3.13 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,

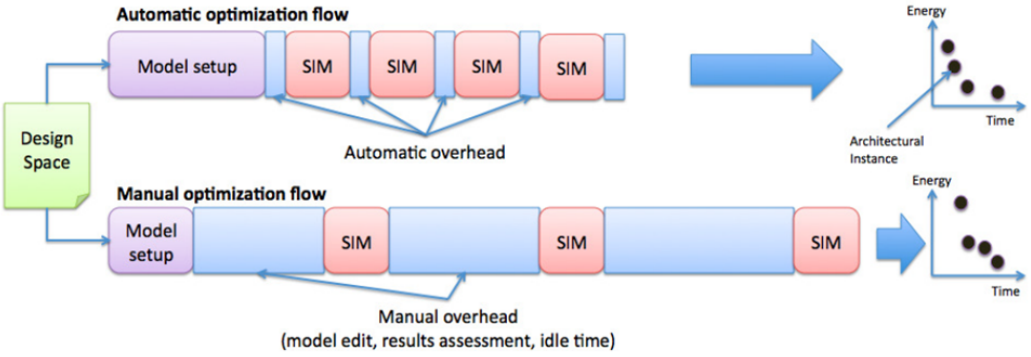


Fig. 3.13: Confronto tra una Design Space Exploration manuale ed una automatica [30]

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 3.14 rappresenta una panoramica della struttura di questo framework. Come già descritto

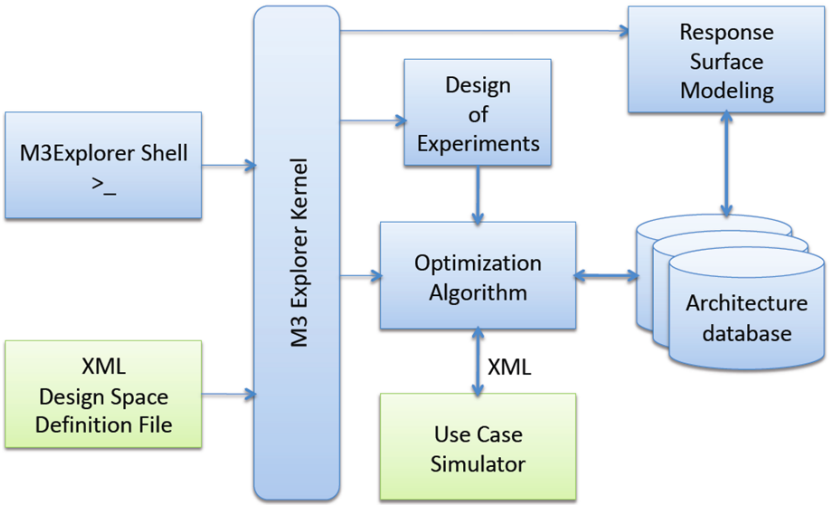


Fig. 3.14: Struttura del framework Multicube explorer [31]

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. Queste informazioni verrano 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.

Un raffinamento di queste strategie di Design Space Exploration è stato proposto da Palermo et al. e consiste nel framework ReSPIR [33]. Questo framework si concentra sullo studio di una corretta esplorazione che permetta di ottimizzare, col minor numero d'esecuzioni possibile, alcuni parametri applicativi come ad esempio il throughput o il consumo di potenza. La strategia usata consiste in un campionamento dello spazio di design, con lo scopo diminuire il numero di simulazioni da effettuare. I risultati forniti da questa fase permetteranno di effettuare un'appros-

simazione del comportamento del sistema tramite una tecnica chiamata Response Surface Methodology (RSM). Queste due fasi vengono effettuate iterativamente, migliorando l'approssimazione del modello all'aumentare dell'iterazioni effettuate. I test effettuati su questo framework confermano la bontà di questa tecnica rispetto ad altre metodologie euristiche allo stato dell'arte come MOSA [34] e NSGA-II [35]. Questi risultati sono ancor più validi soprattutto nel caso di un numero di simulazioni non elevato, dove i precedenti algoritmi soffrono la mancanza di un numero sufficiente di simulazioni necessarie per la convergenza del risultato.

Le tecniche di Design Space Exploration fornite da Multicube sono state utilizzate oltre che in [7, 28, 29, 32, 33] anche nel nostro lavoro di tesi. Il framework d'esplorazione utilizzato comprende le caratteristiche di Multicube ma ne estende le funzionalità rendendo più semplice la definizione di una Design of Experiment e di una successiva esplorazione. Nel Capitolo 6 verranno spiegati più dettagliatamente alcuni di questi concetti.

Come ultimo esempio di adattività 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 Massachussetts Institute of Technology e dal Dipartimento di Elettronica e Informazione del Politecnico di Milano. Consiste in un sistema autoadattivo capace di gestire a run-time applicazioni che espongono diversi goal al sistema [36–39]. 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 già descritto nel Capitolo 2, dove la fase di decisione è formata dai blocchi Analyze e Plan.

In Figura 3.15 possiamo vedere il ciclo di controllo di un'applicazione self-aware composto da tre diverse fasi:

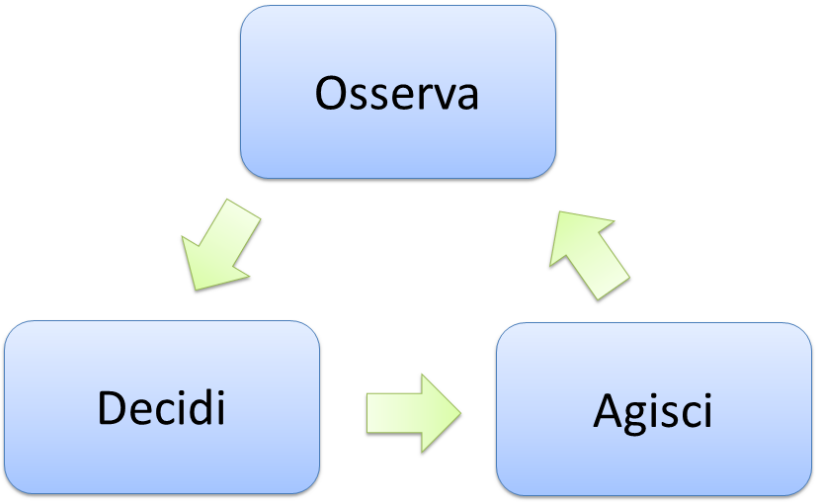


Fig. 3.15: Ciclo Observe-Decide-Act

- 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 [38], le caratteristiche interessanti di questo framework sono:

- 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 , [40–42]. Questo framework si basa sul concetto di Heartbeat che consiste in un'indicazione del progresso dell'esecuzione di un programma verso un goal definito in termini di throughput o latenza tra Heartbeat.

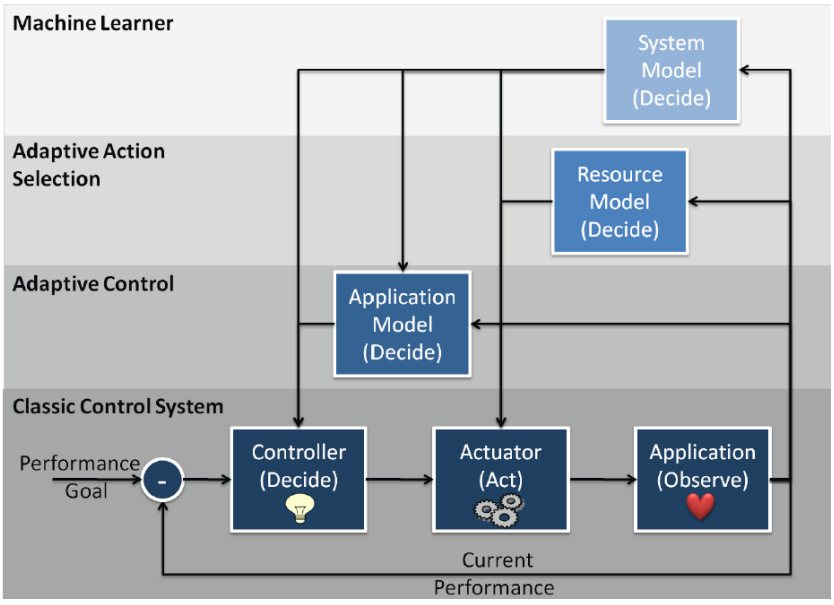


Fig. 3.16: Schema concettuale del framework SEEC [38]

La Figura 3.16 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 un 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 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).

3.4 Conclusioni

In questo capitolo sono stati illustrati alcuni lavori dello stato dell'arte nei diversi ambiti a cui fa riferimento questa tesi. Le diverse tecniche proposte variano ad esempio tra soluzioni hardware, software, statiche o dinamiche. I lavori analizzati si focalizzano spesso su problemi specifici e risultano quindi molto specializzati. Altri invece, nonostante cerchino di risolvere un problema più generale, garantiscono una certa efficienza e velocità d'esecuzione soltanto nel caso sia stato fornito un modello accurato dell'applicazione. Nonostante questa caratteristica essi non propongono nessuna strategia sul come creare questo tipo di modello. Poichè non esiste un'implementazione finale che risolva ottimamente questi problemi, abbiamo cercato di sfruttare al meglio le varie idee presenti in questo capitolo per proporre il nostro framework ARGO.

Capitolo 4

Metodologia proposta

Questo capitolo si pone l'obiettivo di descrivere la metodologia e le strategie seguite durante le fasi di progettazione e di sviluppo di questo framework. La descrizione di queste idee chiave saranno molto utili per capire meglio i concetti che verranno descritti nei capitoli seguenti. Più in dettaglio verrà esaminata e definita l'architettura del framework ARGO, descrivendone componenti e usi. Verrà inoltre effettuata una suddivisione concettuale del framework di monitoring creato. Per concludere verrà descritta la metodologia usata per supportare la creazione e l'ottimizzazione di un'applicazione dalla fase di design fino alla sua esecuzione.

4.1 Architettura di riferimento

L'architettura di riferimento per questo lavoro di tesi è quella illustrata in Figura 4.1. In questa piattaforma è presente un Run-Time Resource manager (RTRM) come quello descritto in [4] e nel Capitolo 2. I compiti del Run-Time Resource Manager sono svolti da due diverse entità dell'architettura: il System-Level Run-Time Resource Manager (SYS-RTRM) e un Application-Specific Run-Time Managers (AS-RTM) per ogni applicazione. Mentre il primo si concentra sulla allocazione delle risorse alle

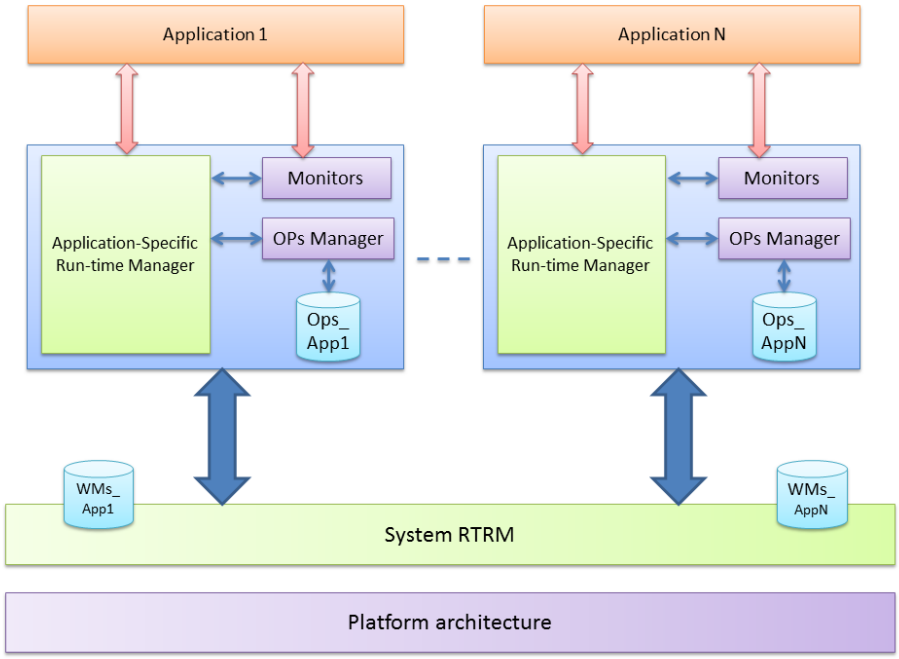


Fig. 4.1: Panoramica dell'architettura di riferimento

applicazioni, il secondo è responsabile di impostare dinamicamente i loro parametri al fine di raggiungere gli obiettivi prefissati. Ogni AS-RTM si basa su due componenti principali del framework: un gestore di *Operating Point* (OP Manager) e una serie di Monitor. Le loro funzioni e i loro usi saranno spiegati più in dettaglio nel Capitolo 5 e 7. Come è possibile notare inoltre dalla Figura 4.1, questa architettura si basa su due concetti: i punti operativi chiamati *Operating Points* (OPs) e le modalità di funzionamento, chiamate *Working Modes* (WMs). Gli Operating Point sono un concetto strettamente legato al dominio applicativo. Essi consistono in una struttura di base che descrive la combinazione dei diversi parametri di configurazione per l'applicazione corrente e contengono sia i parametri dell'applicazione che le metriche prodotte in fase di profiling a design-time. Queste metriche, quali il consumo di potenza o il throughput, possono essere usate per derivare un modello del comportamento dell'applicazione sotto diverse combinazioni di configurazioni. L'insieme di Operating Point

fa parte della conoscenza minima necessaria al nostro Application-Specific Run-Time Manager per operare correttamente.

A differenza degli Operating Point, i Working Mode hanno un dominio diverso da quello applicativo. Essi infatti rappresentano comunque un insieme di scenari dell'applicazione ma hanno parametri e uso differenti. I Working Mode rappresentano la conoscenza necessaria al System-Level Run-Time Resource Manager per decidere quante risorse allocare per una precisa applicazione. Ogni Working Mode è legato ad una insieme di punti operativi e rappresenta il massimo utilizzo di risorse in quella serie di punti. Nel caso del SYS-RTRM utilizzato nel nostro lavoro di tesi le risorse controllate sono solamente CPU e memoria. Ogni Working Mode ha un valore specifico che rappresenta la preferenza di un'applicazione ad essere eseguita con tale punto rispetto ad un altro. Il numero di Working Mode è per definizione inferiore o uguale al numero di Punti Operativi. Questo capitolo tratterà in particolar modo il livello applicativo dando qualche informazione sul livello di sistema quando necessario.

Analizzando più in dettaglio l'architettura proposta è necessario fare una precisazione riguardo la parte di monitoring. La nostra idea è stata quella di rendere il framework il più generale possibile ed usabile sia in fase di progettazione che d'implementazione. Inoltre, nonostante il lavoro sia stato svolgo nell'ambito di un particolare Resource Manager, abbiamo cercato di rendere il tutto più indipendente possibile. Per questo motivo abbiamo fornito anche delle funzionalità per monitorare non solo le prestazioni pure dell'applicazione ma anche altre caratteristiche più legate al sistema e alle risorse. La Figura 4.2 rappresenta graficamente la suddivisione logica e concettuale dei nostri monitor. Com'è possibile vedere dall'immagine, i monitor sono divisi logicamente in due gruppi: un gruppo di monitor orientato all'applicazione (MOA) ed uno orientato al sistema e alle risorse in generale (MOR). La classe base Monitor, istanziabile ed usabile come monitor generale, è stata invece considerata neutrale poiché i suoi usi sono dipendenti dal tipo di dato contenuto.

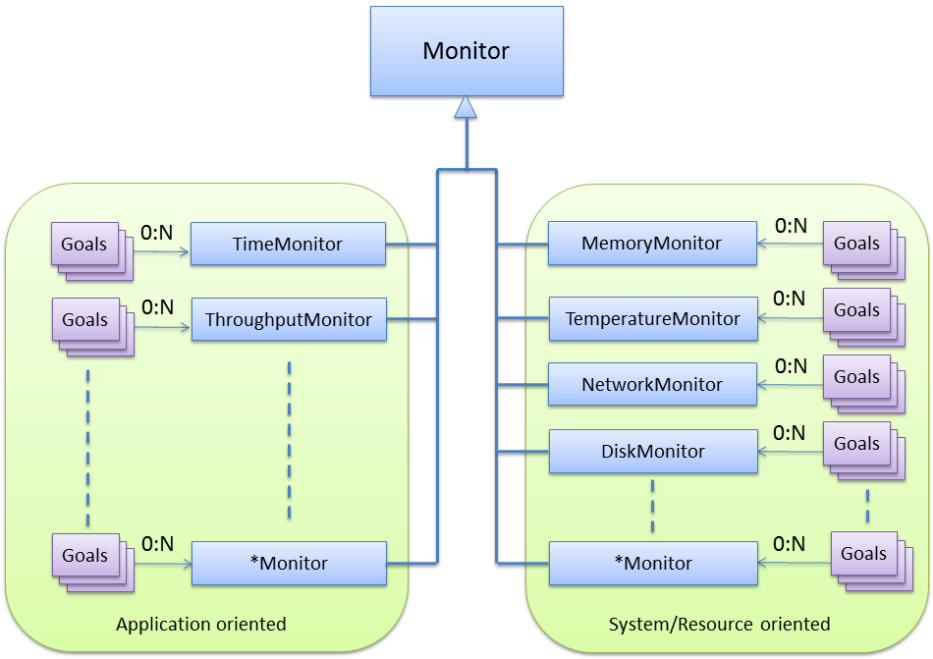


Fig. 4.2: Panoramica dell'architettura di monitoring

Le due tipologie di monitor sopracitate si differenziano per il loro uso; i monitor MOA si occupano di monitorare in generale metriche che sono tipiche dell'applicazione stessa e che possono essere influenzate in qualche modo tramite la modifica di alcuni parametri interni. In questo caso questi monitor verranno usati sia in fase di design per profilare e caratterizzare il comportamento dell'applicazione, sia in fase d'esecuzione per modificarne dinamicamente il comportamento e rispondere a variazioni delle performance dell'applicazione stessa.

Per descrivere il comportamento e gli utilizzi dei monitor MOR è meglio effettuare una breve precisazione riguardo a due possibili scenari d'utilizzo del nostro framework. Il primo è quello di un sistema in cui il System-Level Resource Manager si occupa sia di scegliere una allocazione adatta delle risorse dei processi che del monitorare e controllare il superamento di tali limiti. Il secondo invece, è dato da Resource Manager più semplici che si occupano soltanto di fornire un'indicazione delle

risorse da allocare per un processo, senza effettivamente controllare o forzare tale processo a non usare più risorse di quelle previste. Nel primo caso i monitor MOR saranno principalmente usati in fase di design per caratterizzare il consumo di risorse e informare il Resource Manager; nel secondo, invece, questi monitor potranno essere utili per controllare autonomamente e in maniera equa che le risorse allocate, non superino tali vincoli. Rispetto ai tipi di monitor visibili in Figura 4.2, soltanto le classi Monitor, TimeMonitor, ThroughputMonitor e MemoryMonitor sono state attualmente implementate.

4.2 Metodologia

Lo scopo di ARGO è quello di determinare in fase di progettazione la relazione tra i parametri applicativi e metriche quali ad esempio tempo d'esecuzione, consumo di potenza e precisione. La Figura 4.3 mostra il flusso di progettazione del nostro framework. Il primo passo consiste

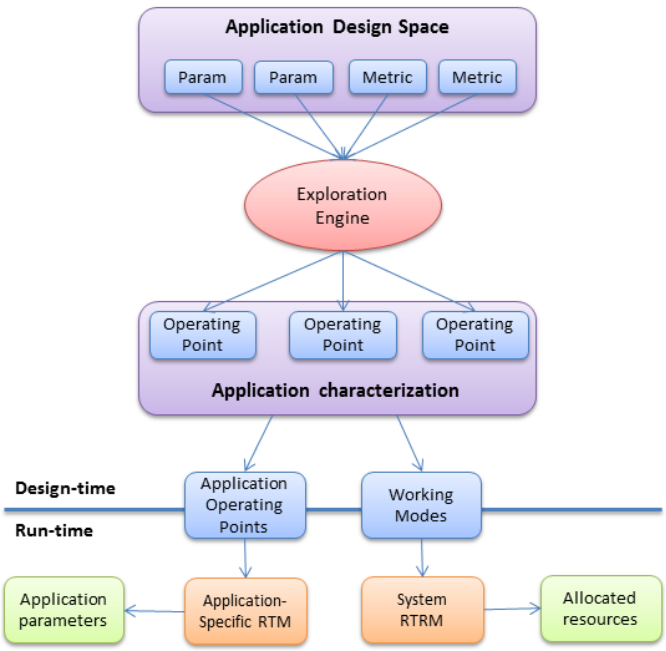


Fig. 4.3: Flusso di progettazione nel framework ARGO

nella definizione del design space dell'applicazione, scegliendo quali sono i parametri e le metriche da testare e valutare di conseguenza. Successivamente il software di esplorazione esegue l'applicazione con tutte le combinazioni di parametri definite fornendo, come risultato, le metriche richieste e un indicazione della correlazione tra ognuna di queste metriche e il variare delle configurazioni dei parametri. Le informazioni ricavate dai passaggi precedenti sono utilizzate per creare un modello dell'applicazione, caratterizzando i suoi parametri variabili altresì conosciuti come *dynamic knobs* [2], e i loro effetti sull'applicazione stessa. Terminata questa fase, i dati raccolti dalla fase di progettazione vengono utilizzati per costruire due differenti database. Il primo è costituito da un insieme di Operating Point, mentre il secondo dai Working Mode. Il Capitolo 6 spiegherà più in dettaglio come questi punti verranno ricavati.

Una volta che tali modelli sono stati creati, possono essere utilizzati con un duplice scopo. Essi consentono al SYS-RTRM di risolvere il problema della ripartizione delle risorse tra le applicazioni e ad ogni AS-RTM di ottimizzare le applicazioni dinamicamente, permettendo loro di adattare il proprio comportamento in base alle risorse disponibili e ai vincoli correnti come consumo di potenza o QoS. Il System-Level Resource Run-Time Manager per operare correttamente necessita di conoscere soltanto quali sono i diversi usi delle risorse di un applicazione e le sue preferenze riguardo i Working Mode. In questo modo potrà decidere per ogni applicazione qual è il miglior Working Mode attualmente disponibile. Questa decisione stabilisce in modo esplicito e incontrovertibile l'ammontare massimo delle risorse che un'applicazione può utilizzare. L'Application-Specific Run-Time Manager, una volta al corrente della scelta effettuata, si occuperà di prendere una decisione relativa al solo dominio applicativo settando i parametri interni in accordo con i vincoli forniti dal SYS-RTRM. La Figura 4.4 mostra il ciclo di esecuzione e di controllo utilizzato sul nostro framework.
Da notare la differente posizione degli ingressi nel sistema. I primi, a

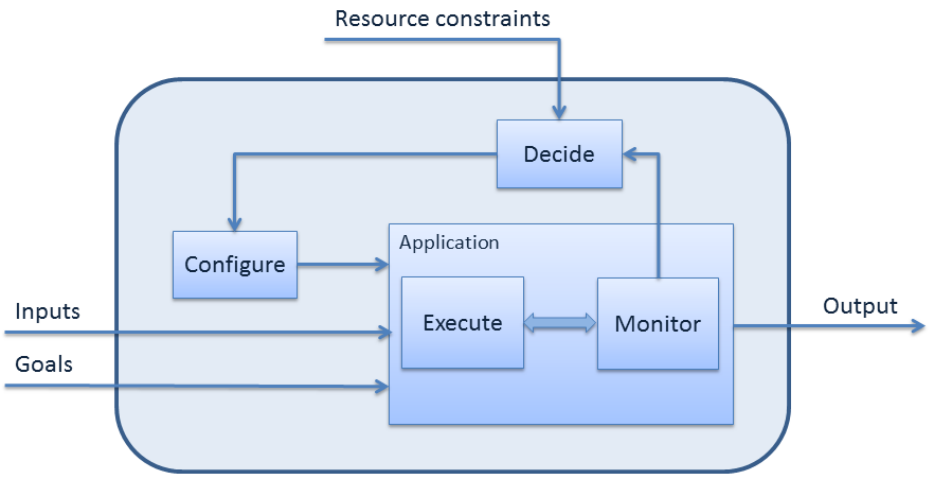


Fig. 4.4: Ciclo d'esecuzione e di controllo di un'applicazione

sinistra, sono forniti dallo sviluppatore mentre i vincoli di risorse, in alto, sono forniti direttamente dal Resource Manager.

4.3 Conclusioni

In questo capitolo è stata introdotta la strategia di sviluppo per il framework ARGO da noi implementato. Abbiamo descritto le sue caratteristiche principali, l'architettura di riferimento e le idee che ci hanno portato a definire questo framework. Nel prossimo capitolo verranno introdotti i monitor, una parte fondamentale del nostro lavoro di tesi.

Capitolo 5

Architettura di monitoring

In questo capitolo ci focalizzeremo su una delle parti centrali di questo lavoro di tesi, i monitor. Nonostante le validissime alternative già presenti si è deciso di implementare una suite di monitoring per spostare verso l'alto il punto di vista comunemente adottato. Vedremo inoltre perché sono stati implementati dei nuovi monitor e le scelte progettuali adottate.

Nelle suite di monitoring analizzate si è evidenziata una tendenza comune a focalizzare l'attenzione sul dato monitorato, ad esempio il risultato prodotto da un timer. Quello che è stata fatto con l'implementazione proposta è stato astrarre il punto di vista dal dato al goal, come illustrato nella Figura 5.1. Seguendo questa idea, nel caso si voglia monitorare un intervallo di tempo, non verrà più restituito un dato rappresentante un intervallo temporale ma un valore che mette in relazione il risultato ottenuto con il goal precedentemente definito. Precisamente, verrà restituito un booleano che indicherà se il goal è stato rispettato o meno ed un valore rappresentante l'errore relativo commesso. Dettaglieremo nel corso del capitolo il suo significato. Questo approccio permette di ridurre al minimo i compiti del programmatore in quanto tutte le elaborazioni sui dati acquisiti sono configurabili ed effettuate a livello monitor, non più a livello applicazione.

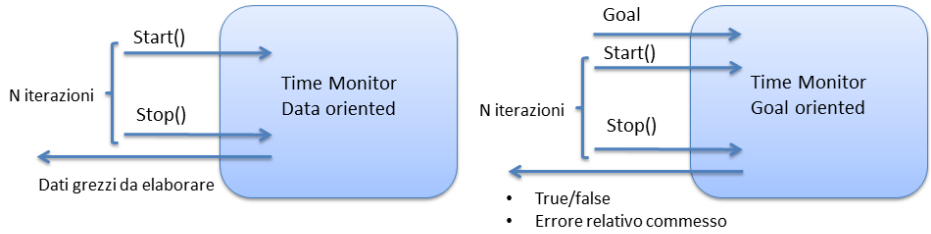


Fig. 5.1: Monitor orientati ai dati e monitor orientati ai goal

I monitor offrono diverse funzionalità predefinite come la possibilità di effettuare le più comuni operazioni matematiche sui dati acquisiti e di archiviare gli stessi attraverso l'uso di un buffer di dimensione configurabile, contenente un archivio di dati più o meno ampio secondo le esigenze del programmatore. Nel resto di questo scritto il buffer sopracitato verrà identificato anche come finestra.

La suite sviluppata include un monitor generico che può essere utilizzato con diversi tipi di dati. Attraverso gli accorgimenti adottati in fase di progettazione, è possibile estendere facilmente le sue funzionalità predefinite o implementare monitor ad hoc. Seguendo questa strategia sono state implementate delle estensioni per supportare gli utilizzi più comuni: monitor di tempo, throughput e memoria. Questi monitor sono stati creati per essere impiegati sia a design-time che a run-time. A run-time possono essere utilizzati per monitorare lo stato di un'applicazione rispetto ai suoi goal mentre in fase di design possono essere impiegati per profilare correttamente l'applicazione.

5.1 Descrizione architettura, scelte progettuali

Occupiamoci ora della descrizione dei dettagli progettuali e dell'architettura dei monitor. Fin dalle prime fasi progettuali è stata posta particolare attenzione riguardo la genericità e l'estendibilità dei monitor; uno degli

obiettivi chiave era creare una suite che potesse essere utilizzata dal più ampio spettro possibile di programmi. Per raggiungere questo obiettivo è stata creata una struttura modulare con alla base la classe Monitor. Questa classe contiene i metodi e le proprietà necessarie per il corretto funzionamento di un monitor nell'ambito di questo progetto; inoltre Monitor è stato reso estendibile per permettere le creazione di varie specializzazioni. Oltre alla classe monitor sono state create le tre estensioni sopracitate rese a loro volta estendibili. La Figura 5.2 illustra l'architettura generale adottata per la parte di monitoring.

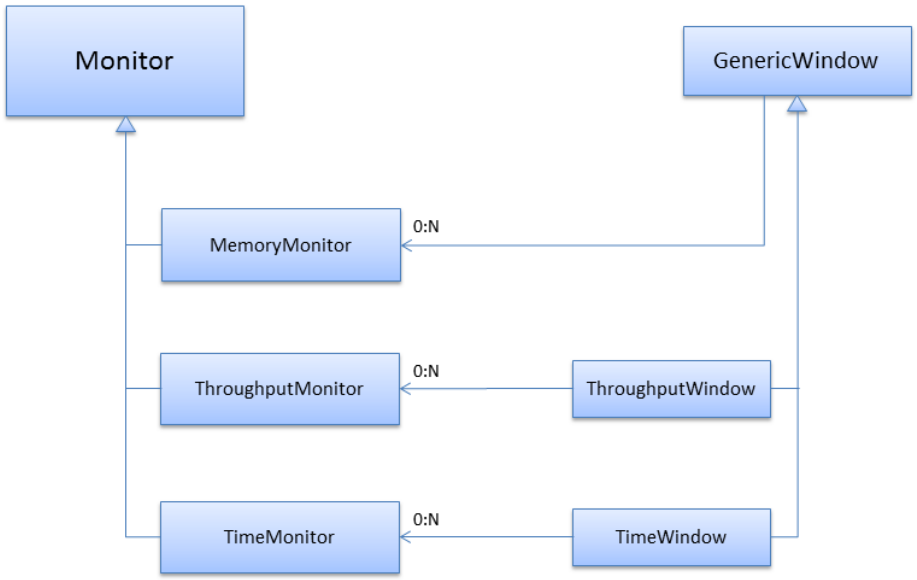


Fig. 5.2: Panoramica dell'architettura del framework di monitoring

Come visibile dal diagramma in Figura 5.2, la classe *Monitor* è stata progettata come classe base comune ad ogni monitor. Essa definisce e implementa le proprietà e le funzionalità necessarie per permettere ad un qualsiasi monitor di funzionare correttamente nell'ambito di questo framework.

Prima di procedere all'analisi della classe è importante notare che ogni monitor permette di definire più goal dello stesso tipo, identificabili con

un numero univoco e utilizzabili continuando a lavorare con un unico oggetto.

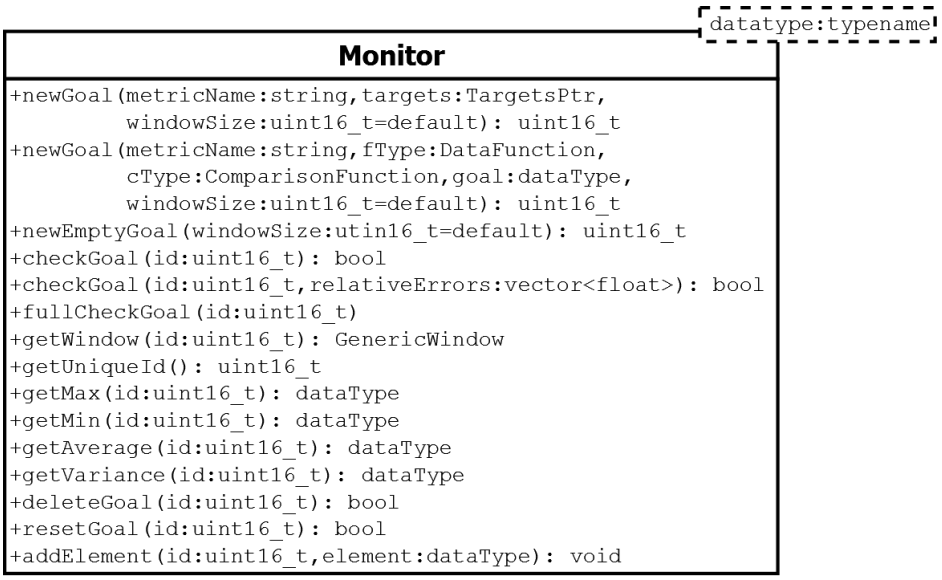


Fig. 5.3: Diagramma UML delle funzioni della classe Monitor

Come è possibile osservare dal diagramma in Figura 5.3, la classe monitor offre diversi metodi per definire goal attraverso varie implementazioni della funzione *newGoal*. In questa funzione come primo parametro troviamo una stringa, chiamata *metricName*, che contiene un identificativo mnemonico del goal che si va a definire. Questa stringa è il nome della metrica del sistema associata al goal. Di seguito è presente un'opzione che prevede la possibilità di definire la modalità di valutazione di un goal in maniera user friendly attraverso le enumerazioni *DataFunction* e *ComparisonFunction*. La prima enumerazione *DataFunction* contiene i seguenti elementi:

- max: richiede un goal valutato sul massimo della finestra di dati.
- min: richiede un goal valutato sul minimo della finestra di dati.
- average: richiede un goal valutato sulla media della finestra di dati.

- variance: richiede un goal valutato sulla varianza della finestra di dati.

Mentre l'enumerazione *ComparisonFunction* contiene:

- greater: richiede un goal maggiore del valore dato.
- greaterOrEqual: richiede un goal maggiore o uguale del valore dato.
- less: richiede un goal minore del valore dato.
- lessOrEqual: richiede un goal minore o uguale del valore dato.

Questa coppia di enumerazioni permette di definire la valutazione di un goal; è possibile dichiarare un limite inferiore, identificato dal parametro *goal*, sulla media dei valori acquisiti con la seguente combinazione di parametri: *DataFunction::Average*, *ComparisonFunction:Greater*, *goal*. Con l'ausilio di questa strategia il programmatore potrà esprimersi in maniera facilitata senza dover conoscere particolari convenzioni. L'esempio successivo mostra come dichiarare un goal legato alla metrica *Frame/sec*. In questo caso un solo obiettivo è stato dichiarato: una media sul throughput maggiore di 25 frame al secondo.

```
ThroughputMonitor t;
t.newGoal("Frame/sec", DataFunctions::Average,
          ComparisonFunctions::Greater, 25);
```

Tornando alla funzione *newGoal* andrà inoltre parametrizzata la dimensione del buffer nel quale acquisire i dati storici con il parametro *windowSize*.

Oltre alla modalità sopracitata è possibile definire un goal multi-obiettivo. Questo tipo di definizione è resa possibile aggregando i valori *DataFunction*, *ComparisonFunction*, *goal* in una classe di tipo *Target*. L'insieme di questi Target rappresenta quindi gli obiettivi del goal come rappresentato in Figura 5.4.

Come ultima modalità la classe offre la possibilità di definire un monitor senza goal continuando però a sfruttare la finestra di valori. Questa

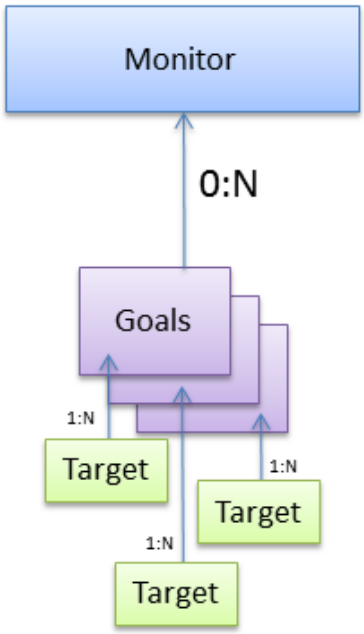


Fig. 5.4: Monitor e goal multiobiettivo

modalità permette di avere a disposizione un archivio di dati nel caso in cui non sia necessario dichiarare un goal ma si vogliano comunque sfruttare le funzionalità statistiche fornite.

Speculari ai metodi per la definizione dei goal abbiamo i metodi *checkGoal* per la loro verifica. Entrambi questi metodi richiedono in input l'identificativo del goal. La prima versione non richiede altro in quanto restituisce solo un indicazione generica attraverso un valore booleano mentre la seconda fornisce anche i diversi errori relativi.

Per una più completa gestione dei goal multi-obiettivo è stata implementata una funzione più avanzata, chiamata *fullCheckGoal*, che e a differenza delle precedenti funzioni restituisce un puntatore ad un oggetto *GoalInfo*. Questo oggetto contiene tutte le informazioni reperibili riguardanti gli obiettivi. Precisamente contiene il nome della metrica *metricName* e una serie di vettori contenenti un valore per ogni obiettivo: *achieved* contenente una serie di booleani che offrono un riscontro immediato sul

superamento del singolo obiettivo, *targetGoals* contenente i diversi valori degli obiettivi, *observedValues* contenente i valori osservati, *relativeErrors* contenente gli errori relativi, *naps* contenente i NAP.

L'errore relativo e il NAP (Normalised Actual Penalty) indicano in modo differente la distanza dal goal. L'errore relativo indica lo scostamento in percentuale dal goal sia in positivo che in negativo. Mentre questa è una metrica facilmente comprensibile, il NAP è un indicatore dalla formulazione più complessa adatto al resource manager. Più precisamente questo valore rappresenta l'elaborazione mancante necessaria al raggiungimento del goal ed è calcolato con la seguente formula $NAP = \mid (x - G)/(x + G) \mid$, dove x è il valore misurato, ad esempio la media dei valori della finestra, e G è il goal. In caso di superamento del goal il NAP vale zero. Questa metrica è stata introdotta per interfacciarsi efficacemente con il framework di system run-time resource manager, BOSP, utilizzato nel lavoro.

Continuando l'analisi della classe monitor vediamo che i valori acquisiti, memorizzati in finestre separate per ogni goal, possono essere valutati tramite i seguenti metodi: *getMin*, *getMax*, *getAverage*, *getVariance*.

Sono state predisposte due funzionalità aggiuntive per la gestione dei goal: la funzione *resetGoal* e la funzione *deleteGoal*. La prima si occupa di cancellare i valori presenti nella finestra mentre la seconda cancella dalla memoria tutti i dati riguardanti il goal.

L'ultima funzione presente in questa classe è *addElement*. Essa si occupa di aggiungere un elemento all'interno del monitor contrassegnato dall'id passato come parametro alla funzione. Nel corso di questo capitolo si è parlato di una finestra nella quale archiviare uno storico dei valori acquisiti. Anche questa funzionalità è stata implementata all'interno dei monitor attraverso la classe *GenericWindow*.

La classe *GenericWindow*, il cui diagramma è presente in Figura 5.5, è istanziata singolarmente per ogni goal. Si occupa della gestione dei valori acquisiti e del confronto con gli obiettivi parametrizzati in fase di inizializzazione della classe *Monitor*. Al suo interno sono presenti tanti oggetti

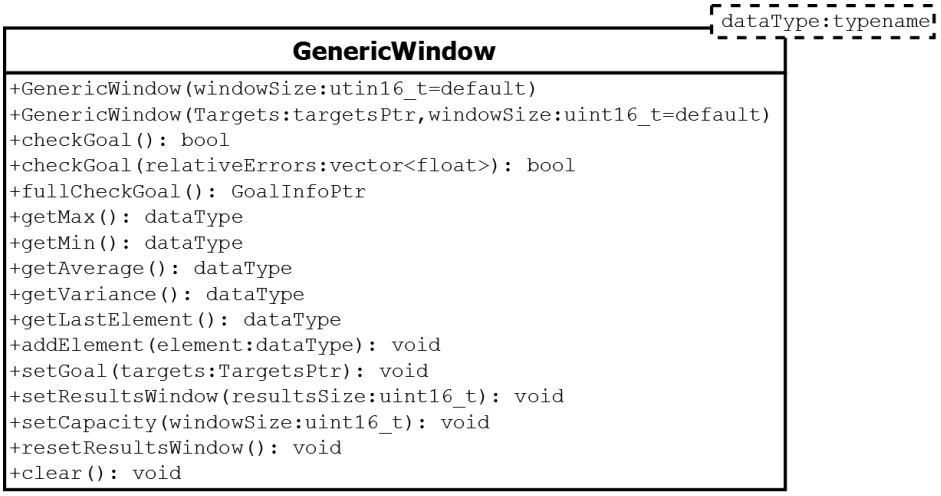


Fig. 5.5: Diagramma UML delle funzioni della classe GenericWindow

Target quanti sono gli obiettivi definiti nel goal. Inoltre sono presenti i valori parametrizzati nella funzione *newGoal* e tutte quelle funzioni che svolgono operazioni sui dati acquisti come i metodi statistici precedentemente citati.Questa classe è dotata di un buffer circolare, contenente i dati acquisiti, che al raggiungimento del numero prefissato di elementi sovrascrive i più vecchi. La Figura 5.6 rappresenta la finestra di valori appena citata.

Sempre nella classe *GenericWindow* troviamo molti altri metodi necessari ad una gestione completa della finestra di valori. Si trovano vari metodi set:

- *setGoal()* per l'impostazione del goal sui dati acquisiti.
- *setCapacity()* per configurare la dimensione fisica del buffer circolare.
- *setResultsWindow()* per modificare il numero di valori sul quale effettuare le operazioni richieste; questo valore può essere diverso dalla capacità effettiva.

Come ultime funzioni troviamo *resetResultsWindow()* per resettare la finestra di valori e *clear()* per cancellare i dati in memoria riguardanti la finestra corrente.

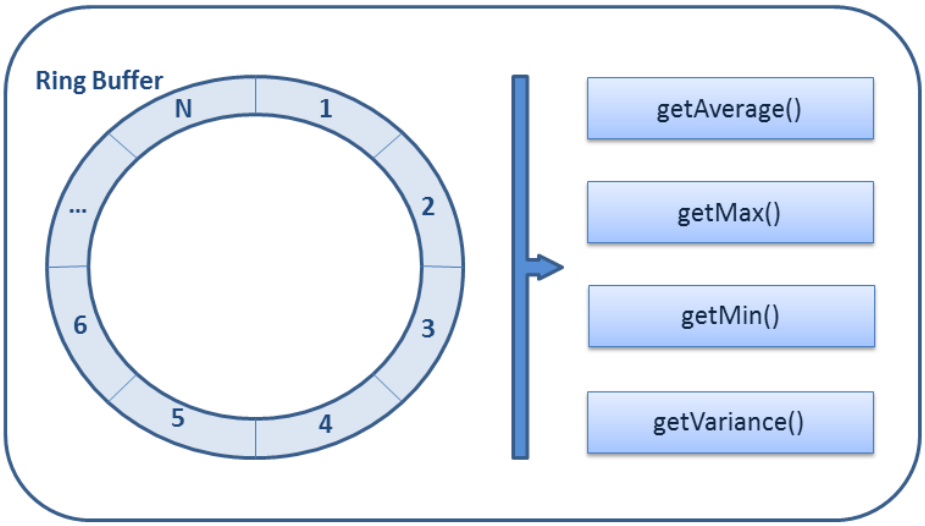


Fig. 5.6: Finestra dei dati e sue funzioni aggregate

5.1.1 Time monitor

Dopo aver introdotto le classi che costituiscono la base di tutti i monitor possiamo analizzare le sue implementazioni specifiche.

La classe *TimeMonitor* a cui si riferisce la Figura 5.7 rappresenta i monitor più comunemente utilizzati in quanto è ricorrente l'esigenza di acquisire tempistiche durante l'esecuzione di un algoritmo. Questi monitor, insieme ai throughput monitor illustrati successivamente, fanno concettualmente parte dei monitor orientati all'applicazione come illustrato nel Capitolo 2. Rispetto alla classe Monitor precedentemente introdotta abbiamo ridefinito le funzioni newGoal e aggiunto una versione semplificata quest'ultima. Data la necessità di effettuare misure di tempo è stato necessario ridefinire le funzioni *newGoal* per inizializzare alcune variabili aggiuntive. Inoltre, vista la frequente esigenza di dichiarare un goal come limite superiore della media dei valori acquisiti, abbiamo introdotto una nuova funzione *newGoal* per semplificare la dichiarazione di un goal di questa tipologia.

Oltre a queste funzioni ne sono state aggiunte altre per la gestione delle

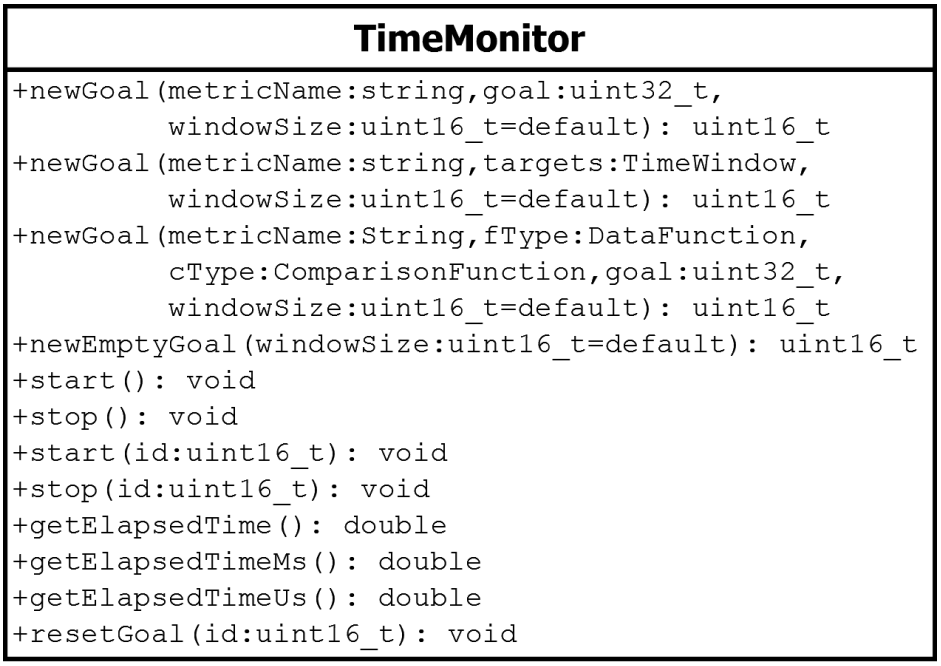


Fig. 5.7: Diagramma UML delle funzioni della classe TimeMonitor

misurazioni del tempo. Sono state aggiunte due coppie di funzioni *start* e *stop* le quali si occupano della misurazione del tempo intercorso tra due successive chiamate con la massima trasparenza per il programmatore. La prima coppia richiede il parametro l'id del goal al quale riferirsi per archiviare i valori nella finestra di dati corretta. Invece attraverso le funzioni *start* e *stop* senza parametri è possibile utilizzare un timer senza alcuna funzionalità avanzata ma al contempo senza dover conoscere le chiamate ad una libreria specifica e utilizzando sempre lo stesso oggetto. Una volta misurato l'intervallo di tempo desiderato è possibile richiedere il valore in diverse unità di misure con le funzioni: *getElapsedTime()*, *getElapsedTimeMs()*, *getElapsedTimeUs()*, che restituiranno il valore misurato in secondi, millisecondi o microsecondi.

5.1.2 Throughput monitor

I throughput monitor rappresentano un evoluzione concettuale dei time monitor; non introducono nessuna novità specifica ma richiedono alcune piccole modifiche. Nella Figura 5.8 viene illustrato il diagramma della classe a cui si fa riferimento.

ThroughputMonitor

```
+newGoal(metricName:string,goal:uint32_t,
         windowSize:uint16_t=default): uint16_t
+newGoal(metricName:string,targets:TimeWindow,
         windowSize:uint16_t=default): uint16_t
+newGoal(metricName:string,fType:DataFunction,
         cType:ComparisonFunction,goal:uint32_t,
         windowSize:uint16_t=default): uint16_t
+newEmptyGoal(windowSize:uint16_t=default): uint16_t
+start(): void
+stop(data:double): void
+start(id:uint16_t): void
+stop(id:uint16_t,data:double): void
+resetGoal(id:uint16_t): void
```

Fig. 5.8: Diagramma UML delle funzioni della classe ThroughputMonitor

Questi monitor non introducono modifiche sostanziali in quanto il calcolo del throughput richiede sempre una misura di tempo; è necessaria in aggiunta la conoscenza dei dati analizzati nel frangente di tempo misurato.

Per questa esigenza le funzioni *stop* richiederanno come parametro la quantità di dati analizzati così da poter passare dalla misura di un tempo alla misura di un throughput. Le modalità di funzionamento sono analoghe a quelle illustrate nella classe *TimeMonitor*.

5.1.3 Memory monitor

Differentemente dalle classi *TimeMonitor* e *ThroughputMonitor*, la classe *MemoryMonitor* appartiene logicamente alla tipologia di monitor orientata

alle risorse. L'implementazione del *MemoryMonitor* di cui possiamo vedere il grafico in Figura 5.9 è differenziata rispetto alle altre tipologie appena viste in quanto il valore di memoria restituito fa riferimento sempre all'occupazione istantanea.

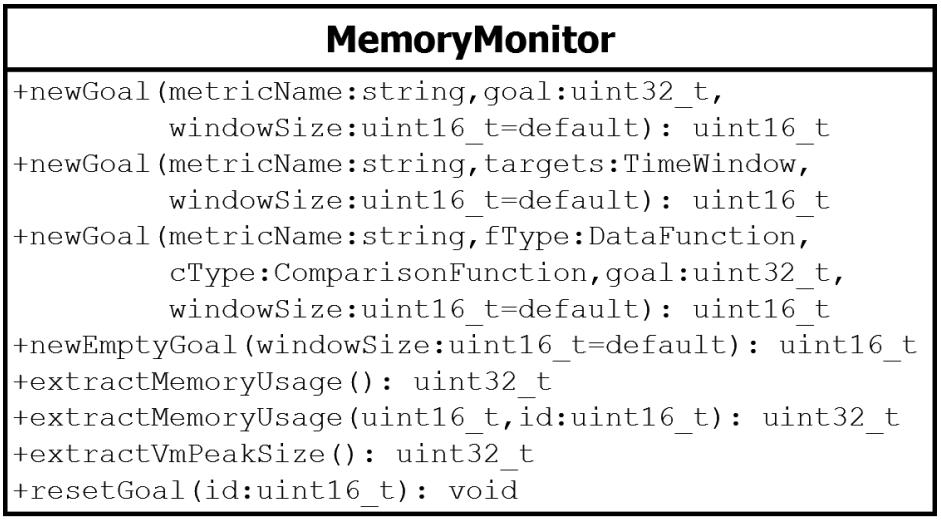


Fig. 5.9: Diagramma UML delle funzioni della classe MemoryMonitor

In questa tipologia di monitor si ha la possibilità di istanziare un nuovo goal attraverso la consueta funzione *newGoal* sia con i soli parametri *metricName*, *goal* sia con la combinazione *metricName*, *goal* e *windowSize*. Per ottenere l'attuale occupazione di memoria è possibile utilizzare la funzione *extractMemoryUsage* senza parametri oppure la stessa funzione con il parametro *id*. La differenza tra queste due funzioni, analoga a quella tra le coppie di metodi start e stop, è l'utilizzo o meno della finestra di dati.

In questa classe è inoltre presente la funzione *extractVmPeakSize* che restituisce la quantità massima di memoria virtuale utilizzata. Questa funzione risulta particolarmente utile durante la profilazione di un'applicazione in quanto la memoria può essere una risorsa critica. È fondamentale per il run-time manager, in particolare per la sua componente resour-

ce manager, conoscere la massima quantità di memoria utilizzata dal programma.

5.1.4 Altri monitor

Oltre ai monitor già illustrati in questo capitolo sono stati studiati, ed in parte implementati, altri tipi di monitor. Questi fanno parte sia del gruppo di monitor orientati all'applicazione, sia di quelli orientati alle risorse o al sistema. L'architettura di questi monitor è già stata illustrata nel Capitolo 4 e nell'immagine 4.2. Come già spiegato precedentemente in questo lavoro di tesi, i monitor orientati alle risorse o al sistema si occupano tipicamente di: input/output, traffico di rete, prestazioni CPU e memoria RAM, accessi ai dischi, controllo termico.

Monitor di queste tipologie, sono stati previsti dalla nostra architettura ma non sono stati interamente implementati in quanto non previsti dal framework Barbeque. L'unico monitor di questo tipo che è stati implementato è il MemoryMonitor. Il CPUMonitor è stato soltanto implementato parzialmente tramite le funzionalità fornite dal Resource Manager utilizzato nel nostro framework. Questi due monitor sono fondamentali per la fase di profiling delle applicazioni durante la fase di progettazione e sono stati utlizzati estensivamente nel corso dei casi d'uso analizzati nel Capitolo 8. Seguendo questo stesso approccio sono stati utilizzati un *QualityMonitor* e un *ErrorMonitor* per tracciare l'accuratezza e l'entità degli errori d'elaborazione di alcune applicazioni studiate nei casi d'uso. Per questi monitor non è stato necessario implementare nessuna funzionalità aggiuntiva rispetto alla classe base Monitor. La sua versatilità ci ha permesso di usarla per gli scopi più disparati.

5.2 Conclusioni

In questo capitolo abbiamo introdotto l'architettura di monitoring a supporto del framework da noi creato. Essa è una componente fondamentale,

sia a design-time che a run-time, del framework ARGO. Sono state illustrate le varie motivazioni che ci hanno condotto alle scelte effettuate ed è stato descritto approfonditamente l'uso delle varie classi dell'architettura. Nel prossimo capitolo invece ci si focalizzerà maggiormente sulla fase di design, alla quale i monitor forniscono supporto per il profiling dell'applicazione.

Capitolo 6

Tecniche a Design-Time

Una dei passaggi chiave di questo lavoro di tesi è caratterizzato dall'intensivo uso di tecniche a design-time al fine di ottenere una caratterizzazione dell'applicazione e di tutti i suoi parametri variabili a run-time. Stabilire le relazioni tra i vari parametri di un'applicazione e le prestazioni dell'applicazione stessa è estremamente importante per sfruttare al meglio la piattaforma in cui l'applicazione viene eseguita, ed ottimizzare caratteristiche come ad esempio le prestazioni o il consumo di potenza. Valutazioni basate sull'esperienza non sono sempre possibili sia perché le combinazioni di parametri potrebbero essere elevate, sia perché non è sempre vero che una soluzione considerata ragionevole sia quella esatta. Per avvalorare questa considerazione vogliamo mostrare i risultati dello studio proposto da Pusukuri et al. riguardo l'esecuzione di applicazioni multithread in sistemi multiprocessore [43]. É un'idea diffusa che si possa raggiungere il massimo delle prestazioni di una applicazione parallela quando si abbiano a disposizione un numero di thread uguale al numero di processori del sistema. Se in alcuni casi queste considerazioni possono essere vere, in altri possono portare a un sotto utilizzo dell'architettura. Pusukuri et al. dimostrano nel loro articolo che in base all'architettura interna dell'applicazione è possibile che siano necessari fino a 63 thread in

una macchina con 24 core per raggiungere il massimo livello di parallelismo. É proprio in questi casi che le tecniche di Design Space Exploration possono rivelarsi fondamentali.

I concetti usati in questa fase si basano sul lavoro sviluppato in [30–33]. La Figura 6.1 presenta un breve riassunto dei passi necessari allo svolgimento corretto della studio dell'applicazione e la generazione delle informazioni utili a run-time per la gestione delle varie applicazioni.

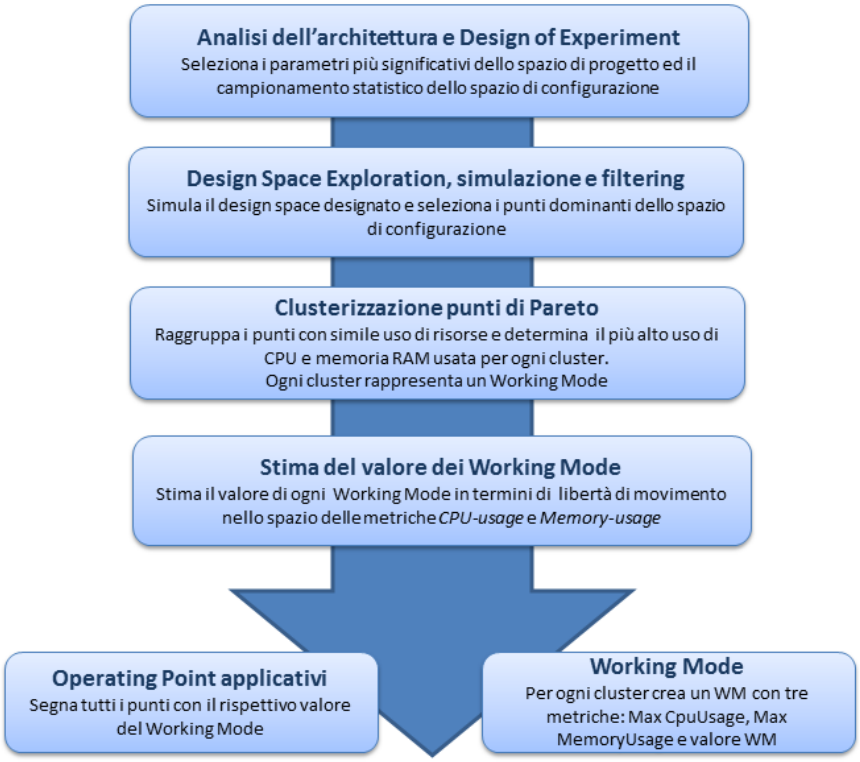


Fig. 6.1: Flusso di sviluppo a design-time

Le tecniche utilizzate sono varie e partono da una preliminare valutazione dei parametri significativi da usare per l'esecuzione delle varie simulazioni e nella 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 di questo capitolo.

Un altro tassello importante per la caratterizzazione delle applicazioni è la fase di Design Space Exploration (DSE), dove viene effettuata l'esecuzione dell'applicazione con tutte le configurazioni di parametri stabilite dalla DoE.

Successivamente, una fase di analisi di questi punti verrà effettuata tramite tecniche di pareto-filtering, clusterizzazione e altre più strettamente specifiche riguardo la generazione di punti per il SYS-RTRM e l'AS-RTM. Anche queste fasi verrano presentate più in dettaglio nelle prossime sezioni di questo capitolo.

6.1 Design of Experiments

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 [33] e in [44], 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 [33], consideriamo 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 nostro caso è 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.

6.2 Design Space Exploration

Dopo che una Design of Experiment è stata selezionata, la DSE esegue tutte le simulazioni necessarie. Questa fase consiste in un'iniziale esecuzione dell'applicazione con tutte le possibili combinazioni di parametri definite dal design space. La configurazione dei parametri verrà effettuata in automatico dal framework d'esplorazione, permettendo di eseguire tutte le simulazioni con il minor overhead possibile. Strategie più avanzate

come quelle proposte in ReSPIR [33] sono supportate dal framework e permettono di effettuare un'approssimazione iterativa del comportamento del sistema tramite tecniche di Response Surface Methodology (RSM). Queste simulazioni forniscono in uscita un insieme di punti operativi che vanno a formare lo spazio di configurazione del sistema che consiste in un insieme di valori rappresentati dai differenti parametri dell'applicazione insieme ad alcune metriche estratte dalla fase di simulazione. I parametri sono i valori attraverso i quali è possibile influenzare l'esecuzione dell'applicazione andando a variare il funzionamento dell'applicazione stessa. Queste modifiche generalmente influenzeranno diversi fattori come le performance o l'accuratezza. Le metriche rappresentano invece una stima di una caratteristica o di una performance dell'applicazione in relazione ai valori dei parametri ad essa applicati. Alcuni tipici esempi di metrica sono l'accuratezza dei risultati, il throughput, la latenza o il consumo di potenza.

I risultati forniti dalla fase di Design Space Exploration forniscono dei punti che sono ancora lontani dall'essere considerati come risultato finale. Il numero di punti risultante da questa fase può essere molto elevato poiché possono essere generati centinaia o migliaia di punti operativi a seconda della dimensione del design space e del campionamento dei punti selezionati per la DSE; questi punti non sono tutti Pareto-ottimali e quindi hanno bisogno di essere filtrati. Questo è generalmente un problema di ottimizzazione multi-obiettivo e non conduce a una soluzione unica poiché i punti risultanti presentano un ordinamento parziale dato da una relazione di dominanza. Come già illustrato nel Capitolo 2, in un problema di minimizzazione un punto dello spazio di configurazione, detto set di configurazione, si definisce come dominato da un altro punto se ha lo stesso valore per tutti le sue quantità tranne una in cui il suo valore risulta peggiore dell'altro punto [6]. La relazione di dominanza tra due punti, rappresenta l'interesse nella scelta di un punto rispetto ad un altro quando esso fornisce un miglioramento di una delle quantità, senza peggiorarne

nessun altra. Vice versa, non è utile selezionare quel punto se peggiora il valore di un parametro senza fornire alcun miglioramento rispetto all'altro punto. Come già detto in precedenza, questa fase di filtraggio svolta grazie all'analisi di Pareto, consiste nella minimizzazione (o massimizzazione) di un insieme di funzioni obiettivo, ad esempio potenza, precisione, latenza e throughput. Quando la soluzione di questo problema è stata raggiunta, il set di punti risultante è costituito soltanto da punti utili.

6.3 Analisi dei punti operativi

Dopo le fasi eseguite nella sezione precedente, viene effettuata un'ulteriore analisi dei punti. Questa seconda analisi è composta dalla clusterizzazione dei punti in base agli usi delle risorse. Dato che nel caso del Resource Manager utilizzato le uniche risorse controllate sono l'utilizzo della CPU e della memoria RAM, la clusterizzazione avverrà soltanto su queste due metriche. Utilizzando una tecnica di K-means clusterisation [45] è possibile ottenere una prima classificazione di punti che verrà utilizzata per determinare i Working Mode e il loro valore. Il numero di cluster, quindi di Working Mode, 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. É stato determinato empiricamente che un numero di Working Mode compreso tra 3 e 10 è ottimale e può essere gestito dal System-level Run-Time Resource manager senza problemi.

Terminata la fase di clustering viene determinato il valore da assegnare a ciascun Working Mode. L'algoritmo 1 mostra uno pseudocodice dell'algoritmo di stima di questi valori. Il primo passo consiste nel trovare il massimo e il minimo utilizzo della CPU e della memoria RAM di tutto l'insieme dei punti clusterizzati. Successivamente, viene determinata la differenza, chiamata *swing* o intervallo di massimizzazione, tra il valore

massimo e minimo per ciascuno di questi parametri (righe 2–7).

Algoritmo 1: Calcolo del valore dei Working Mode

Input:
- Lista di Operating Point ($opList$) etichettati con il loro cluster-ID
- Lista di cluster-ID ($clusterIDs$)
- Peso della metrica cpuUsage (W_{cpu})
- Peso della metrica memUsage (W_{mem})

Output:
- Lista dei valori di ogni Working Mode ($wmValues$)

1 **begin**
2 min_{cpu} = Min(cpuUsage, $opList$)
3 max_{cpu} = Max(cpuUsage, $opList$)
4 min_{mem} = Min(memUsage, $opList$)
5 max_{mem} = Max(memUsage, $opList$)
6 $swing_{cpu} = max_{cpu} - min_{cpu}$
7 $swing_{mem} = max_{mem} - min_{mem}$
8 **foreach** *id in clusterIDs* **do**
9 $clusterMax_{cpu}$ = Max(cpuUsage, $opList$, id)
10 $clusterMax_{mem}$ = Max(memUsage, $opList$, id)
11 $clusterSwing_{cpu} = clusterMax_{cpu} - min_{cpu}$
12 $clusterSwing_{mem} = clusterMax_{mem} - min_{mem}$
13 $value_{cpu} = W_{cpu} * (clusterSwing_{cpu} / swing_{cpu})$
14 $value_{mem} = W_{mem} * (clusterSwing_{mem} / swing_{mem})$
15 $wmValues[id] = 100 * (value_{cpu} + value_{mem}) / (W_{cpu} + W_{mem})$
16 **end**
17 **end**

Dato che ogni applicazione potrà cambiare il suo Operating Point corrente, quindi l'utilizzo della CPU e/o della memoria RAM, modificando i parametri di configurazione interni, questi valori esprimeranno la libertà di movimento disponibile all'interno dello spazio di configurazione dato

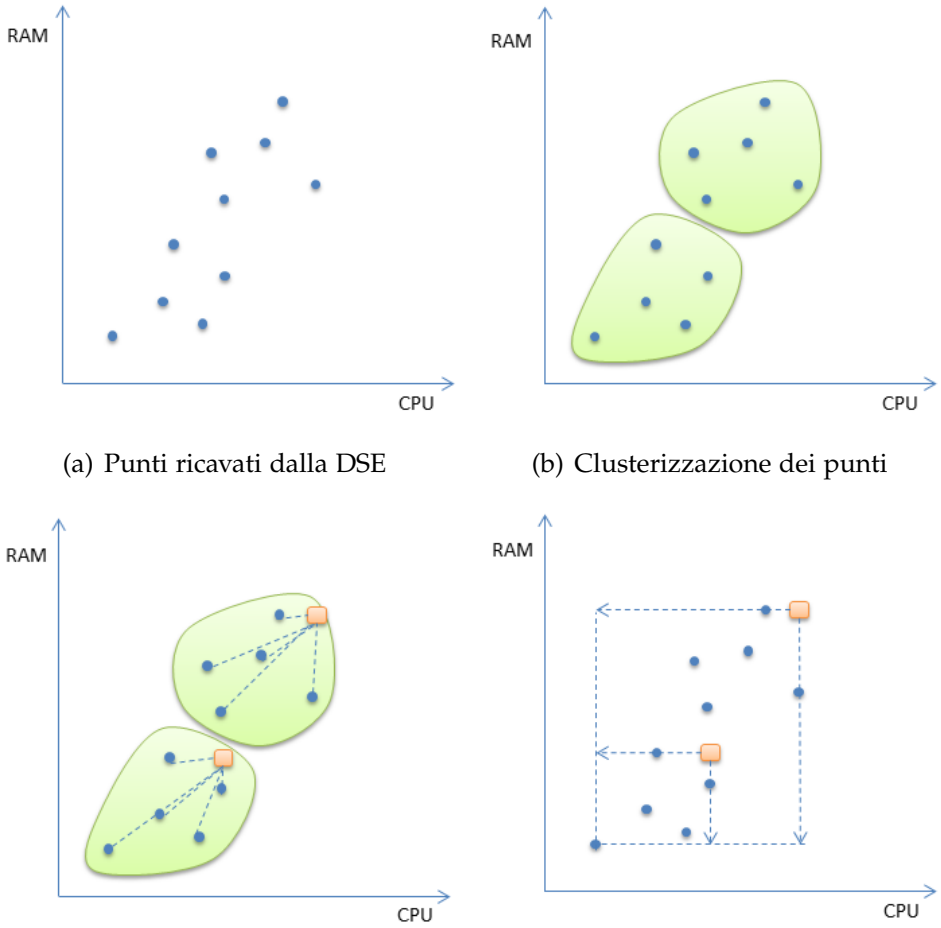
dalle metriche rappresentanti l'utilizzo della CPU e della memoria RAM.

Dopo questo passo, per ogni cluster viene effettuato un insieme di azioni. Innanzitutto viene determinato l'utilizzo massimo della CPU e della memoria associato ai punti appartenenti al cluster (righe 9–10) per poi calcolarne la sua oscillazione rispetto all'uso di risorse (righe 11–12). Questa non è un'oscillazione interna al cluster, ma rappresenta i valori ammessi della CPU e della memoria con risorse minori o uguali a quelle del cluster. Con questo concetto vogliamo affermare che se un Working Mode viene scelto, nulla vieta all'applicazione di andare in un punto di un altro cluster a meno che non provenga da un cluster con un utilizzo di risorse superiore.

Un'altra cosa da notare è che, siccome ogni punto di un cluster verrà rappresentato da un solo punto nella lista dei Working Mode, un Operating Point condividerà con gli altri punti del cluster la medesima oscillazione, quindi anche il valore del Working Mode. La Figura 6.2 rappresenta la fase di clusterizzazione dei punti e di creazione dei Working Mode. In Figura 6.2-a è possibile osservare l'insieme dei punti ricavati durante le fasi di Design Space Exporation e Pareto-filtering. La Figura 6.2-b mostra invece due possibili cluster di punti generati dalla fase di K-means clustering. Infine, in Figura 6.2-c, è possibile vedere come venga creato un nuovo punto per ogni cluster che ha come valori associati il massimo utilizzo di risorse tra i punti del cluster. Questo punto sarà il Working Mode associato al cluster. La figura 6.2-d visualizza lo swing dei working mode per ogniuna delle risorse.

Il penultimo passo dell'algoritmo (righe 13–15), consiste nel calcolo del valore effettivo di ogni Working Mode. Il rapporto tra lo swing del cluster e quella totale per ogni risorsa fornisce ciò che viene chiamato Cpu-Value e Memory-Value. Questi valori verranno poi ponderati in base alle preferenze dello sviluppatore per formare una metrica unica che rappresenta il valore di un Working Mode.

A questo punto la fase di progettazione è terminata. L'ultimo passo



(a) Punti ricavati dalla DSE

(b) Clusterizzazione dei punti

(c) Creazione dei Working Mode

(d) Calcolo swing dei Working Mode

Fig. 6.2: Fase di clusterizzazione e creazione dei Working Mode

consiste nella creazione della lista di Operating Point per l'applicazione e l'elenco dei Working Mode per il resource manager. Nella lista di punti operativi vi saranno sia i parametri di configurazione sia le metriche ricavate. Allo stesso tempo, l'elenco dei Working Mode conterrà il valore di ognuno di essi, assieme all'utilizzo massimo della CPU e della memoria RAM per il cluster associato.

Ogni qual volta il Resource Manager selezionerà un Working Mode per un'applicazione, l'Application-Specific Run-Time Manager e il relativo Operating Points Manager selezionerà il punto operativo corretto a secon-

da del valore WM e dei vincoli sulle risorse fornite dal Working Mode.

6.4 Conclusioni

Questo capitolo ha descritto approfonditamente le tecniche utilizzate all'interno di ARGO che permettono di caratterizzare le applicazioni ed usare queste informazioni durante la fase d'esecuzione. La strategia adottata permette di dividere le informazioni in due diverse classi ed associarle alle componenti decisionali che hanno più informazioni rispetto al problema da risolvere. In questo caso i Working Mode verranno usati dal SYS-RTRM mentre gli Operating Point dall'AS-RTM. Nei prossimi due capitoli verrà illustrato dettagliatamente l'uso che viene fatto di queste informazioni, descrivendo le funzionalità dell'Operating Points Manager e dell'Application-Specific Run-Time Manager.

Capitolo 7

Tecniche a Run-Time

Dopo aver visto nel Capitolo 6 quali sono le tecniche a design-time usate nel nostro framework, questo capitolo illustrerà invece i vari componenti responsabili per una corretta esecuzione delle applicazioni all'interno di ARGO. Le informazioni generate dalla fase a design-time, le cui tecniche sono state illustrate precedentemente, sono fondamentali per la fase successiva della gestione delle applicazioni poiché permettono di rispondere in maniera efficace e con basso overhead ai cambiamenti del sistema o dell'applicazione stessa. In questo capitolo verranno descritte le due classi responsabili, assieme a quella dei monitor descritta nel Capitolo 5, per l'adattamento dinamico dell'applicazione: OP_Manager e ApplicationRTM. Oltre alla descrizione di queste due classi verrà illustrato interamente il flusso d'esecuzione di un'applicazione all'interno del framework ARGO, permettendo di avere una visione completa di come viene svolta la fase di controllo di queste applicazioni da parte dell'Application-Specific Run-Time Manager.

7.1 Operating Points Manager

L'Operating Points Manager (OP Manager) è la componente di ARGO responsabile per la gestione della lista degli punti operativi di un'applicazione. Questa componente usa i dati raccolti a design-time per creare un struttura generica e versatile usata per immagazzinare tutte le informazioni riguardanti gli Operating Point.

Una delle funzionalità più importanti dell'OP Manager è l'abilità di dichiarare una priorità tra le metriche permettendo di effettuarne un ordinamento in base alle preferenze dello sviluppatore. É quindi possibile dichiarare ad esempio massima priorità a configurazioni di punti con basso uso di potenza e lasciare in secondo piano il throughput in casi in cui il dispositivo che esegue l'applicazione abbia poca energia disponibile. Una volta il dispositivo viene messo in carica sarebbe poi possibile tornare ad una politica ad alte prestazioni.

L'Operating Points Manager è in grado di selezionare il miglior punto operativo, in accordo con la priorità data, che soddisfi tutti i vincoli ricevuti in ingresso.

Ad esempio, considerando un'applicazione power-aware con dei vincoli sulla qualità del servizio in relazione alla latenza, l'Operating Points Manager selezionerà il punto operativo con il minor consumo di potenza che, in accordo con l'analisi effettuata a design-time, sia in grado di fornire una latenza inferiore a quella richiesta. Il processo di decisione considererà anche i vincoli sulle risorse notificati dal resource manager, presente a livello di sistema, ogni qual volta le risorse disponibili cambino.

Nella Figura 7.1 è possibile osservare un estratto della classe OP_Manager rappresentante il gestore dei punti operativi utilizzato nel nostro framework. Questa classe provvede alla memorizzazione e alla gestione degli Operating Point permettendo di effettuare un ordinamento secondo varie metriche con priorità diverse. Una volta ordinato l'elenco secondo la metrica richiesta, è possibile spostarsi tra i vari punti e filtrarli secondo le

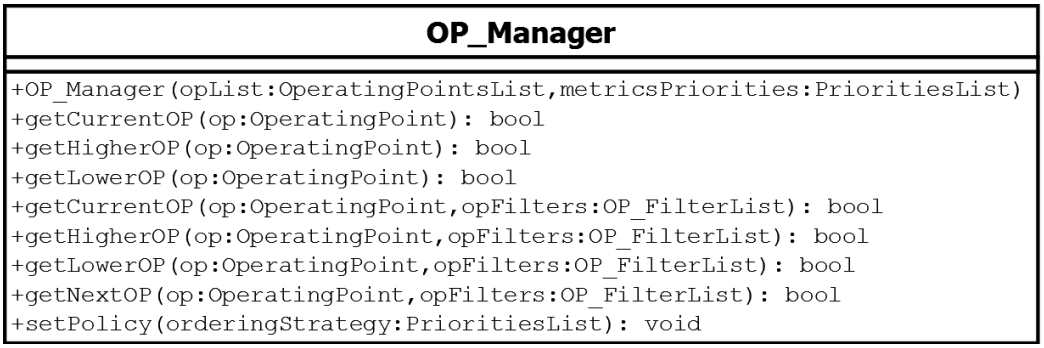


Fig. 7.1: Diagramma UML delle funzioni della classe OP_Manager

richieste dell'utente.

Procedendo nell'analisi dei metodi della classe, è possibile riconoscere il costruttore della classe *OP_Manager* il quale richiede come parametri la lista degli Operating Point *opList* e la lista delle priorità secondo le quali effettuare l'ordinamento, definita dal parametro *metricsPriorities*.

Oltre al costruttore è possibile notare le funzioni per la gestione dei punti operativi. Queste funzioni possono essere divise in due gruppi. Il primo gruppo è composto dalle funzioni: *getCurrentOP*, *getHigherOP* e *getLowerOP*. Queste funzioni si occupano rispettivamente di restituire il punto operativo corrente, superiore o inferiore secondo l'ordinamento stabilito. Come parametro, le tre funzioni elencate, richiedono l'Operating Point nel quale salvare il dato richiesto e restituiscono un booleano indicante se la richiesta è andata a buon fine o no. Il secondo gruppo è composto dalle funzioni: *getCurrentOP*, *getHigherOP*, *getLowerOP* e *getNextOP*. Questo gruppo si differenza dal precedente in quanto oltre all'Operating Point dove immagazzinare il risultato richiedono il parametro *opFilters* contenente il filtro da applicare alla lista di punti.

In aggiunta al primo gruppo è anche disponibile la funzione *getNextOP*, una funzione che ha lo scopo di incapsulare la modalità di ricerca di un nuovo punto operativo che soddisfi i vincoli forniti dal parametro *opFilters*. Questa è la funzione principalmente usata dall'AS-RTM per ottenere un nuovo Operating Point. La scelta di creare questa funzione è

stata fatta per permettere di nascondere allo sviluppatore le complessità d'implementazione e permettendo di cambiare in futuro la strategia di ricerca di nuovi punti operativi senza dover cambiare l'interfaccia con lo sviluppatore. Nella corrente implementazione questa funzione effettua una ricerca lineare tra i punti operativi della lista ma sarà oggetto di ulteriori sviluppi futuri con lo scopo di implementare una ricerca di punti operativi più efficiente. Un possibile esempio dell'utilizzo della funzione *getNextOP* è il seguente:

```
OperatingPoint currentOP;
OP_FilterList opFilter;
opFilters.push_back(OP_Filter("nthreads", ComparisonFunctors::LessOrEqual, 4));
opManager.getNextOP(currentOP, opFilters);
```

In questo caso è stata dichiarata una variabile di tipo OperatingPoint, che conterrà il punto operativo da calcolare, ed un vincolo sul numero massimo di thread allocabili dall'applicazione. La chiamata a *getNextOP*, usata con i parametri dichiarati precedentemente, fornirà il punto operativo che soddisfa il vincolo dichiarato, salvandolo all'interno della variabile *currentOP*.

L'ultima funzione della classe è *setPolicy*, la quale si occupa dell'assegnazione della politica con la quale ordinare i punti operativi.

7.2 Application-Specific Run-Time Manager

Situato in cima ai vari blocchi di base descritti in precedenza, si trova l'Application-Specific Run-Time Manager (AS-RTM). Il suo obiettivo è quello di utilizzare le informazioni fornite dai monitor e dall'Operating Points Manager per prendere decisioni autonome riguardanti l'applicazione. L'AS-RTM utilizza le caratteristiche dei monitor per controllare il raggiungimento dei goal di un'applicazione e l'Operating Points Manager per richiedere i punti operativi che soddisfano i vincoli decisi dal SYS-RTRM, dall'AS-RTM stesso e dallo sviluppatore.

Ogni applicazione può decidere se utilizzare o meno le funzionalità dell'AS-RTM. Nel primo caso l'AS-RTM gestirà autonomamente tutti i parametri di applicazione, mentre nel secondo sarà lo sviluppatore a dover gestire manualmente i dati forniti dalle classi Monitor e dall'Operating Points Manager. Tuttavia è possibile anche un approccio ibrido: gli sviluppatori possono contare sull'AS-RTM e nel contempo prendere decisioni autonome se necessario.

L'AS-RTM ha due particolari caratteristiche che lo rendono così utile. La prima è che è in grado di notificare automaticamente ed autonomamente al resource manager quando, date le risorse attuali e vincoli di QoS, l'applicazione non è in grado di trovare un punto operativo utilizzabile. Questa notifica consiste in una descrizione di quanta potenza computazionale aggiuntiva, quindi risorse, è necessaria per raggiungere gli obiettivi desiderati. Con tale notifica il resource manager potrebbe cambiare le sue politiche di allocazione delle risorse al fine di garantire l'allocazione corretta per ogni applicazione.

La seconda caratteristica dell'AS-RTM è la sua adattabilità a discrepanze proporzionali tra il modello fornito e le effettive prestazioni dell'applicazione. Definiamo come discrepanza proporzionale non solo quella relativa a modifiche lineari delle prestazioni ma più in generale a cambiamenti che non influenzano l'ordinamento degli Operating Point del modello. L'AS-RTM adatta i propri vincoli per gestire una variazione proporzionale delle prestazioni causata ad esempio dalla diminuzione della frequenza o l'usura dei componenti della piattaforma.

Di seguito viene proposto un breve esempio di gestione dei punti operativi da parte del AS-RTM. Consideriamo un goal richiedente una metrica avente uno specifico intervallo di valori, ad esempio una latenza massima tra $10ms$ e $20ms$. L'AS-RTM consulta il gestore dei punti operativi chiedendo un punto che rispetti sia i vincoli imposti sia le risorse disponibili. L'Operating Points Manager fornirà un punto che dovrebbe garantire $18ms$ di latenza, ad esempio poiché trai i punti ammissibili è

quello con minor consumo di potenza. A causa di problemi di temperatura sulla piattaforma, la frequenza del sistema viene dimezzata causando una latenza effettiva di 36*ms*.

Questo comportamento anomalo viene osservato dai monitor che notificano il goal mancato. Considerando che le prestazioni sono state dimezzate rispetto alle previsioni, l'AS-RTM richiede un'altro punto operativo con vincoli raddoppiati: latenza massima tra 5*ms* e 10*ms*. L'applicazione continuerà quindi la sua esecuzione nel rispetto degli obiettivi e dei vincoli risorse/potenza. Solo quando la frequenza del sistema aumenterà nuovamente il punto operativo tornerà ad essere quello originale portando il sistema in uno stato simile a quello descritto dal modello profilato.

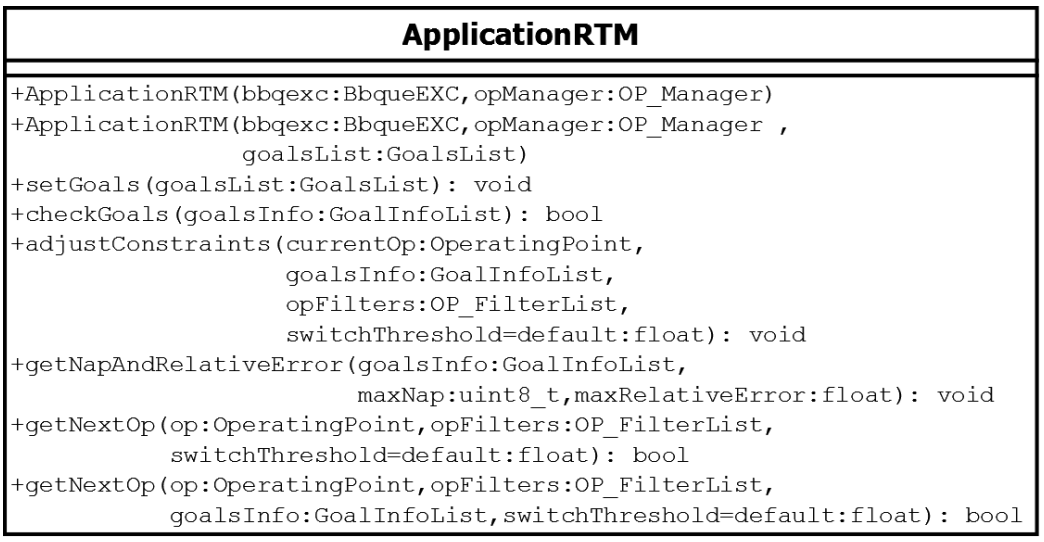


Fig. 7.2: Diagramma UML delle funzioni della classe Application-RTM

In Figura 7.2 è possibile esaminare la classe ApplicationRTM la quale rappresenta l'Application-Specific Run-Time Manager utilizzato per la gestione dinamica delle applicazioni e la loro comunicazione con il framework Barbeque, operante a livello di sistema.

Com'è possibile osservare, la classe definisce due diversi costruttori *ApplicationRTM*. Queste due funzioni servono per inizializzare le funzionalità dell'AS-RTM e permettere quindi una sua corretta esecuzione. Il

parametro *bbqueEXC* è un riferimento a la classe del SYS-RTRM associata all'applicazione e serve per permettere all'AS-RTM di comunicare direttamente con esso. Gli altri due parametri sono invece un riferimento all'Operating Point Manager associato all'applicazione *opManager* e la lista di goal *goalList* che l'AS-RTM dovrà monitorare per prendere le sue decisioni riguardo l'applicazione.

Le due funzioni successive sono delle funzioni usate per la gestione dei goal. La funzione *setGoals*, che richiede come parametro la lista dei goal contenuta nel parametro *goalsList*, si occupa di assegnare i goal da utilizzare all'interno della classe ApplicationRTM. La funzione *checkGoals*, invece, controlla tutti i goal e i loro rispettivi obiettivi dichiarati all'AS-RTM sono stati raggiunti o meno; il parametro *goalsInfo* conterrà le informazioni riguardo la realizzazione di ogni goal. Più precisamente, il parametro *goalsInfo* è una lista di oggetti della classe *GoalInfo* che permette di conoscere ogni dettaglio riguardo ai goal ed al loro corrente raggiungimento o meno. Questa classe contiene diversi parametri:

- *metricName*: identifica il nome della metrica associata al goal;
- *achieved*: è una lista di valori booleani che indicano per ogni obiettivo del goal se sono stati raggiunti o meno;
- *targetGoals*: è la lista dei valori desiderati per ognuno degli obiettivi del goal;
- *observedValue*: indica il valore attuale della metrica, aggiornato all'ultima chiamata di *checkGoal*
- *relativeErrors*: è la lista di tutti gli errori relativi, rappresentanti lo scostamento percentuale dal valore desiderato per ogni obiettivo del goal;
- *naps*: è una lista di NAP (Normalised Actual Penalty), il formato di dato utilizzato per comunicare con il SYS-RTM le richieste aggiuntive di risorse necessarie per soddisfare gli obiettivi del goal.

Sono presenti anche alcune funzioni che semplificano l'accesso aggregato per alcuni di questi parametri, come ad esempio *isAchieved* che permette di conoscere rapidamente se tutti gli obiettivi di un goal sono stati raggiunti o meno o *getMaxRelativeError* che restituisce il massimo errore relativo tra tutti i vari obiettivi di un goal.

Ritornando alla classe ApplicationRTM, si procede all'analisi della funzione *adjustConstraints*. Questa funzione si occupa di modificare i vincoli di ricerca dei punti operativi in base allo stato attuale dell'applicazione rispetto al raggiungimento dei suoi goal. É una dei componenti chiave dell'AS-RTM poichè è la funzione che si occupa di adattare il modello calcolato a design-time rispetto alle perturbazioni del sistema o dell'applicazione. Accetta quattro diversi parametri:

- *currentOP* è il punto operativo correntemente usato;
- *goalsInfo* è, come già spiegato precedentemente nel capitolo, una lista contenente diverse informazioni rispetto ogni goal;
- *opFilters* è la lista di vincoli fornita dal SYS-RTRM e dallo sviluppatore. Verrà modificata dalla funzione, se necessario, per adattare il comportamento dell'applicazione al nuovo modello stimato a run-time;
- *switchThreshold* è una misura che rappresenta quanto un'applicazione sia disposta a rilasciare parte delle sue risorse quando le metriche monitorate sono al di sopra del goal. Ogni qual volta un goal viene raggiunto dall'applicazione, la funzione *adjustThreshold* deciderà se modificare i vincoli dell'applicazione per far in modo che il comportamento dell'applicazione si avvicini il più possibile al limite fornito dal vincolo del goal. Questa operazione viene effettuata per evitare un inutile spreco di risorse se il goal è stato superato per un numero di volte pari a *switchThreshold*. Per chiarire il concetto supponiamo di avere un'applicazione con un goal dichiarato come limite inferiore sul throughput di 100Mb/s ed il parametro *switchThreshold* con

valore 0.2. In caso di raggiungimento del goal i vari vincoli sulla scelta di un nuovo punto operativo verranno modificati nel caso di throughput superiori a 120Mb/s, cercando di portare le prestazioni dell'applicazione vicino alla soglia dei 100Mb/s.

La funzione *getNapAndRelativeError* è una funzione di supporto che a partire dalla lista di informazioni riguardo ai vari goal, riempie le altre due variabili ricevute in ingresso con il massimo errore relativo e il massimo NAP ricavato dalla precedente fase di verifica dei goal dell'applicazione; queste due differenti metriche sono già state discusse nel Capitolo 5. Normalmente questa funzione non viene chiamata direttamente ma può essere usata anche dallo sviluppatore nel caso preferisca gestire autonomamente i punti operativi dell'applicazione.

Le due funzioni rimanenti, entrambe chiamate *getNextOp* restituiscono il punto operativo successivo a quello passato come parametro tramite la variabile *op*. Entrambe le versioni di questa funzione accettano come parametri *opFilters* e *switchThreshold*. Il significato di questi due parametri è già stato illustrato in questo capitolo. L'unica differenza tra queste due funzioni è la presenza in una di queste del parametro *goalsInfo*. Il motivo di questa differenza è dato dal fatto che se lo sviluppatore decide di affidarsi completamente alle funzionalità dell'AS-RTM allora sarà necessario chiamare soltanto la funzione *getNextOP* con il minor numero di parametri. Sarà la funzione stessa ad occuparsi di verificare lo stato dei vari goal ed agire di conseguenza. In caso contrario, invece, è lo sviluppatore che si occupa di verificare lo stato dell'applicazione. Quindi nel caso volesse poi usare le funzionalità dell'AS-RTM senza dover rivalutare inutilmente per una seconda volta i goal, potrà chiamare la funzione *getNextOP* e passare come parametro le informazioni sui vari goal determinate in precedenza. L'utilizzo automatico delle funzionalità della classe verrà illustrato più chiaramente nella prossima sezione.

7.3 Flusso d'esecuzione

Questa sezione si occupa di descrivere il flusso d'esecuzione di un'applicazione all'interno del framework ARGO. La Figura 7.3 mostra, tramite un diagramma di flusso, i suoi concetti principali.

Con questa rappresentazione si presuppone che la fase a design-time sia stata conclusa con successo e che siano quindi disponibili una lista di Working Mode per il SYS-RTRM e una lista di Operating Point per l'AS-RTM. Inoltre, per evitare inutili complicazioni nel diagramma, abbiamo evitato di rappresentare la fase in cui l'applicazione inizializza la comunicazione con il SYS-RTRM tramite alcune sue chiamate a funzione specifiche. Questi concetti sono stati volutamente omessi perchè nonostante la nostra infrastruttura si interfacci con uno specifico SYS-RTRM, l'architettura è stata implementata per essere adattata semplicemente a diversi potenziali Resource Manager. Per questo motivo, e poiché il SYS-RTRM non fa parte del nostro lavoro di tesi, abbiamo preferito ometterne i dettagli quando possibile.

La prima fase del flusso d'esecuzione è formata da tre passi. Il primo consiste nella dichiarazione dei goal dell'applicazione e di altri vincoli sulla qualità del servizio che si vogliono raggiungere. In aggiunta, è possibile inserire degli ulteriori vincoli rappresentanti le condizioni iniziali per la scelta del primo punto operativo. Questi vincoli verranno, se necessario, aggiornati automaticamente durante la fase d'esecuzione dell'algoritmo. L'ultima delle tre fasi consiste invece nella dichiarazione dell'ordinamento preferito tra i punti operativi. Per semplicità d'esposizione questo blocco è stato posto solo all'inizio del flusso d'esecuzione ma le sue operazioni possono essere ripetute ogni qual volta si riveli necessario. Un tipico esempio di modifica dell'ordinamento in fase d'esecuzione è dato dalla scelta di prediligere le pure prestazioni in termini di throughtput dell'applicazione quando il sistema è connesso ad una fonte costante d'energia e passare invece alla preferenza di punti operativi a basso consumo di potenza nel

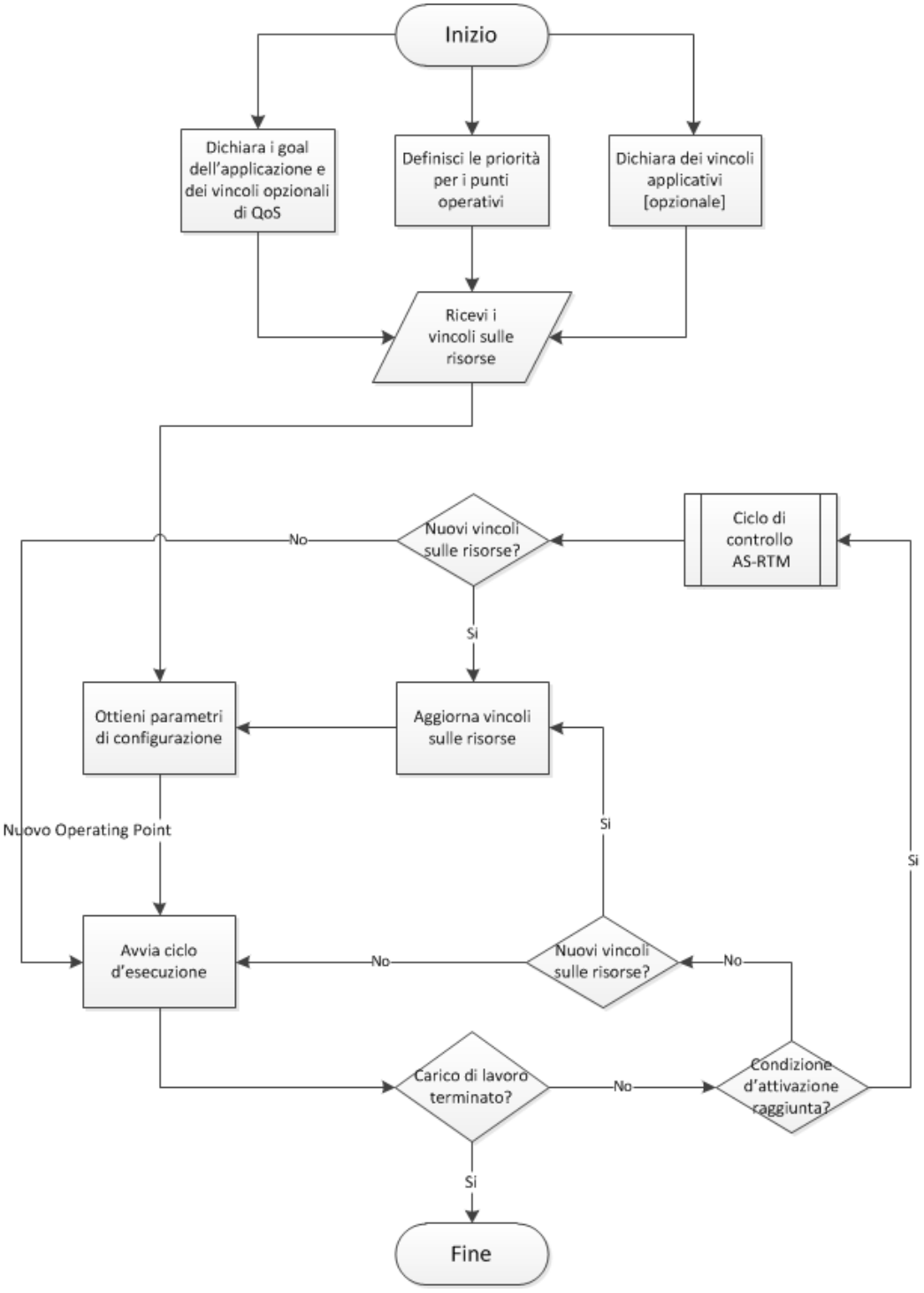


Fig. 7.3: Flusso d'esecuzione di un'applicazione all'interno del framework ARGO

caso invece in cui ci si trovi in una situazione in cui il sistema opera con una carica limitata.

Una volta inserite queste informazioni il SYS-RTRM comunicherà all'applicazione quali sono i limiti di risorse disponibili per l'applicazione stessa. A questo punto sarà possibile richiedere un punto operativo che rispetti tutti i vincoli forniti dall'applicazione e dal SYS-RTRM. La richiesta del punto operativo viene effettuata tramite l'AS-RTM e quindi comincerà l'esecuzione dell'applicazione.

Alla fine di ogni ciclo d'esecuzione saranno possibili diverse scelte. La prima, il caso più semplice, è quella data dalla fine del carico di lavoro per l'applicazione. In questo caso non sarà necessario effettuare nessun controllo aggiuntivo e l'applicazione concluderà la sua esecuzione normalmente. Nel caso in cui il carico di lavoro non sia terminato, le modalità di scelta del punto operativo saranno differenti dipendentemente dal fatto che la condizione d'attivazione per la fase di controllo sia stata verificata positivamente o meno. Questa condizione viene definita dallo sviluppatore ed identifica la frequenza con la quale viene effettuata la fase di controllo. Essa permette quindi di decidere se valutare lo stato dei goal ed effettuare le opportune modifiche ai parametri di configurazione alla fine di ogni esecuzione oppure dopo un certo numero di iterazioni. Nel caso questa condizione non sia stata raggiunta, l'applicazione innanzitutto verificherà che lo stato dei vincoli sulle risorse assegnate non sia cambiato. Se i vincoli sono invariati si procede alla fase di esecuzione successiva, altrimenti prima di passare alla fase di esecuzione verrà aggiornata la struttura *opList* permettendo di selezionare l'Operating Point corretto. Invece, se la condizione viene verificata il controllo passerà all'AS-RTM che effettuerà una serie di passi per decidere come configurare l'applicazione:

1. controllo goal: ogni goal associato all'AS-RTM viene controllato per stabilire se tutti i suoi obiettivi siano stati raggiunti o meno. Il risultato di questa fase sarà una lista di oggetti della classe *GoalInfo*, descritta precedentemente, che permetterà di effettuare delle analisi

approfondite sui vari goal;

2. analisi goal: in questa passo del ciclo di controllo vengono analizzati i risultati della fase precedente. Verrà verificato lo stato dei vari goal e determinato il massimo errore relativo e NAP per ognuno di essi. In base a queste metriche il passo successivo deciderà se e come modificare i vincoli dell'applicazione per correggerne l'esecuzione;

3. aggiornamento vincoli: nel caso ci siano dei goal che non sono stati raggiunti o vi siano goal che siano stati rispettati oltre il limite percentuale fornito da *switchThreshold*, verrà effettuato un aggiornamento dei vincoli che permettono di filtrare la lista dei punti operativi da scegliere. La modalità in cui viene scelto come modificare tali vincoli è stata già spiegata tramite un esempio fornito nella Sezione 7.2;

4. aggiornamento Operating Point: una volta che i vincoli dell'applicazione sono stati aggiornati, l'AS-RTM utilizzerà le funzioni della classe OP_Manager per ottenere un nuovo punto operativo che soddisferà i nuovi vincoli dichiarati.

Queste fasi verranno ripetute finché l'applicazione non avrà terminato la sua esecuzione.

7.4 Conclusioni

Questo capitolo ha descritto in maniera dettagliata le modalità di funzionamento e le classi del framework ARGO che si occupano della gestione a tempo d'esecuzione delle applicazioni. Grazie alle informazioni ricavate dalla fase effettuata a design-time, l'AS-RTM può operare in maniera veloce ed efficace fornendo tempestivamente la configurazione ottimale di parametri di configurazione per ogni applicazione. Gli Operating Point forniti dall'Application-Specific Run-Time Manager saranno quelli che permettono all'applicazione di raggiungere i loro goal, restando all'interno

dei vincoli di risorse forniti dal SYS-RTM. Inoltre, nel caso di perturbazioni temporanee o permanenti dello stato del sistema, quindi anche del modello dell'applicazione stimato a design-time, l'AS-RTM è capace di adattare le sue previsioni per fornire il punto operativo corretto.

Il prossimo capitolo si occuperà di validare le strategie illustrate in questo capitolo ed in quello precedente, mostrando come le tecniche sviluppate in ARGO possano essere utilizzate in ambiti realistici.

Capitolo 8

Casi d'uso e valutazioni sperimentali

Dopo aver illustrato tutte le caratteristiche e le strategie usate all'interno del nostro framework, questo capitolo presenterà una serie di casi d'uso e le relative valutazioni sperimentali. Abbiamo deciso di presentare diverse applicazioni per illustrare le tecniche a design-time e quelle a run-time. Oltre allo studio della fase di design verranno forniti dei test per ogni applicazione così da illustrare con esempi concreti il corretto funzionamento del sistema. Oltre ai test singoli per ogni applicazione verrà anche effettuato un test combinato su due applicazioni così da aumentare le problematiche di gestione ed effettuare un test di conseguenza più severo. I dati ottenuti da questi test verranno utilizzati per dimostrare come le tecniche sviluppate in ARGO possano essere applicate in ambienti realistici. La fase di design per ogni applicazione è stata eseguita su un computer dotato di un cluster di quattro Quad-Core AMD Opteron 8378 operanti alla frequenza massima di 2.4GHz [46]. Inoltre, dove non diversamente specificato, questo sistema è stato usato per i test descritti successivamente.

8.1 Applicazione MandelbrotSetArea

Questo programma calcola un'approssimazione dell'area dell'insieme di Mandelbrot usando il metodo Montecarlo. L'insieme di Mandelbrot è un frattale definito da una serie di valori c nel piano complesso con la sequenza $z_{n+1} = z_n^2 + c$. La condizione iniziale $z_0 = 0$ garantisce la convergenza della serie. Il problema del calcolo dell'area di Mandelbrot è molto discusso in quanto non è semplice determinare una buona approssimazione della stessa. Il programma implementato esplora il rettangolo che va dalle coordinate (-2.0, 0.5) sull'asse reale alle coordinate (0.0, 1.125) sull'asse immaginario; questo rettangolo copre la metà superiore del set di Manderlbrot.

L'applicazione viene profilata ricavando un insieme di punti composto da diverse metriche: *cpuUsage*, *memUsage*, *avgExecutionTime*, *avgError*. Il valore cpuUsage è stato acquisito tramite i servizi offerti dal SYS-RTRM. Il valore memUsage è stato misurato utilizzando la funzione *extractVmPeakSize* offerta dai MemoryMonitor discussi nel Capitolo 5. Per quanto riguarda i valori restanti, avgExecutionTime è stato ricavato tramite le funzionalità fornite dalla classe TimeMonitor e avgError, essendo intrinseco dell'applicazione, è stato restituito dalla stessa.

Le diverse configurazioni dell'applicazione sono composte da tre parametri:

- *Numero di core*: rappresenta il numero di core disponibili sulla macchina. I valori disponibili sono 1, 2, 4, 8 e 16.

- *Numero di thread*: rappresenta il parallelismo dell'applicazione. I valori disponibili sono 1, 2, 4, 8 e 16.

- *Numero di punti*: rappresenta il numero di punti utilizzati dal metodo Montecarlo per il campionamento dell'area. I valori ammissibili sono 1024 e 2048.

N° WM	Utilizzo CPU [%]	Utilizzo RAM [KB]	Valore WM [%]
1	200	58152	7
2	400	74544	20
3	796	107460	47
4	1576	173028	100

Tab. 8.1: MandelbrotSetArea: Working Mode generati dalla fase di design

Lo spazio di design per questa applicazione consiste nelle 50 combinazioni dei parametri sopra citati. Visto il numero contenuto di combinazioni possibili, è stato effettuato un Design of Experiment completo. Dopo aver effettuato l'esplorazione dell'applicazione, si è passati allo svolgimento delle altre fasi del flusso descritte nel Capitolo 6. Precisamente è stato effettuato un filtraggio e una clusterizzazione dei punti per la generazione dei Working Mode. Il filtraggio dei punti è essenziale per omettere dalla classificazione finale le combinazioni non ottimali di configurazioni. Nel nostro caso è stato effettuato in due fasi rimuovendo prima le configurazioni con numero di thread maggiore del numero di core e successivamente effettuando un filtraggio di Pareto sui valori di utilizzo della CPU. Il primo gruppo di punti è stato rimosso poiché le sue configurazioni sono state ritenute non significative dopo un'analisi preliminare del parallelismo dell'applicazione. Queste due fasi hanno portato lo spazio di design dai precedenti 50 punti ad un insieme di 22 punti ottimali. Durante il filtraggio abbiamo selezionato i punti Pareto ottimali considerando le seguenti quattro metriche: numero di core, memoria RAM, tempo di esecuzione e precisione dell'algoritmo. Il calcolo del valore dei Working Mode è stato effettuato utilizzando le metriche definite dall'utilizzo di CPU e RAM, alle quali è stato dato rispettivamente il peso di 0.8 e 0.2. Dopo aver effettuato i passi indicati, il risultato consiste in un insieme di 22 punti Pareto ottimali appartenenti a 4 Working Mode. I due insieme di punti sono visualizzati in Tabella 8.1 e Tabella 8.2.

Come è possibile vedere dalle tabelle riportate, il Working Mode 1 permette una percentuale di utilizzo massimo della CPU corrispondente

OP	WM	Core	Thread	Punti	CPU [%]	RAM [KB]	Tempo [s]	Errore
1	1	1	1	1024	100	49796	0.19600	0.04446
2	1	1	1	2048	100	50084	0.39600	0.03324
3	1	2	1	1024	100	49796	0.19000	0.04576
4	1	2	1	2048	100	50084	0.41500	0.03235
5	1	2	2	1024	200	57992	0.11500	0.04614
6	1	2	2	2048	200	58152	0.21300	0.03328
7	1	4	1	1024	100	49796	0.19200	0.04470
8	1	4	1	2048	100	50084	0.39800	0.03335
9	1	4	2	1024	200	57992	0.10400	0.04634
10	2	4	4	1024	399	74384	0.05500	0.04597
11	2	4	4	2048	393	74544	0.11300	0.03289
12	1	8	1	2048	100	50084	0.42000	0.03322
13	1	8	2	1024	200	57992	0.09900	0.04779
14	1	8	2	2048	200	58152	0.23200	0.03350
15	2	8	4	1024	399	74384	0.05000	0.04686
16	2	8	4	2048	400	74544	0.10900	0.03338
17	3	8	8	1024	794	107300	0.03100	0.04858
18	1	16	1	2048	100	50084	0.40800	0.03296
19	1	16	2	2048	199	58152	0.19400	0.03386
20	3	16	8	2048	796	107460	0.05300	0.03227
21	4	16	16	1024	1560	172868	0.01800	0.04506
22	4	16	16	2048	1576	173028	0.03400	0.03196

Tab. 8.2: MandelbrotSetArea: Operating Point generati dalla fase di design

al 200% e un utilizzo massimo di RAM di 58152KB. Il suo valore è molto basso e se ne può capire la ragione osservando la tabella dei punti operativi. Come è possible osservare, i punti di questo Working Mode forniscono una bassissima variabilità sui valori di utilizzo della CPU e della memoria RAM. La variabilità dei valori di utilizzo della CPU va da 100% a 200% mentre per la memoria RAM va da 49796KB a 58152KB. Al contrario, il WM 4 permette di variare l'uso della CPU tra il 100% e il 1576% e quello della memoria RAM tra 49796KB a 173028KB. Per questo motivo il Working Mode 4 ha valore massimo e permette di selezionare tutti i punti

operativi dell'applicazione.

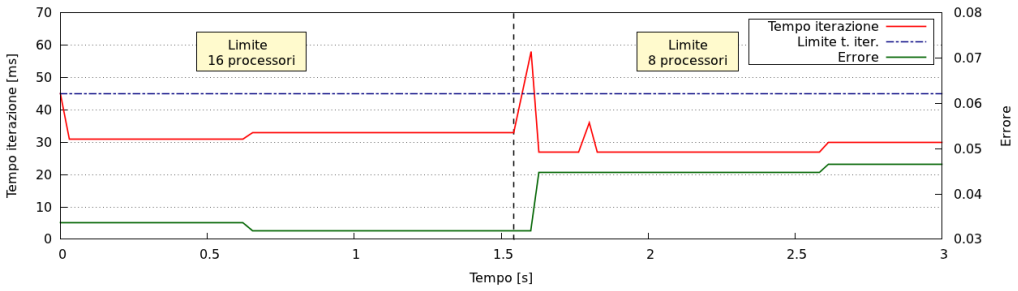


Fig. 8.1: Esecuzione dell'applicazione MandelbrotSetArea

Dopo aver descritto la caratteristiche dei Working Mode e dei punti operativi, è possibile ora analizzare il test effettuato per verificare la corretta caratterizzazione dell'applicazione e la sua efficiente gestione a run-time. Il test, la cui esecuzione è rappresentata in Figura 8.1, è stato effettuato attraverso l'esecuzione di 100 iterazioni dell'algoritmo per il calcolo dell'area dell'insieme di Mandelbrot. Il goal iniziale consiste in una latenza massima di 45ms, dove i punti operativi con miglior accuratezza sono stati definiti a priorità massima. A metà delle iterazioni previste è stata forzata una notifica di diminuzione delle risorse, portando il numero di core disponibili da 16 a 8. Nella prima fase del test, prima della diminuzione delle risorse, il punto operativo scelto prevede 16 core e 2048 punti; dopo la diminuzione di risorse il punto operativo scelto prevede invece 8 core e 1024 punti così da mantenere la latenza entro i limiti richiesti. Come è possibile confermare visionando le tabelle precedenti, l'Operating Point scelto inizialmente sarà il numero 22, mentre il successivo sarà il numero 17. In questo passaggio verrà testato anche il punto 20 in quanto la fase di design ha stimato questo punto caratterizzandolo con una precisione migliore ed un tempo d'esecuzione in grado di rispettare il goal previsto. Vista l'impossibilità di eseguire l'applicazione nel tempo richiesto, l'OP 20 verrà scartato, preferendo un punto capace di garantire le prestazioni richieste sacrificando parte della precisione dell'algoritmo.

L'overhead di controllo per questa applicazione consiste in 280μs, quindi circa 3μs ad iterazione. Esso varia da un minimo di 2μs ad un massimo di 28μs e rappresenta lo 0.009% del tempo d'esecuzione totale.

8.2 Applicazione MolecularDynamic

Questa applicazione implementa una simulazione della dinamica delle particelle. Simulando l'interazione tra molecole, per un intervallo di tempo predeterminato, si ottiene una visione di massima del loro movimento. Le diverse configurazioni dell'applicazione sono composte da due parametri:

- *Numero di core*: rappresenta il numero di core disponibili sulla macchina. I valori disponibili sono 1, 2, 4, 8 e 16.
- *Numero di thread*: rappresenta il parallelismo dell'applicazione. I valori disponibili sono 1, 2, 4, 8 e 16.

Anche in questo caso sperimentale sono state effettuate tutte le fasi precedentemente descritte per lo studio e la caratterizzazione dell'applicazione. La fase di filtraggio ha prodotto 7 punti Pareto ottimi rispetto ai 25 di partenza generati dalle metriche numero di core, memoria RAM e tempo di esecuzione. Come nel caso precedente il calcolo del valore dei Working Mode è stato effettuato utilizzando le metriche definite dall'utilizzo di CPU e RAM assegnando rispettivamente i pesi 0.8 e 0.2. Dopo aver effettuato le consuete procedure, abbiamo ottenuto un insieme di 7 punti Pareto ottimi appartenenti a 4 Working Mode. I due insieme di punti sono visualizzati in Tabella 8.3 e Tabella 8.4.

N° WM	**Utilizzo CPU [%]**	**Utilizzo RAM [KB]**	**Valore WM [%]**
1	200	58672	7
2	400	75064	20
3	799	107848	47
4	1594	173416	100

Tab. 8.3: MolecularDynamic: Working Mode generati dalla fase di design

OP	WM	Core	Thread	CPU [%]	RAM [KB]	Throughput [iter/s]
1	1	1	1	100	50476	0.38080
2	1	2	2	200	58672	0.76103
3	2	4	4	400	75064	1.51745
4	3	8	4	400	75064	1.51975
5	3	8	8	799	107848	3.03030
6	3	16	8	799	107848	3.03951
7	4	16	16	1594	173416	5.98802

Tab. 8.4: MolecularDynamic: Operating Point generati dalla fase di design

Da queste tabelle è possibile osservare come il Working Mode 1 permetta un utilizzo massimo di CPU e RAM rispettivamente del 200% e di 58672KB. Anche in questo caso il valore associato a questo WM è molto basso in quanto permette di utilizzare solo due punti operativi su sei fornendo una variabilità per l'utilizzo della CPU del 100% e per quanto riguarda la RAM di circa 8000KB. Il WM con valore massiamo, in questo caso il numero 4, potrà invece utilizzare tutti i 7 punti operativi massimizzando la variabilità delle metriche.

I test svolti con l'applicazione MolecularDynamic hanno previsto 100 iterazioni per analizzare ogni volta i movimenti delle 8192 particelle. La Figura 8.2 descrive il suo flusso d'esecuzione dei primi 20 secondi dell'applicazione. I restanti 21 secondi non sono stati visualizzati poichè non significativi.

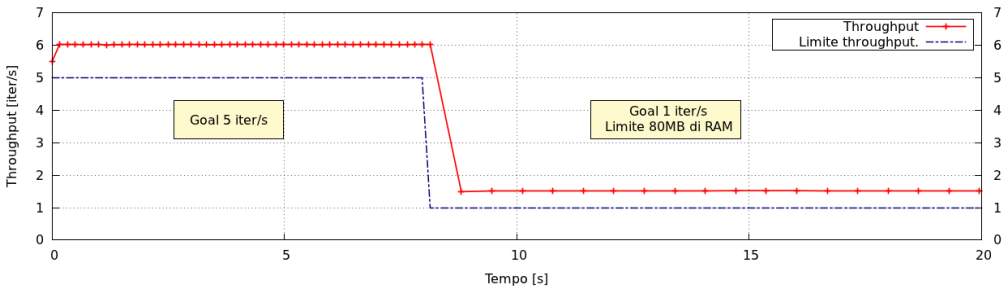


Fig. 8.2: Esecuzione dell'applicazione MolecularDynamic

L'esecuzione dell'applicazione inizia con un vincolo sul throughput minimo corrispondente a cinque iterazioni al secondo; successivamente, a metà delle iterazioni previste, viene modificato questo vincolo impostandolo ad una iterazione al secondo. Contestualmente a questa modifica, viene forzato un limite addizionale sulla memoria utilizzabile a 80MB. La configurazione iniziale di parametri utilizza 16 processori e 173376KB di memoria, corrispondenti al punto operativo 7, per poter soddisfare il vincolo sul throughput. Dopo le modifiche ai vincoli, l'esecuzione continua sfruttando 4 core e 75024KB di memoria come indicato dal punto operativo 3; questa configurazione è utilizzabile in virtù del rilassamento del vincolo sul tempo di esecuzione. In questo test l'overhead medio è $3\mu s$ e varia da un minimo di $2\mu s$ a un massimo di $20\mu s$. L'overhead totale rappresenta lo 0.0007% del tempo d'esecuzione.

8.3 Applicazione StereoMatching

Per effettuare un'analisi dettagliata della fase decisionale a run-time svolta dal framework ARGO, abbiamo interfacciato un'applicazione di Stereo Matching col nostro framework, eseguendola sull'architettura Intel Core i7-2620M. Questa architettura fa parte della famiglia di processori Sandy Bridge e fornisce 2 core fisici capaci di gestire 4 thread simultaneamente per mezzo della tecnica di *Hyperthreading*. I dettagli su questa architettura si possono consultare su [47]. La scelta di eseguire i test su questa architettura ha permesso di verificare come l'AS-RTM sia in grado di reagire a fronte di un modello dell'applicazione non esplicitamente creato per l'architettura testata.

L'applicazione Stereo Matching accetta come valori d'ingresso due immagini che rappresentano lo stesso soggetto ma provengono da due diverse angolazioni. L'applicazione calcola una mappa di disparità che rappresenta la profondità dell'oggetto raffigurato nelle immagini. Una descrizione dell'applicazione e il suo algoritmo risolutivo è disponibile

nell'articolo [48]. Questa applicazione è intrinsicamente parallela e usa molto intensamente i processori disponibili nel sistema tramite il modello di programmazione OpenMP. L'insieme dei parametri di configurazione per l'applicazione è definito in [48]. Da questo insieme abbiamo estratto quattro parametri:

- *Numero di thread*: Rappresenta il massimo parallelismo dell'applicazione. I valori considerati sono 1, 2, 4, 8.
- *Maximum arm length*: Si riferisce alla lunghezza massima del ramo della croce utilizzata dall'algoritmo per generare le regioni di supporto. I valori analizzati variano da 1 a 18.
- *Confidenza*: Rappresenta il valore di soglia utilizzato per costruire la regione di supporto per determinare se il pixel analizzato appartiene o meno alla stessa regione. I valori analizzati variano tra 14 a 64 a passi di 5 unità.
- *Hypo step*: É l'intervallo di campionamento usato dall'algoritmo per verificare l'ipotesi di disparità. Può assumere i valori 1, 2 e 3.

Lo spazio di design è composto da 2376 possibili combinazioni di questi 4 parametri.

Abbiamo esteso le funzionalità dell'applicazione Stereo Matching usando il nostro framework per monitorarne le prestazioni durante la fase d'esecuzione. I parametri misurati sono il throughput, espresso in KPixel/s, l'utilizzo del processore e della memoria RAM. Oltre a queste metriche ne è stata aggiunta un'altra a design-time. Questa metrica è chiamata Disparity Error ed è inversamente correlata alla qualità dell'immagine fornita come risultato dall'algoritmo: più alto è il disparity error, minore è la qualità del risultato. Questo valore è verificato in fase di design utilizzando una tabella di verità per le immagini elaborate. Il disparity error verificato a design-time verrà usato in fase d'esecuzione per predire la qualità del risultato per la configurazione di parametri selezionata.

N° WM	Utilizzo CPU [%]	Utilizzo RAM [KB]	Valore WM [%]
1	99	568844	8
2	194	568844	38
3	350	568844	88
4	386	568844	100

Tab. 8.5: StereoMatching: Working Mode generati dalla fase di design

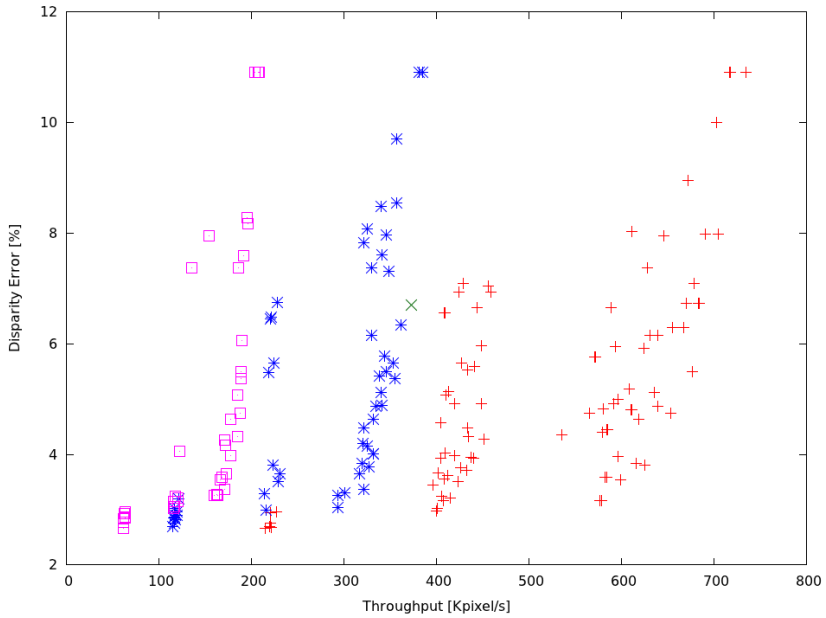
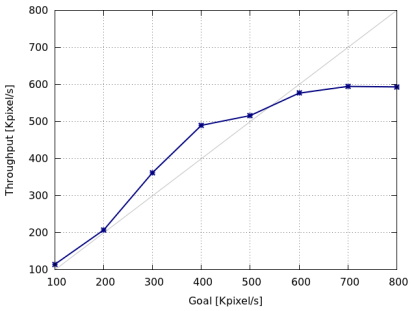
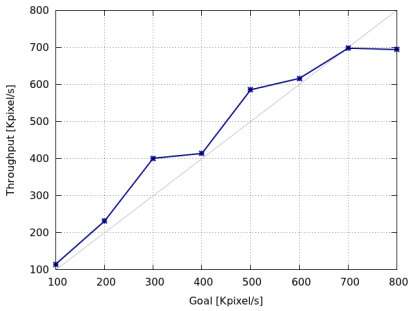


Fig. 8.3: StereoMatching: Grafico della dispersione dei punti operativi

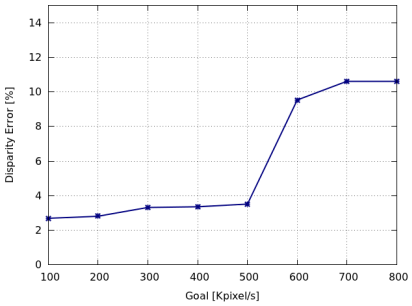
Abbiamo applicato la metodologia a design-time presentata nel Capitolo 6, utilizzando un Design of Experiment di tipo Random e un algoritmo MOSA (Multi-objective Simulated Annealing) per la fase d'esplorazione dei punti. Sia la fase di ottimizzazione sia quella di filtraggio includono tutte le metriche descritte precedentemente e forniscono come risultato un insieme totale di 241 punti operativi ottimali, clusterizzati a loro volta in 5 Working Mode. Poichè il numero di punti generati è estremamente elevato per essere visualizzato in una tabella, queste configurazioni verranno visualizzate in Figura 8.3 dove gli Operating Point sono colorati in base al rispettivo Working Mode presente in Tabella 8.5. I dati visualizzati



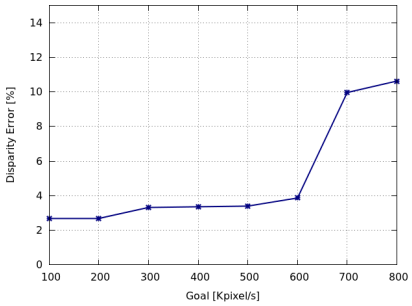
(a) Throughput osservato usando un massimo di 2 thread



(b) Throughput osservato usando un massimo di 4 thread



(c) Disparity error ottenuto con 2 thread



(d) Disparity error ottenuto con 4 thread

Fig. 8.4: StereoMatching: throughput osservato e disparity error ottenuto variando il goal riferito al throughput e il numero di risorse disponibili

consistono nei 216 punti dei 4 Working Mode che utilizzano un numero di processori minore di quattro; questi punti saranno utilizzati per i test su questa architettura.

I risultati forniti dalla fase a design-time sono stati utilizzati per effettuare alcuni test intensivi sul framework a fronte di diversi eventi.

I primi due test sull'applicazione sono stati effettuati per verificare come l'AS-RTM risponda effettivamente alle richieste di punti operativi con dei vincoli forniti dall'applicazione. Per questo motivo è stato eseguito un test nel quale viene fornito un limite inferiore al throughput e una priorità su punti operativi ad alta precisione. Ogni 10 coppie di immagini elaborate l'applicazione varia le sue richieste di throughput, incrementan-

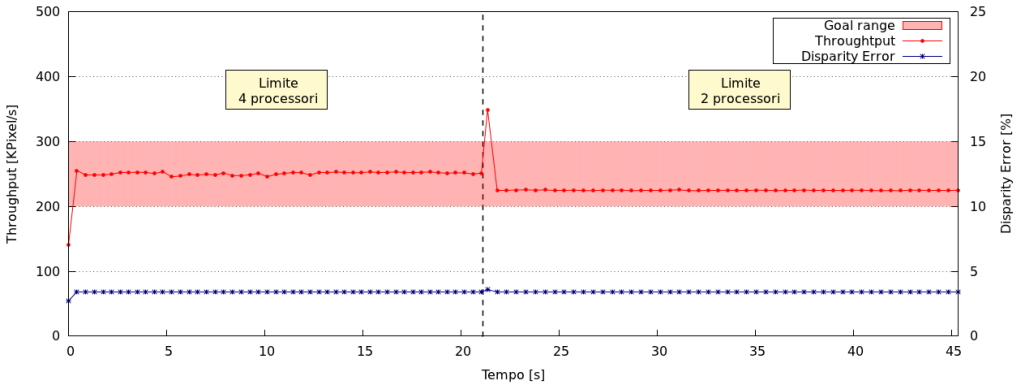


Fig. 8.5: Esecuzione dell'applicazione Stereo Matching con una perturbazione sull'uso massimo di risorse

dole di 100KPixel/s. Il test è effettuato forzando due limiti diversi sul numero massimo di thread disponibili sull'architettura: 2 o 4 thread.

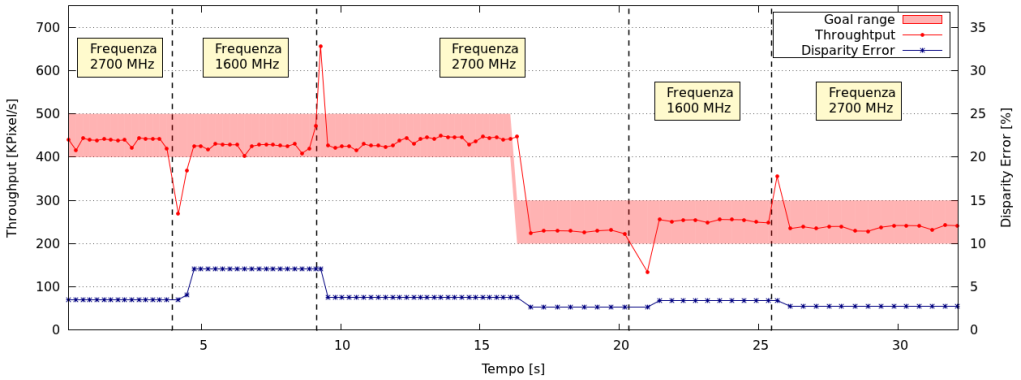
La Figura 8.4 mostra i risultati di questo test dove i due grafici in alto rappresentano il throughput fornito all'applicazione a fronte dei goal richiesti. Questo valore è dato dalla media dei throughput misurati nelle 10 elaborazioni. I due grafici in basso, invece, mostrano come l'accuratezza dell'algoritmo varia all'aumentare del troughput richiesto.
É facile identificare l'approccio conservativo dell'applicazione nei confronti della precisione dell'algoritmo. Infatti l'AS-RTM sceglie punti operativi a massima precisione finchè è possibile non peggiorare l'accuratezza dei risultati. Una volta riconosciuta l'impossibilità di rispettare i goal richiesti dall'applicazione senza peggiorare la qualità del risultato finale, l'AS-RTM sceglierà i punti migliori che siano capaci di garantire la velocità d'elaborazione richiesta.

Il secondo test effettuato consiste nell'esecuzione dell'applicazione Stereo Matching con un vincolo espresso come un intervallo di valori ammissibili per il throughput dell'applicazione. A metà della fase d'esecuzione viene forzata una notifica riguardante la modifica delle risorse disponibili, modificando il numero massimo di thread da 4 a 2. La Figura 8.5 mostra il comportamento dell'applicazione in risposta a tale notifica. É possibile

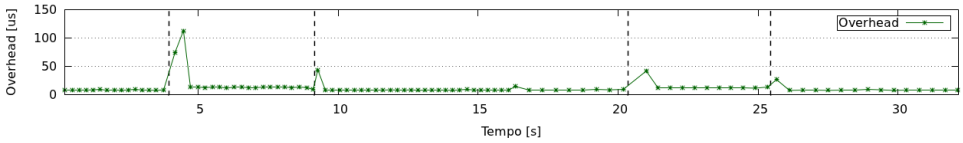
vedere come l'applicazione adatta rapidamente le sue caratteristiche a fronte dei diversi vincoli di risorse forniti. In questo caso l'AS-RTM è capace di trovare un punto operativo a precisione simile a quello precedente sacrificando leggermente il throughput dell'applicazione.

Il secondo set di test è stato effettuato con lo scopo di verificare il comportamento e la reattività del nostro framework a fronte di combinazioni di perturbazioni sull'esecuzione di un'applicazione. Abbiamo eseguito l'applicazione Stereo Matching con un carico di lavoro totale di 100 coppie di immagini e durante l'esecuzione dell'algoritmo abbiamo forzato alcuni cambiamenti al sistema. L'applicazione viene avviata con la frequenza dei processori a 2700MHz e un gol sul throughput di 400KPixel/s e 500Kpixel/s come limiti inferiore e superiore. Dopo circa 4 secondi viene effettuato un abbassamento della frequenza dei processori portando quest'ultima a 1600MHz. Questa modifica è stata forzata per simulare un possibile scaling di frequenza dovuto a problemi termici dell'architettura. L'esecuzione dell'applicazione continua con questa frequenza per 5 secondi prima di passare a quella originale. Una volta che il 65% del carico di lavoro è terminato viene modificato il goal dell'applicazione con throughput compreso tra 200Kpixel/s e 300Kpixel/s. Si ripete lo stesso cambiamento di frequenza effettuato precedentemente. Lo stesso test è stato successivamente effettuato con l'ordine delle frequenze invertite. I valori di frequenza sono stati scelti in modo da rappresentare possibili scelte del sistema operativo o del SYS-RTM e rappresentano una frequenza medio-alta ed una medio-bassa.

La figure 8.6 e 8.7 mostrano i grafici dell'esecuzione di questi test. É possibile notare come la reazione del sistema sia molto veloce: i cambiamenti di goal e dell'ambiente esterno sono bilanciati entro l'elaborazione di uno o due frame, mantenendo il sistema in un Operating Point quasi ottimale. Nella fasi dell'applicazione in cui la frequenza diminuisce, l'AS-RTM decide di cambiare i parametri dell'applicazione riportando il throughput al valore desiderato a costo di un peggioramento della preci-



(a) Comportamento dell'applicazione



(b) Overhead dell'AS-RTM

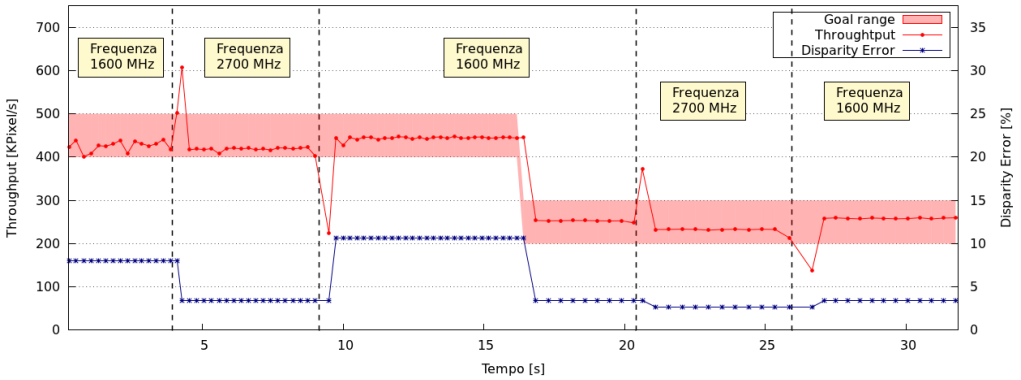
Fig. 8.6: Esecuzione dell'applicazione Stereo Matching con goal differenti e perturbazioni di frequenza - test 1

sione dell'applicazione. Una volta che questa perturbazione termina, la precisione originale viene ripristinata.

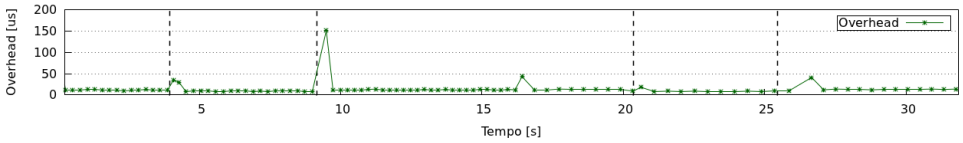
É importante notare che l'AS-RTM è in grado di selezionare una configurazione più precisa quando la richiesta di throughput, in questo caso il goal, diventa meno esigente. In questo caso l'applicazione può essere eseguita più lentamente ma più accuratamente mantenendo il throughput all'interno dei vincoli indicati.

Il motore decisionale dell'AS-RTM è in grado di stimare, come già illustrato nel Capitolo 7, il corretto punto operativo anche quando il modello dato a design time non è più valido ma ha variazioni proporzionali, ad esempio a causa di frequency scaling o di usura di componenti.

Per l'esecuzione rappresentata in Figura 8.6, l'overhead generato dal controllo del goal e dalla fase decisionale consiste solo in 1.31ms su un tempo d'esecuzione totale di 33,23s. Ciò significa che l'overhead com-



(a) Comportamento dell'applicazione



(b) Overhead dell'AS-RTM

Fig. 8.7: Esecuzione dell'applicazione Stereo Matching con goal differenti e perturbazioni di frequenza - test 2

plessivo equivale soltanto allo 0.0039% del tempo di esecuzione. Questo overhead è in media 13μs per ogni frame e varia da un minimo di 8μs ad un massimo di 113μs. Invece per l'esecuzione rappresentata dalla Figura 8.7, l'overhead di controllo è di 1.88ms su un tempo d'esecuzione totale di 33,18s. Esso rappresenta lo 0.0056% del tempo d'esecuzione ed è caratterizzato da media 19μs per frame, valore minimo 8μs e massimo 143μs. La differenza dei valori di overhead ricavata si deduce dal fatto che mentre nel primo dei due test l'esecuzione dell'applicazione è stata svolta a 2700MHz con due interferenze brevi di 1600MHz, nel secondo caso la situazione è l'opposta. La prima esecuzione viene effettuata ad una frequenza media dei core di 2367MHz mentre la seconda a 1933MHz. Da notare che il comportamento dell'applicazione non varia: il tempo d'esecuzione totale dei due test si discosta soltanto di 0.05s.

Un'altra nota sulle prestazioni dell'AS-RTM riguarda la condizione

d'attivazione della fase di controllo. Questi test rappresentano il caso peggiore in cui la verifica dello stato dell'applicazione viene svolta ad ogni iterazione. Nel caso si scelga una frequenza diversa per la fase di controllo l'overhead verrebbe ridotto di conseguenza. Inoltre, in questa applicazione abbiamo utilizzato un insieme di 241 punti operativi. Ridurre il numero di Operating Point, quindi lo spazio di ricerca per l'algoritmo di decisione, può portare a significative riduzioni dell'overhead, in particolare per l'esecuzione nel caso peggiore in cui sia necessario scandire tutta la lista di Operating Point.

Il test appena effettuato ha un'altra valenza oltre quelle già descritte. Il modello di questa applicazione è stato generato su una macchina diversa da quella su cui è stato eseguito, quindi è stato possibile mostrare la sua adattabilità su diverse architetture.

8.4 Applicazione PatternMatching

Il pattern matching è una procedura atta a riconoscere una o più sequenze di simboli in una stringa chiamata pattern. L'algoritmo per il confronto dei simboli utilizza un'automa a stati finiti per rappresentare l'insieme dei pattern. Questo automa è un modello di calcolo semplice rappresentabile come un piccolo dispositivo che legge una stringa di input, un carattere alla volta, su un nastro e la elabora facendo uso di un meccanismo elementare. L'automa rappresenta tutti i pattern da riconoscere; per avere un riscontro, dato un input, basta percorrere l'automa dall'inizio. Per avere ulteriori dettagli sul pattern matching e sull'implementazione utilizzata è possibile riferirsi a [49]. In questa applicazione le diverse configurazioni verranno create variando solamente in numero di thread; i valori disponibili sono 1, 2, 4, 8 e 16.

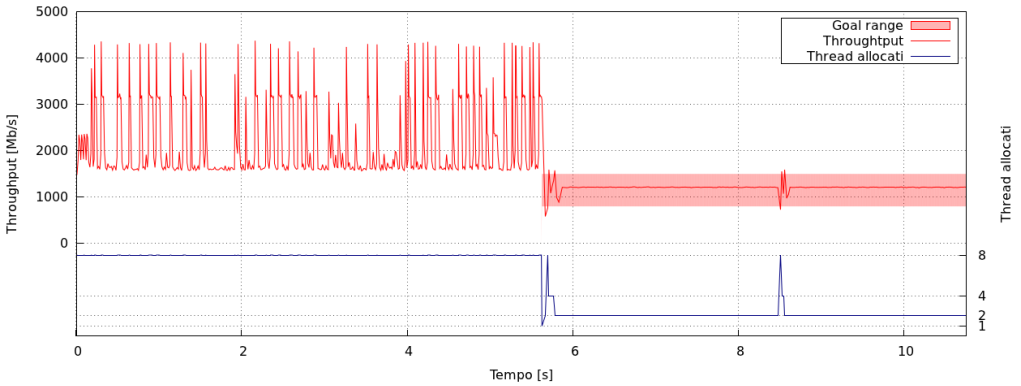
WM	CPU [%]	RAM [KB]	Valore WM [%]
1	99	127120	0
2	187	242972	25
3	315	324900	55
4	547	357684	100

Tab. 8.6: PatternMatching: Working Mode generati dalla fase di design

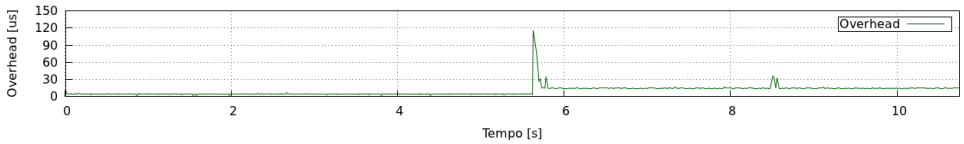
OP	WM	Thread	Throughput [Mb/s]	Utilizzo CPU [%]	Utilizzo RAM [KB]
1	1	1	621	99	127120
2	2	2	1198	187	242972
3	3	4	2108	315	324900
4	4	8	2272	547	357684

Tab. 8.7: PatternMatching: Operating Point generati dalla fase di design

Essendo questo un algoritmo molto utilizzato all'interno degli IDS (Intrusion Detection Sistem), gli input scelti per questo test sono quanto più possibile aderenti alla realtà. Essi sono stati generati in maniera casuale utilizzando firme nocive realmente riconosciute dagli IDS; di conseguenza anche i pattern sono costituiti da firme reali. Tutti i test sono stati eseguiti con pacchetti da 1500 Byte, simulando l'MTU (Maximum Transmission Unit) massimo delle reti ethernet. In questo test gli input processati contengono una percentuale di traffico da riscontrare del 25%. Il test dell'applicazione inizia senza alcun goal definito. Dato che la priorità è stata impostata sul throughput più alto, l'esecuzione inizia con 8 thread. Arrivati al 65% del carico totale, viene impostato un goal sulla media degli ultimi dieci throughput tra 800 e 1500Mb/s e l'esecuzione come visibile dal grafico in Figura 8.8-a procede correttamente. Come osservabile dal grafico in Figura 8.8-b si ottiene un overhead medio di 14μs per ogni blocco di 2048 pacchetti, con minimo 2μs e massimo 116μs.



(a) Comportamento dell'applicazione pattern matching



(b) Overhead dell'applicazione pattern matching

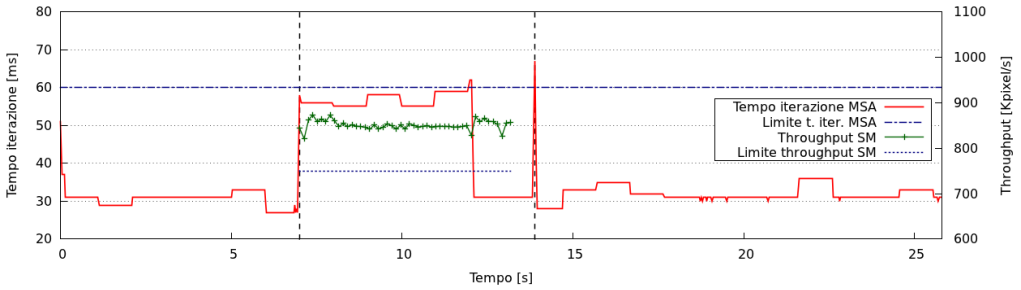
Fig. 8.8: Esecuzione dell'applicazione di pattern matching

8.5 Esecuzione combinata di più applicazioni

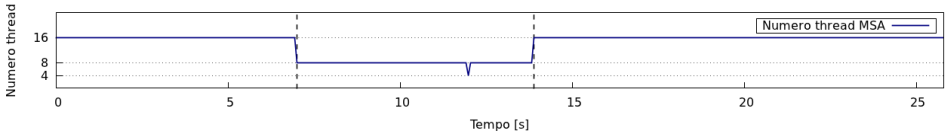
Come ultimo test, a riprova del corretto funzionamento del sistema progettato, abbiamo eseguito le applicazioni StereoMatching e MandelbrotSetArea in contemporanea così da valutare la corretta gestione di più applicazioni che concorrono all'uso delle risorse.

8.5.1 Test 1: controllo reattivo

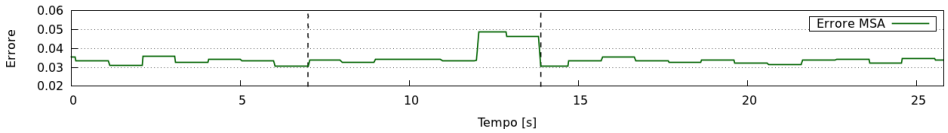
Il grafico in Figura 8.9-a visualizza l'esecuzione combinata delle due applicazioni. Inizialmente viene eseguita solo l'applicazione MandelbrotSetArea con un vincolo sulla latenza minima di 60ms. Essendo l'unica applicazione presente nel sistema viene eseguita con il punto operativo che permette di ottenere le massime prestazioni: 16 thread e 2048 punti. Dopo sette secondi viene eseguita l'applicazione StereoMatching con priorità più alta; per questo motivo il SYS-RTRM designa il Working Mode



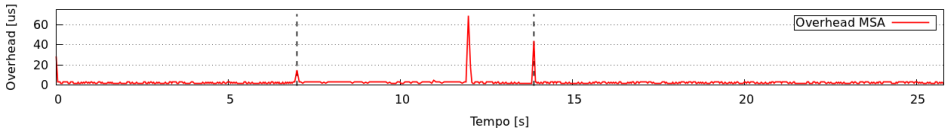
(a) Comportamento delle applicazioni MandelbrotSetArea e StereoMatching



(b) MandelbrotSetArea: numero di thread allocati



(c) MandelbrotSetArea: overhead dell'AS-RTM



(d) MandelbrotSetArea: overhead dell'AS-RTM

Fig. 8.9: Comportamento esecuzione combinata delle due applicazioni

con valore più alto per l'esecuzione di StereoMatching con 8 thread. A seguito dell'esecuzione del secondo programma viene notificata all'applicazione MandelbrotSetArea una diminuzione delle risorse passando di conseguenza ad un'esecuzione con 8 thread.

L'esecuzione dell'applicazione MandelbrotSetArea continua con questi punti operativi per un paio di iterazioni, precisamente fino a che non si verifica un fallimento nella verifica del goal. A seguito del goal non verificato vengono effettuate una serie di modifiche del punto operativo con cui eseguire questa applicazione; maggiori dettagli verranno forniti in

seguito.

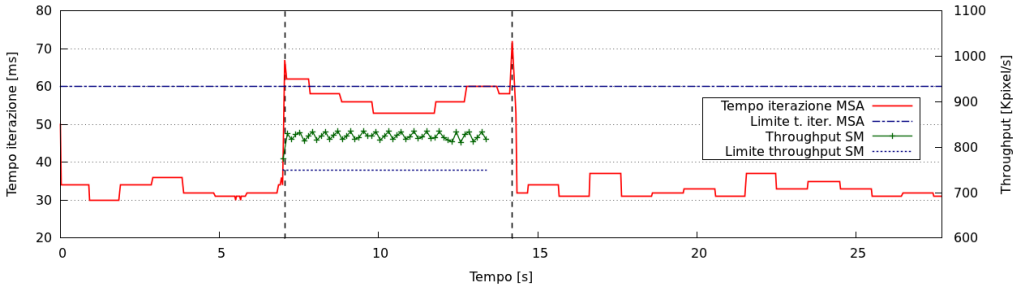
L'applicazione StereoMatching, essendo quella con priorità maggiore, non risente in alcun modo della presenza nel sistema di un'altra applicazione; viene eseguita per tutta la sua durata con il Working Mode a più alte prestazioni rispettando il suo goal di 750Kpixel/s Il picco osservabile in Figura 8.9-a attorno al dodicesimo secondo, indica che il goal non è stato rispettato. In questo caso l'AS-RTM modifica la modalità di esecuzione attraverso i punti operativi cercando di rispettare il goal imposto. A seguito del controllo effettuato viene selezionata una modalità a 4 thread con una precisione migliore rispetto alle altre e che, secondo le analisi effettuate a design-time, dovrebbe garantire una precisione minore di 60ms. Dopo un iterazione dell'algoritmo eseguita con il punto operativo selezionato, il goal non è ancora rispettato; per questo l'applicazione passa ad un esecuzione con 8 thread e 1024 punti per rispettare il goal aumentando le prestazioni a discapito della qualità. Questo comportamento è osservabile anche in Figura 8.9-b dove è presente il passaggio all'esecuzione con 4 thread e l'immediato ripristino dell'esecuzione a 8 thread; la variazione della precisione di questi punti operativi è osservabile in Figura 8.9-c.

Terminata l'esecuzione dello StereoMatching il resource manager notifica all'applicazione MandelbrotSetArea la nuova disponibilità di risorse passando all'esecuzione di quest'ultima nella modalità più performante, coincidente con quella iniziale.

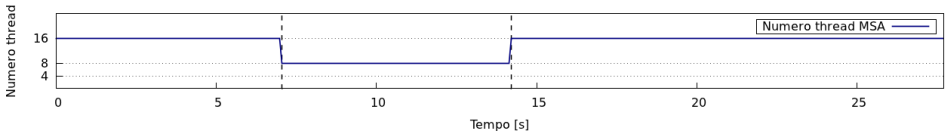
Un osservazione importante osservabile nella Figura 8.9-a, nell'intorno del 14 secondo, riguarda il picco del tempo d'esecuzione per l'applicazione MandelbrotSetArea. Questo comportamento è imputabile agli overhead dati dal SYS-RTRM che si occupa della notifica del nuovo Working Mode. Nella Figura 8.9-d è possibile osservare gli overhead introdotti dall'AS-RTM. Si possono notare tre picchi in corrispondenza del cambio di Working Mode o nel caso del non raggiungimento del goal. L'entità di questo overhead è di tre ordini di grandezza inferiore rispetto all'overhead introdotto dal SYS-RTRM. Più precisamente ad un tempo di esecuzione

totale di 26.41s corrisponde un overhead di 2.03ms, equivalente allo 0.008% del tempo totale. L'overhead introdotto ha valore medio 3μs con valore minimo 2μs e massimo 69μs.

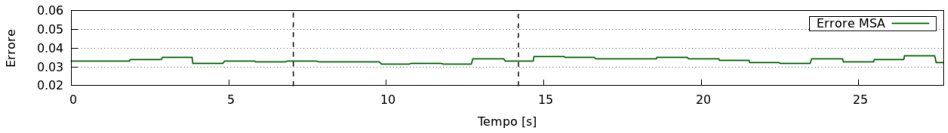
8.5.2 Test 2: controllo a finestra



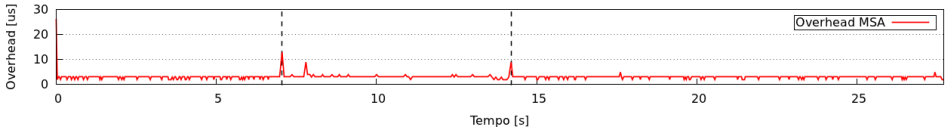
(a) Comportamento delle applicazioni MandelbrotSetArea e StereoMatching



(b) MandelbrotSetArea: numero di thread allocati



(c) MandelbrotSetArea: overhead dell'AS-RTM



(d) MandelbrotSetArea: overhead dell'AS-RTM

Fig. 8.10: Comportamento esecuzione combinata delle due applicazioni con finestra di dimensione 20

Abbiamo effettuato un test simile al precedente con le stesse condizioni iniziali ma con il goal dell'applicazione MandelbrotSetArea dichiarato

sulla media delle ultime 20 iterazioni invece che sull'ultima.

Considerando una finestra di 20 elementi si ottiene un rilassamento locale del goal in quanto la valutazione non viene effettuata sul valore istantaneo prodotto ma su una combinazione lineare degli ultimi 20 valori. L'introduzione di questa finestra permette quindi di tollerare un goal non rispettato se la sua combinazione lineare con gli altri elementi rispetta il goal. Considerato il goal di $60\mu s$ e la finestra di dimensione 20, la tolleranza media per ogni elemento della finestra è dell'1.6%. Questo comportamento è visibile in Figura 8.9-a.

Rispetto al caso precedente l'applicazione MandelbrotSetArea continua la sua esecuzione con i punti a massima precisione che rispettano i vincoli delle risorse; l'allocazione dei thread è visibile in Figura 8.9-b. Una conseguenza di questo comportamento è un errore medio d'esecuzione molto più basso rispetto al caso precedente ed un overhead di quasi tre volte inferiore. L'esecuzione di questo test richiede un tempo totale di $28.19s$ con un overhead di controllo di $2.2ms$; la media per ogni iterazione è $3\mu s$ con un minimo di $2\mu s$ e massimo $26\mu s$. In Figura 8.9-a sono visibili due picchi del tempo di esecuzione durante la fase di modifica dei Working Mode, imputabili sempre all'interazione con il SYS-RTRM.

Le informazioni ricavate da questo test evidenziano l'importanza delle gestione autonoma dell'applicazione diminuendo il carico di lavoro del SYS-RTRM e di conseguenza l'overhead totale.

8.6 Conclusioni

Questo capitolo ha introdotto i test pianificati per valutare il framework ARGO. I risultati ottenuti applicando la metodologia proposta su quattro applicazioni di natura diversa possono considerarsi promettenti ed aderenti alle aspettative presentate nei capitoli introduttivi. Infatti, i risultati dimostrano una corretta gestione delle applicazioni nei termini dei goal impostati con overhead molto contenuti e spesso trascurabili. Il

capitolo successivo proporrà alcuni possibili sviluppi futuri insieme a delle considerazioni generali.

Capitolo 9

Conclusioni

In questo lavoro di tesi sono stati affrontati alcuni problemi derivanti dall'adozione di architetture multiprocessore. Queste architetture, per essere sfruttate correttamente, richiedono un approccio alla programmazione diverso da quello utilizzato per le soluzioni a singolo core considerate ora obsolete. Come illustrato nei capitoli precedenti, sono stati adottati diversi criteri per fornire un supporto alle applicazioni.

É stato implementato il framework ARGO che, attraverso le fasi descritte nei capitoli precedenti, offre un supporto a run-time per l'applicazione adattando la sua esecuzione alle risorse correntemente disponibili. Per l'analisi delle applicazioni e per il calcolo e l'analisi delle modalità di funzionamento sono state adottate tecniche di Design Space Exploration e di Design of Experiment.

L'implementazione del framework ha considerato due livelli logici differenti, chiamati SYS-RTRM e AS-RTM; il primo è usato per gestire le risorse di sistema mentre il secondo è orientato alla gestione delle applicazioni. Il vantaggio di questo approccio risiede nella possibilità di prendere decisioni nel livello logico con la maggiore conoscenza del problema. Questo lavoro di tesi si è occupato della definizione e dell'implementazione di tutte le caratteristiche dell'AS-RTM proposte.

Un'ulteriore strategia adottata parallelamente a quelle appena elencate riguarda il cambiamento dell'approccio con il quale si valuta un'applicazione. Si è passati da un approccio orientato alle prestazioni ad uno orientato ai goal permettendo allo sviluppatore di semplificare la fase di controllo dell'applicazione.

9.1 Obiettivi raggiunti

Questo lavoro di tesi ci ha permesso di affrontare il problema della gestione di applicazioni a run-time. Per assolvere questo obiettivo, la prima fase affrontata ha riguardato la Design Space Exploration, utilizzata per la ricerca di un insieme di Operating Point e Working Mode adatti ad una corretta esecuzione dell'applicazione gestita. Successivamente è stata eseguita un'analisi delle attività di run-time management che ci ha permesso di definire e disaccoppiare le fasi a design-time e a run-time. Il disaccoppiamento di queste attività ha facilitato l'adozione di un approccio decisionale distribuito in cui ogni decisione è presa al livello logico con maggiore competenza per il problema posto. Nel nostro caso queste decisioni vengono effettuate dal SYS-RTRM e da una serie di AS-RTM, uno per ogni applicazione.

Un'altro sviluppo effettuato con l'intento di facilitare il compito degli sviluppatori, riguarda l'adozione di un approccio orientato ai goal. É stata introdotta la possibilità di dichiarare dei goal per un'applicazione con l'intento di semplificare l'analisi delle prestazioni comunemente svolta per valutare l'applicazione stessa.

Gli aspetti specifici appena illustrati sono stati concretizzati all'interno del framework ARGO, un Application-Specific Run-Time Manager orientato ai goal che utilizza sia tecniche a design-time sia tecniche a run-time. L'implementazione di ARGO consente di effettuare offline gran parte delle elaborazioni necessarie a caratterizzare l'applicazione. Poichè general-

mente queste fasi richiedono un'elevata quantità di risorse, spostarle a design-time rende possibile l'uso di un run-time manager più leggero.

I test effettuati hanno confermato la validità della soluzione proposta evidenziando il corretto funzionamento di ARGO e la sua corretta gestione dell'applicazione a run-time. La fase di controllo dell'applicazione, eseguita in tempo trascurabile rispetto al tempo di esecuzione totale, presenta un bassissimo overhead non appesantendo l'esecuzione.

9.2 Sviluppi futuri

In questa sezione verranno trattati gli sviluppi futuri ai quali abbiamo pensato durante questo lavoro di tesi. Una prima evoluzione del framework riguarda lo sviluppo di alcuni monitor aggiuntivi. Come anticipato nel Capitolo 5 non tutte le tipologie di monitor proposte in figura 5.2 sono state implementate. Per estendere maggiormente le funzionalità di monitoring tutte le diverse classi monitor proposte verranno implementate.

Un altro di questi sviluppi riguarda uno studio più approfondito delle tecniche decisionali al fine di migliorare, diminuendolo, il sovraccarico complessivo generato dal framework ARGO. Al momento la strategia di ricerca dei punti operativi consiste in una ricerca lineare; nel caso peggiore è necessario scorrere tutta la lista per trovare il punto operativo corretto o per poter dire che non esiste un punto utile. Per questo motivo diverse strutture di dati o modalità di ricerca verrano vagliate.

Un ulteriore sviluppo futuro consiste nella fase di testing delle varie applicazioni illustrate nel Capitolo 8 utilizzando un metodo alternativo per il calcolo del valore di un Working Mode. Questa operazione è attualmente basata su un'analisi delle risorse usate dall'applicazione; Il valore del Working Mode risulta essere una metrica obiettiva in quanto è calcolato allo stesso modo per tutte le applicazioni. Una delle limitazioni di questa tecnica è che, nel caso di applicazioni in cui l'aumento di risorse non implichi sempre un miglioramento delle metriche dell'applicazione, il

valore del Working Mode perde parte del suo significato. Un esempio è dato dalle applicazioni le cui prestazioni sono limitate dall'I/O, come ad esempio l'applicazione PatternMatching, dove un indicazione del valore del Working Mode riferita al throughput rifletterebbe meglio le preferenze dell'applicazione stessa. Per questo motivo, abbiamo studiato e definito una versione alternativa dell'algoritmo del calcolo del Working Mode che cerca di generalizzare la versione precedente. Questa nuova modalità permette di avere un valore del Working Mode fornito dall'applicazione stessa e non dal suo uso di risorse. Lo svantaggio dato dall'adozione di questa soluzione è la perdita di obiettività del valore dei Working Mode aumentando la difficoltà nel trovare una relazione tra Working Mode di applicazioni diverse. Una preliminare definizione e descrizione dell'algoritmo a cui si fa rifemento può essere trovato nell Appendice A

Per quanto riguarda la valutazione del framework stiamo attualmente lavorando per supportare l'interfacciamento della versione OpenCL dell'applicazione Stereo Matching con ARGO. Questa applicazione è stata sviluppata per sfruttare le caratteristiche dell'architettura hardware definita dal progetto 2PARMA e quindi potrebbe fornire informazioni interessanti riguardo l'utilizzo del framework con architetture embedded. Inoltre, stiamo progettando di trasferire alcune suite di benchmark come ad esempio la suite PARSEC [50] per poter testare il nostro framework con diversi gruppi di applicazioni e carichi di lavoro.

Appendici

Appendice A

Algoritmo per il calcolo dei Working Mode

Come anticipato nel Capitolo 9, una delle possibili evoluzioni di questo lavoro di tesi consiste in una valutazione alternativa del valore dei Working Mode.

Abbiamo già formalizzato i passi della nuova versione dell'algoritmo che è possibile visualizzare nell'Algoritmo 2. La differenza rispetto alla versione precedente (Algoritmo 1) è che vengono fornite in ingresso tutte le metriche applicative e i loro rispettivi pesi. La modalità di calcolo del valore del working mode rimane uguale ma, invece di usare soltanto le metriche relative all'uso di risorse ora vengono utilizzate quelle dell'applicazione.

Algoritmo 2: Calcolo del valore dei Working Mode

Input:

- Lista di Operating Point ($opList$) etichettati con il loro cluster-ID
- Lista di cluster-ID ($clusterIDs$)
- Lista delle metriche dell'applicazione ($ASMetrics$)
- Lista dei pesi delle metriche dell'applicazione ($weights$)

Output:

- Lista dei valori di ogni Working Mode ($wmValues$)

```
1  begin
2      foreach clusterIDs id do
3          max["cpuUsage''][id] = max(op_x["cpuUsage''], op_x ∈ opList[id])
4          max["memUsage''][id] = max(op_x["memUsage''], op_x ∈ opList[id])
5          foreach ASMetrics asm do
6              max[asm][id] = max(op_x[asm], op_x ∈ opList ∧
7                  op_x["cpuUsage''] ≤ max[''cpuUsage''][id] ∧
8                  op_x["memUsage''] ≤ max[''memUsage''][id])
9              min[asm][id] = min(op_x[asm], op_x ∈ opList ∧
10                 op_x["cpuUsage''] ≤ max["cpuUsage''][id] ∧
11                 op_x["memUsage''] ≤ max["memUsage''][id])
12             swing[asm][id] = max[asm][id] − min[asm][id]
13         end
14     end
15     foreach ASMetrics asm do
16         totalSwing[asm] = max(op_x[asm],op_x ∈
           opList) − min(op_x[asm],op_x ∈ opList)
17     end
18     foreach clusterIDs id do
19         wmValues[id] = 0
20         foreach ASMetrics asm do
21             swingRate[asm] = swing[asm][id]/totalSwing[asm]
22             wmValues[id]+ = weights[asm] ∗ swingRate[asm]
23         end
24     end
25 end
```

Appendice B

Articolo proposto

Questo appendice include l'articolo sottomesso alla conferenza CODES+ISSS che si svolgerà dal 7 al 12 ottobre 2012. I risultati riguardo l'accettazione di questo articolo verranno pubblicati nei prossimi mesi.

ARGO: An Application-Specific Run-Time Goal Management Framework for Embedded Multiprocessor Architectures

[1]Vincenzo Consales, [1]Andrea Di Gesare, [1]Gianluca Palermo, [1]Vittorio Zaccaria, [1]Cristina Silvano, [2]Chantal Ykman-Couvreur

[1]Dipartimento di Elettronica ed Informazione - Politecnico di Milano, Italy
[2]IMEC, Belgium
{vincenzo.consales, andrea.digesare}@mail.polimi.it
{gpalermo, zaccaria, silvano}@elet.polimi.it
ykman@imec.be

ABSTRACT

Multi-Processors Systems on Chip (MPSoCs) have been increasingly more popular. These architectures are expected to become a standard in the next few years and yet there are still a lot of challenging issues. Among them, system designers have found more and more difficulties to design systems able to allow applications developers to exploit properly all the underlying hardware provided by new architectures. Moreover, the lack of run-time optimization strategies tends to lead to a waste of power and/or performance: there are too many variables to consider to make such decisions at design time and just by experience. Letting an application achieve its goal with respect to the resource allocated and to power constraints it's of the utmost importance.

This paper describes a new framework aimed at simplifying applications design flow enabling developers to define with a minimum effort their own applications goals and making achieve them. Our design flow mainly is composed of two different phases: the former is made at design-time and consists in a detailed characterization of the application through Design Space Exploration (DSE), the latter consists of run-time techniques, based on the information gathered at design-time and from the current environment.

Categories and Subject Descriptors

D4.8 [**Operating Systems**]: Performance – *Monitors*; C.3 [**Computer Systems Organization**]: Special-Purpose and Application-based Systems – *Real-time and embedded systems*

General Terms

Performance, Design, Management

CODES+ISSS'12, October 7-12, 2012, Tampere, Finland

1. INTRODUCTION

Nowadays, multiprocessor architectures can be considered the standard for computing systems. These architectures, composed of multiple homogeneous and heterogeneous processing units tightly interconnected with memory controllers and storage elements, offer the high performance required by today's applications in the embedded, general purpose and high-performance computing worlds. Also on such type of architectures, the first era of computing was mostly devoted to find the highest speed, the highest clock and in general the highest performance of a system. What became clear after few years is that it is not possible to achieve performance without drawbacks. Power and energy limits are becoming more and more important in the design of a system. These concepts are major issues in the embedded systems field: it is not rare anymore to find multi-processors even in small embedded systems or in portable devices. As an example, these systems need to achieve their own goals taking in account the energy provided by their batteries. That's why we are witnessing a change of approach on computing. For the same reasons given above, applications are switching from a performance-oriented behavior to goal-oriented one: developers are becoming conscious of concepts such as power limits and energy consumption and want their application to reach their goals while respecting those limits. While the targeted problem is clear, it is still difficult to create such systems.

In this paper, we will exploit design-time and run-time concepts to create an alternative and effective way of self-aware computing [1]. The key concepts of this framework, called ARGO[1] are: design-time analysis, monitoring of the application Quality of Service (QoS) and run-time decision-making.

Our framework aims to determine the relation between application parameters and their metrics (such as throughput, power consumption, accuracy) at design-time. These information are used to create a model of the application, char-

[1]The name of this framework, ARGO, it's not just an acronym. Its name has been borrowed by Greek mythology. Argo was the ship on which Jason and the Argonauts sailed to retrieve the Golden Fleece. As that boat was a means for achieving the Golden Fleece (their goal), this framework aims to let applications reach their goals too.

acterizing its tunable parameters (also known as dynamic knobs [2]) and their effect on the overall performance of the application. Once such model has been created, it can be used to tune applications dynamically, letting them adapt their behavior according to the available resources and current constraints.

In this work, we used a two-layers approach by considering both a system-wide resource manager, managing the resource sharing in the platform, and an application-specific manager, that eventually manages the application parameters. Following this idea, we called *Operating Points* the configurations of application-specific parameters that can be changed by the application-specific manager without requiring the interaction with the resource manager, while the *Working Modes* are different configurations of resource requirements needed by the application that have to be selected by the resource manager depending on the system state.

The contributions of this paper are the following:

- Introduce an efficient DSE methodology to find an optimal set of Operating Points and Working Modes;
- Decouple run-time management tasks between design-time and run-time, combining run-time adaptivity with low overheads;
- Provide a distributed approach of run-time management, enabling to take decisions where there is the maximum knowledge of the problem;
- Allow developers to define a set of goals for their applications and to enable the achievement of them without taking care of the run-time techniques to follow.

The remainder of this paper is organized as follows. The next section provides an overview of the related work. Section 3 introduces the target architecture for this framework, explaining the reasons behind this solution. Our proposed methodology is described afterwards in Section 4 while some use-cases and experimental results are presented in Section 5. Section 6 concludes, providing some ideas for further improvements.

2. RELATED WORK

In the last years, different techniques have been proposed in the field of autonomic, adaptive, self-aware computing. They are most of the times very specialized works, focusing on particular problems or systems. There are few generic systems able to provide in different ways a solution to the self-aware demand. In this section we will briefly provide some of them. The first example is an OS-level autonomic system, AQuoSA [3, 4]. It consists of a set of non-invasive changes on the top of the Linux kernel aimed at providing the best schedule for real-time applications and in general for all time-sensitive ones (e.g. multimedia tasks). In their work, tasks ask for CPU quotas (CPU bandwidth in their paper) that the scheduler provides. When some bandwidth is unused, the scheduler reclaims it and gives it to one or more tasks. A different approach on adaptive computing has been proposed in [5]. The authors use a PID controller to provide a feedback control system able to keep the throughput within a certain rate reducing overallocation of resources. Another interesting solution, used as inspiration within our paper, is the one proposed in [6, 7]. In their work, the authors combine design-time and run-time techniques in order to instruct a run-time manager to manage the allocation of resources. Differently from our solution, the run-time manager is only one (system-wide) and it is intended to optimize energy consumption while respecting application deadlines. Their design-space-exploration strategy follows the same key concepts of Respir [8], a design-space methodology that provides a powerful implementation of Design Space Exploration and Design of Experiment techniques. This strategy is the starting point for the design phase of our work. The same authors of Respir have made a step forward on their previous work, developing a framework [9] for run-time management based on their design-time solution. It is based on queue theory and aims to improve performance of applications on a matter of average response time. Even in this case the run-time manager is only one and system-wide. It provides only means to change the parallelisation of the application (if available). Last but not least, there is the work presented in [2, 10–12]. They use the concept of heartbeat to provide a way to monitor applications latency and throughput. Their framework, SEEC, uses a run-time manager with different levels of adaptation. They vary from a simple closed-loop control scheme to a more refined machine learning manager. However, this solution lacks of design-time techniques. Often its functioning is guaranteed to be fast and reliable only when a model of the application is provided but there are no hints on how to create such a model.

There is not yet available a complete framework capable of handling all the problems for the management of self-aware multiprocessors systems. In this paper we present ARGO, our framework that is inspired by most of these state of the art solutions and tries to improve them coupling design-time and run-time approaches in order to achieve better results.

3. TARGET ARCHITECTURE

The reference target architecture of this paper (see Figure 1) is a shared-memory multi-processor system. On this platform there is a Run-Time Resource Manager (RTRM) as described on [13]. The tasks of the Run-Time Resource Manager are split up in two different entities: a System-Level Run-Time Resource Manager (SYS-RTRM) and a set (one for each application) of Application-Specific Run-Time Managers (AS-RTM). While the former focuses on the applications' resources allocation, the latter is responsible of dynamically setting applications parameters, in order to let them reach their goals. Each AS-RTM relies on two main components of the framework: an Operating Points Manager (OP Manager) and a set of Monitors. Their functions and uses will be explained in Sections 4.1 and 4.3.

As shown in Figure 1, this architecture relies on two different knowledge-bases: *Operating Points* (OPs) and *Working Modes* (WMs).

Operating Points represent a concept closely related to the application domain. They consist of a basic structure that describes the different combination of configuration parameters (if any) for the current application. They contain both the parameters of the application and the profiled metrics observed at design-time, such as power consumption and throughput. These metrics can be used to derive a model of the application behavior under different configuration sets. This set of Operating Points represents part of the

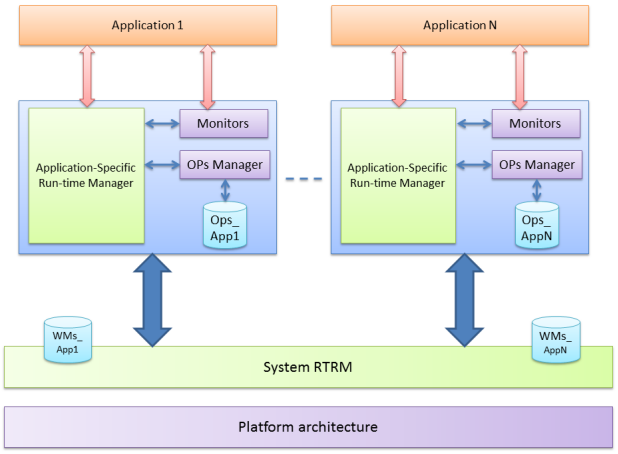


Figure 1: Target architecture

knowledge-base used by our Application-Specific Run-Time Manager.

Unlike Operating Points, *Working Modes* have a different domain than the application one. They indeed represent a set of scenarios of the application, but they have different metrics and a different use. Working Modes represent the knowledge needed by the System-Level Run-Time Resource Manager to decide how many resources to allocate for a determined application. Each Working Mode is tied to a set of Operating Points and represents the maximum resource usage (CPU usage and memory usage) of that set of Operating Points. Each working mode has a specific value, which represents the application preference of running on a specific Working Mode instead of another. The number of Working Modes is by definition less or at most equal than the number of Operating Points.

This paper will focus only on the application-side domain, giving some information on the system-level domain when necessary.

4. PROPOSED METHODOLOGY

ARGO aims at determining the relation between application parameters and their metrics (e.g throughput, power consumption, accuracy) at design-time. Figure 2 shows the design flow of our framework. The first step consists of a definition of the application design space, choosing which parameters and metrics are needed to be tested and valuated. Then, the exploration engine runs the application with all the combination of parameters defined and provides some information about the program and its trade-offs in term of metrics values between different configurations of the parameters. This information is used to create a model of the application, characterizing its tunable parameters and their effects on the overall performance of the application.

Once this phase is concluded, the data gathered from design-time is used to build two different knowledge-bases: the set of Operating Points and Working Modes, as will be explained in Section 4.2.

Once such models have been created, they can be used to let the SYS-RTRM resolve the problem of allocating resources to applications [14,15] while each AS-RTM will tune-up applications dynamically, letting them adapt their behavior according to the available resources and current constraints (e.g. QoS, power consumption).

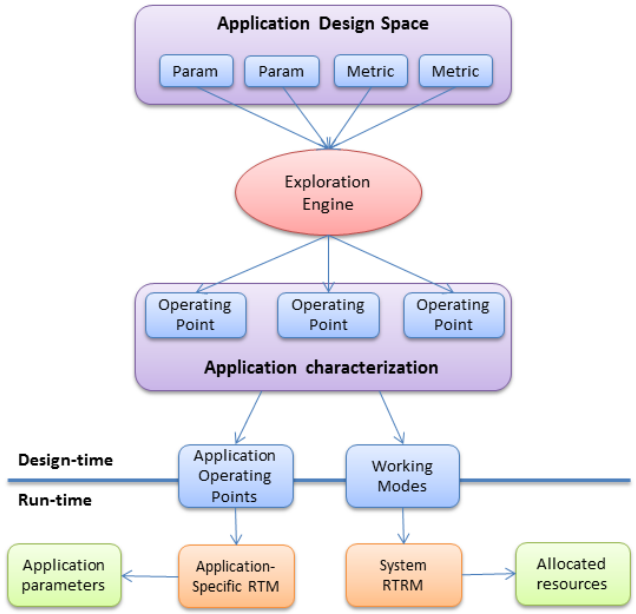


Figure 2: ARGO Design Flow

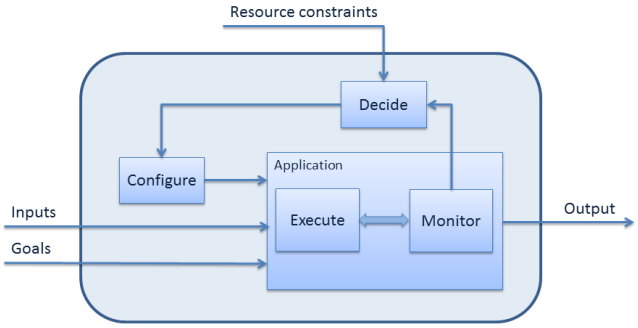


Figure 3: Execution and Control Loop of an Application

The System-Level Run-Time Resource Manager only needs to know which are the different uses of resources of an application and the preference of a Working Mode respect to another (quantified by a *value*). It will then decide for each application the best Working Mode that is currently feasible. This approach for the resource management is the same used in [15]. This decision sets explicitly the maximum amount of resources an application can use. The application (therefore also the AS-RTRM), is notified of such decision. The Application-Specific Run-Time Manager will then take care only of the decision related to the application-specific domain such as setting internal parameters, with a resource constraint given by the SYS-RTRM.

Figure 3 shows the execution and control loop used on our framework for the applications. It is a classical control loop for autonomic computing [1] in which the inputs of the applications are provided by the user/environment together with the performance/QoS goals. The other inputs, representing the resource constraints, is provided by the System-Level Resource Manager. These data will be used by the

Decide block in order to determine the right setting for the application parameters. The *Configure* phase will set then the Operating Points of the applications and let the *Execute* block run it with those parameters. Then, while running, *Monitors* will observe the status of the goals of the applications, giving the right input for the *Decide* block that will select the configuration parameters for the next execution phase.

4.1 Monitors and Goals definition

One of the enabling features of autonomic computing, is the ability to observe and analyze the internal status of an application [1, 16, 17]. We developed a monitoring suite for our framework, able to potentially fit different needs. In ARGO we preferred to focus on high-level monitoring metrics (e.g latency, throughput) instead of architecture-specific ones (like Instructions Per Cycles (IPC)). It has been already proved that low-level metrics can be misleading in multiprocessors systems [18]. Moreover, because higher-level goals are not dependent directly on the underlying hardware, we can provide a more generic framework that allow to directly compare metrics between different architectures. However, in case of particular needs, it is possible to easily extend these monitors with plugins, allowing to use them in more specific situations. These monitors have been developed in C++, with a strong object-oriented approach.

The structure of our monitors is modular. It consists of a base class *Monitor* that includes all the functionality needed by all the monitors, such as functions to initialize them, declare goals and check them. This main class is straightforward to use and allows to declare multiple goals on simple sets of data. As specialization of the Monitor class, we defined different specific monitors that satisfy the most common needs. According to our own experience, we defined several specialized monitors that can be grouped into two main classes: *application* and *system-resource* oriented monitors. In the first set there are all those monitors strictly related to the application behavior/performance (such as *TimeMonitor*, *ThroughputMonitor* and *QualityMonitor*), while in the second one there are all the platform specific monitors (such as *MemoryMonitor*, *CpuMonitor* and *TemperatureMonitor*). These classes inherits all the functionality from the basic Monitor class and their definition need only to provide the new monitor-specific features.Our monitoring suite abstracts on an higher level the concept of sensor data. Figure 4 shows a simple outline of Monitors' architecture.

The focus of the monitoring suite is not only on the gathered data, but also on the concept of goal. A goal is a predicate on a metric of the application and it describes a range of values admitted for that metric. These values are usually given as result of statistical function on a sliding window of data, not just on a single collected value. So far, our framework supports the use declaration and estimation of goals on statistics such as average, maximum, minimum value and variance on a sliding window of data. Moreover, each goal can be composed of more targets, such as upper bounds and lower bounds in order to identify a target interval.

As it is usual, monitors can be used as well as mere observer without requiring any goal definition. They can manipulate directly gathered data through statistic functions. A sliding window of variable size is provided in order to allow a fine grain decision on the level of statistics to collect, whether global ones or local ones depending on the selected window size.

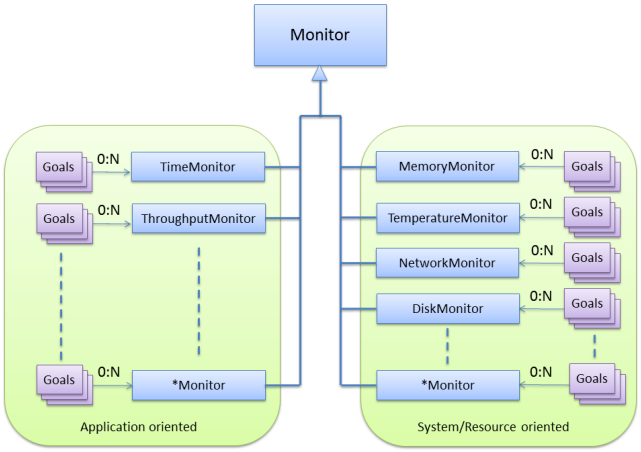


Figure 4: Basic monitors architecture

Following the previous description, we defined two operative mode for the monitors: the *goal-mode*, when all the goal features (e.g. the sliding window) are used, and the *sensor-mode*, when only the basic functionality of them are used (merely utility functions such as simple timers or getters for specific values).

A peculiar characteristic of our monitoring framework is its versatility. It has been implemented to be useful both during Run-Time and Design-Time. While during the run-time execution of the application, it is useful to check its status towards one or more goals and to react correspondingly (making the application autonomous), during the design phase, monitors can be used to profile the application and emit metric data without the need of writing extra code by the developer. This strategy is the one we used to perform the application profiling during the Design Space Exploration phase.

Declaring goals can be pretty easy and straightforward. An example below shows how to declare a goal of type throughput, tied with the metric *Frames/sec* of the application. In this specific case, only one target has been declared: an average throughput of more than 25 frames per second.

```
ThroughputMonitor t;
t.newGoal(DataFunctions::Average,
          "Frames/sec",
          ComparisonFunctions::Greater, 25);
```

As it is easy to declare applications goals, it is also simple to check their status. In particular, a *checkGoal* function takes care of checking the status of the application towards that goal. It returns a boolean value indicating whether the goal has been achieved or not, and a list of relative errors, one each target of the goal (e.g. in the example above, there is just one target so only one relative error will be provided).

4.2 Design-Time Analysis

In this section, we are going to present the necessary steps we follow to generate the knowledge-base needed by the Run-Time management infrastructure both at system and application-level. As already discussed before, it is composed by the list of Working Modes and Operating Points. Figure 5 presents a brief summary of the necessary steps.

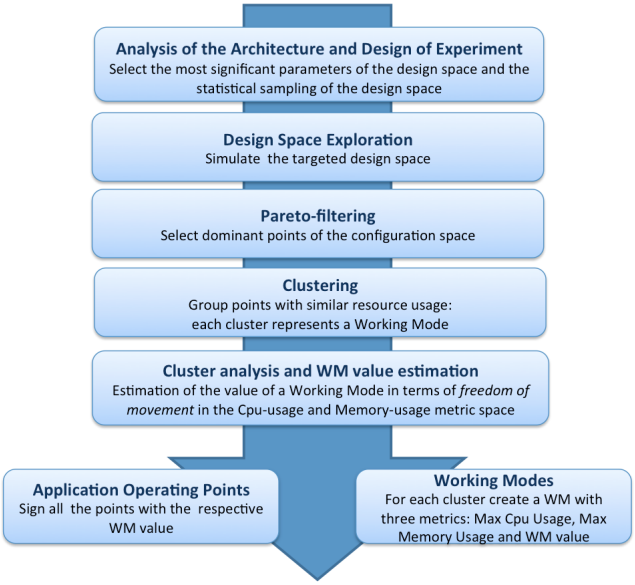


Figure 5: Overview of the design-time steps

One of the key points of this part of the work at-design time is a thorough use of Design Space Exploration (DSE) and Design of Experiments (DoE) techniques, in order to achieve a characterization of the application and all its tunable parameters.

A Design of Experiment consists on the planning of the layout of the design points to select for the design space. These points are the ones that will take part on the simulations. As remarked in [8] and in [19], this phase is really important. Statistical sampling of the design space can affect dramatically simulation time with a negligible loss of significance of the final result. A correct choice of a small subset, even just the 5% of the points of the design space, together with the right Design Space Exploration techniques can characterize with more than 95% of precision the overall system.

After the selection of the most appropriate DoE plan, the Design Space Exploration phase runs all the needed simulations. The selection of the most appropriate DSE techniques depends mainly on the size of the design space. Since the faced problem is multi-objective, consisting in the minimization (or maximization) of a set of application-level metrics (e.g. latency, throughput) and resource usage (e.g. memory and CPU), it doesn't lead to an unique solution. The output of this phase is a set of configurations visited during the design space exploration. The amount of points can be large and eventually needs to be filtered depending on the objectives through *smart-Pareto filters* [20]. The usage of smart-Pareto filters helps in reducing the set of points, maintaining only the most interesting ones. The remaining configurations are those that will be later transformed in the application Operating Points.

A further analysis of the points is necessary to generate the Working Modes useful to the SYS-RTRM. It consists of a clusterisation of the points according to resource usage. The usage we adopted refers to CPU and Memory since the used SYS-RTRM was able to manage those shared resources. Using a K-means clusterisation technique [21] we achieve a first

Algorithm 1: Working Modes value estimation

Input:
- Operating Points list ($opList$) tagged with cluster-IDs
- Cluster-IDs list ($clusterIDs$)
- Application-Specific Metrics list ($ASMetrics$)
- Weights for the metrics ($weights$)

Output:
- Working Modes values ($wmValues$)

```
1  begin
2      foreach clusterIDs id do
3          max["cpuUsage"][id] = max(op_x["cpuUsage"],
4              op_x ∈ opList[id])
5          max["memUsage"][id] = max(op_x["memUsage"],
6              op_x ∈ opList[id])
7          foreach ASMetrics asm do
8              max[asm][id] = max(op_x[asm], op_x ∈ opList ∧
9                  op_x["cpuUsage"] ≤ max["cpuUsage"][id] ∧
10                 op_x["memUsage"] ≤ max["memUsage"][id])
11             min[asm][id] = min(op_x[asm], op_x ∈ opList ∧
12                 op_x["cpuUsage"] ≤ max["cpuUsage"][id] ∧
13                 op_x["memUsage"] ≤ max["memUsage"][id])
14             swing[asm][id] = max[asm][id] − min[asm][id]
15         end
16     end
17     foreach ASMetrics asm do
18         totalSwing[asm] = max(op_x[asm], op_x ∈ opList) −
19             min(op_x[asm], op_x ∈ opList)
20     end
21     foreach clusterIDs id do
22         wmValues[id] = 0
23         foreach ASMetrics asm do
24             swingRate[asm] =
               swing[asm][id]/totalSwing[asm]
25             wmValues[id] += weights[asm] ∗ swingRate[asm]
26         end
27     end
28 end
```

classification of points that will be further used to determine Working Modes and their values. The number of clusters, hence of Working Modes, is a choice made by the designer. It should be a good trade-off between an high number (allowing to have a fine grain resource allocation) and a low one (providing a lower flexibility to the SYS-RTRM). It has been determined that a number smaller than 10 Working Modes is optimal since it can be managed by the System-level Run-Time Resource Manager without problems (large overhead) also considering a multi-application context [14].

Once the clusterisation phase has taken place, we determine the *value* to assign to each Working Mode as needed by the SYS-RTRM [15]. This will be done by using the idea of freedom of movement in the application-specific parameter space available within the WM through the Operating Points. Algorithm 1 shows a pseudo-code for the WMs value estimation algorithm.

The algorithms starts receiving the list of Operating Points after the clustering phase, the list of application-specific metrics (and weights) that are useful to evaluate the application performance. The first part (lines 3–6) consists of finding the maximum CPU and Memory usage in each cluster. Remembering that each cluster will be associated with a WM, those values will be used to understand what are the resource requirements for each WMs. Moreover for each cluster (WMs), we determine also (lines 7–14) the maximum and minimum values for the application-specific metrics. Those values have been extracted considering all the operating points within the maximum resource usage values

assigned to WMs[2], and represent the freedom of movement (*swing*) within the application-specific metrics space. Indeed, playing with the operating points the AS-RTM is able to move the application within the metrics space. Hence, we determine the total swing between the maximum and minimum value for each metrics (lines 17–20) within the full set of Operating Points for normalization purposes (line 24). The last step of the algorithm (line 25), consist of the actual WMs value calculation obtained by weighting the normalized swing of each application-specific metric, according with the designer preferences.

At this point the design phase is ended. The last step consists only of the creation of the list of Operating Points for the AS-RTM and the list of Working Modes for the Resource Manager. Using the appropriate format, in the former list there will be both the configuration parameters and the metrics for that Operating Point, while in the latter only the WM-value, CPU usage and Memory usage will be included. When the Resource Manager will select a Working Mode for an application, the Application-Specific Run-Time Manager (and its related component: the Operating Points Manager) will select the right Operating Point according to the WM value and the resource constraints given by that Working Mode.

4.3 Operating Points Manager

The Operating Points Manager is the component of ARGO responsible for the management of the Operating Points found for an application. It uses the data collected at design time to create a versatile and generic structure used to store all the information regarding Operating Points. One of the key features of the OP Manager, is the ability to declare a priority between metrics, allowing to order Operating Points according to some of them. The Operating Points Manager is able to select the best Operating Point (according to the priorities given) that satisfies all the constraints received as input, by applying a *Pareto-optimization* [22]. For example, considering a power-aware application with a QoS constraint on latency, the OP Manager will select the OP with the least power consumption that (according to design-time analysis) is able to satisfy the latency constraint requested by the application. The decision task is performed considering also the resource constraints notified by the System-Level Resource manager each time the available resources change.

4.4 Application-Specific Run-time Manager

Lying on the top of all the basic blocks described before, there is the Application-Specific Run-time Manager. Its targets is to use the knowledge provided by the Monitors and the Operating Points Manager blocks to make autonomous decisions for the application. The AS-RTM uses the features of the Monitors to check the status of the application regarding the achievement of its goals, while it uses the OP Manager to request Operating Points satisfying the constraints posed by the SYS-RTRM on the resource usage and by the AS-RTM itself to drive the search.

Each application can either have an AS-RTM or not. In the former case, the AS-RTM will manage autonomously all the application parameters, while in the latter case the developer will have to take care of the data provided by the Monitors and the OP Manager blocks to take manual decision. However, an hybrid approach is also possible. Developers can rely on the AS-RTM while making occasionally manual decisions if needed. The AS-RTM has two special peculiarities that make it so useful. The first is that it is able to automatically notify the Resource Manager when, given the current resource and QoS constraints, the application is not able to find a feasible Operating Point. This notification consists of a description of how much additional effort (hence resources) is needed to reach the desired goals. With such notification, the system-wide resource manager could change its resource allocation policies in order to guarantee the right allocation of resources for the application. The second feature is the AS-RTM adaptability to discrepancy between the model provided by the design-time profiling and the actual performance of the application. The AS-RTM adapt its constraints to manage proportional[3] variation of the performances caused for example from frequency scaling or the wearing of components. In particular here is an example to help understanding the approach. Let's consider a goal requiring to reach at least a specific value for a metric, for example a maximum latency of $20ms$. The AS-RTM then consults the OP Manager asking for a point respecting resource constraints plus the constraint on the latency metric. The OP Manager provides a point that is supposed to give $18ms$ of latency. Because of temperature issues on the platform, the frequency of the system is halved with respect to the one used during the design time exploration, causing an actual latency of 36ms. Monitors observe this behavior and notify the missed goal. Considering that the monitored performance have been halved, the AS-RTM requests again an other Operating Point with doubled value on the constraint: maximum latency smaller than $10ms$. If the new selected configuration is able to reach its goal, the application continues its execution on that configuration, otherwise the AS-RTM will adopt the same approach in a greedy way. When the frequency of the system rises again putting back the system on the "profiled" status, the same approach will be used by the AS-RTM switching to the original Operating Point.

5. EXPERIMENTAL RESULTS

In order to validate ARGO, we ported a Stereo Matching application in our framework by executing it on Intel Core i7-2620M at 2.7GHz with 4MB L3-Cache. The CPU is based on the Sandy Bridge architecture, has two cores and offers Hyperthreading to handle 4 threads at once (4Hw Threads). The target application takes as input two images representing the same subject but taken from two different cameras in the same line of sight. The application calculates a disparity map, representing the depth of the object forming the picture. A description of the application and its solving algorithm can be found in [23].

The application uses intensively the processors available on the system and has been implemented by using OpenMP. Moreover, we implemented the basic algorithm [23] by exposing some of the application parameters in order to give the possibility to change it. An application configuration is composed by four parameters extracted from [23]:

[2]In fact, if a Working Mode is chosen, nothing forbids the application to go on an Operating Point of another cluster unless it is from a cluster with higher resource usage.

[3]We refer to proportional changes not only referring to linear modifications of the performance but more in general to changes that do not affect the operating points order

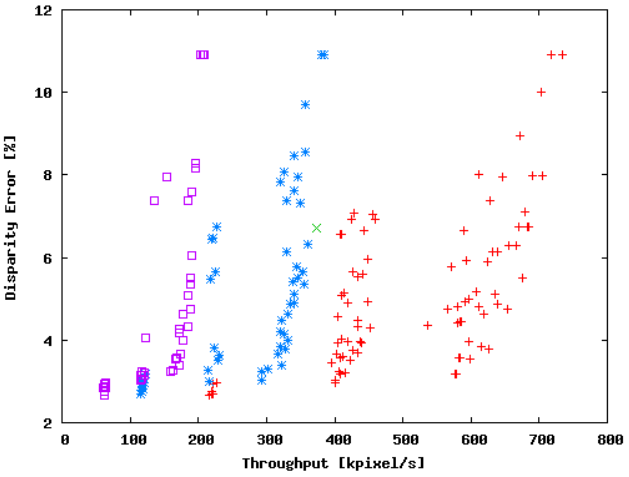


Figure 6: Results after the clustering phase

- *Number of threads*: It is the maximum parallelism of the application. Considered values are 1, 2, 4, 8 threads.
- *Maximum arm length*: It refers to the maximum value of the arm length for the crosses used by the algorithm to build the support regions. The range of analyzed values is between 1 and 18.
- *Confidence*: It represents a threshold value used when building the support region to determine if the analyzed pixels are within the same regions or not. The range of values is between 14 and 64 considering a step of 5;
- *Hypo step*: It is the sampling step used by the algorithm to verify the disparity hypothesis. Considered values are 1, 2 and 3;

The design space is composed of 2376 possible combinations of configuration parameters.

We enhanced the Stereo Matching by using our framework to monitor the application performance at run-time in terms of throughput, expressed in KPixel/sec, and CPU and memory usage. Moreover at design-time we introduced an additional metric called disparity error, that is inversely correlated to the quality of the output image. Higher is the disparity error, lower is the output quality. This values at design time is evaluated by verifying the output image with a truth version of the disparity map. The disparity error verified at design-time (and averaged on a large number of data-set) will be used at run-time to forecast the quality of the output for the selected application configuration.

We applied the design-time methodology presented in Section 4 by adopting a randomized design of experiment and a Multi-objective Simulated Annealing algorithm for the exploration phase. Both the optimization and the Pareto filtering steps included all the application metrics described before, finding a total of 241 optimal operating points to be managed by the AS-RTM. For the clustering phase, we decided to map the operating points into 4 sets (later transformed in Working Modes), exposing to the SYS-RTRM a good trade-off between flexibility and overhead for the given architecture. Figure 6 shows the results of the clustering phase on the resource usage metrics, in the Throughput and Disparity Error space.

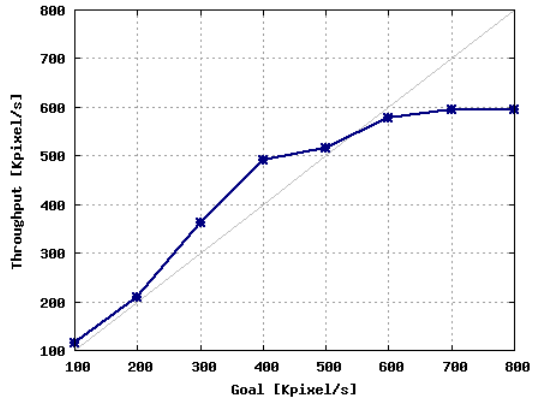
For validating the usage of the ARGO framework to reach the application goal, we varied the throughput goal between 100 Kpixel/s and 800 Kpixel/s and we observed the throughput and disparity error associated to the configuration selected by the AS-RTM (see Figure 7). Figures 7.(a) and 7.(c) show the results of this analysis when we limited the available resources to 2 Hw threads. Figure 7.(a) shows that increasing the goal up to 500 Kpixel/s, the AS-RTM is able to provide to the application a configuration to satisfy the required throughput. Increasing the value over this threshold, the goal is not more met. In fact, starting from a request of 600 Kpixel/s, the AS-RTM is able to provide a configuration that is close to the target but without reaching it. Moreover, it is possible to see in Figures 7.(c) that while up to 500Kpixel/s the disparity error remains very small, to further increase the throughput the application quality should be sacrificed. Similar behavior can be observed in Figures 7.(b) and 7.(d) when the number of available resources rise up to 4 Hw threads. In fact, the application throughput reaches the goal up to 700 Kpixel/s, paying only in the last case a significant quality reduction on the output image.

Finally, to give more details on the effectiveness of the proposed AS-RTM, we run the application with a total workload of 100 frames and we force some run-time changes (both to goal values and system frequency) to test our framework's reactivity (see Figure 8). In detail, the application starts with the system's processor speed of 2700 MHz and a goal on the throughput of 400K Pixel/s and 500 KPixel/s as lower and upper bounds. After around 4 seconds we switch the frequency to 1600 MHz, hence simulating a possible frequency scaling due to heat issues, and we keep that frequency for other 5 seconds before switching to the original one. Then, when the 65% of the workload has ended, we change the goal (200 KPixel/s and 300 KPixel/s as bounds) and we repeat the same frequency change as done in the first phase.

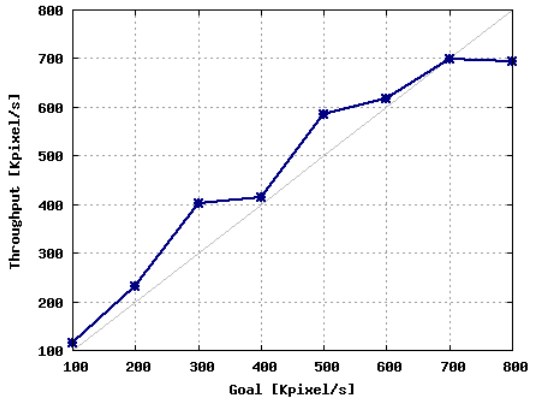
Figure 8 shows a graph of the execution of the test where the time elapsed is on the x-axis. While in Figure 8(a) on the y-axis there are the current throughput and the disparity error, in Figure 8(b) there is the AS-RTM overhead.

The interesting part of the execution is when something happens either due to a system change (frequency in our test, around to second 4, 9, 20.5 and 25.5), or due to the goal change (around to second 16.5). As we can notice from Figure 8(a), the reaction of the system is pretty fast: changes of goals and of the outside environment are balanced within 1 or 2 elaborations of frames, keeping the system on a near-optimal Operating Point. When the frequency goes down (after second 4), the AS-RTM decides to change the application parameters, bringing back the throughput to the desired value at the cost of worsening the application accuracy. Then, when the frequency perturbation ends (after second 9), the original accuracy is restored. It is worth to notice as well, that the AS-RTM is able to select a more accurate configuration when the throughput demand (the goal) changes for a lower value (after second 16). In this case the application can run slower but more accurately keeping its throughput within the given constraints.

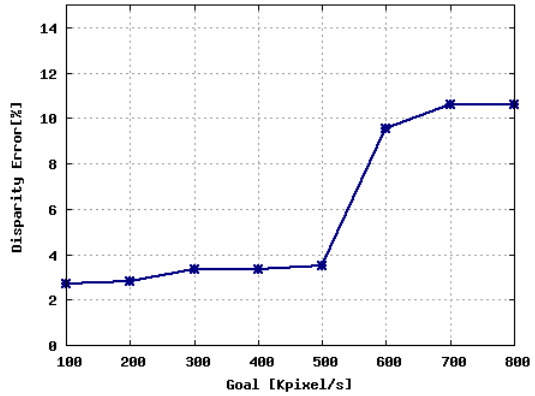
We used the change in frequency as an unpredictable source of change in performance with respect to what has been statically profiled. It is important to notice how the decision engine is able to estimate (in less than two step for the pro-



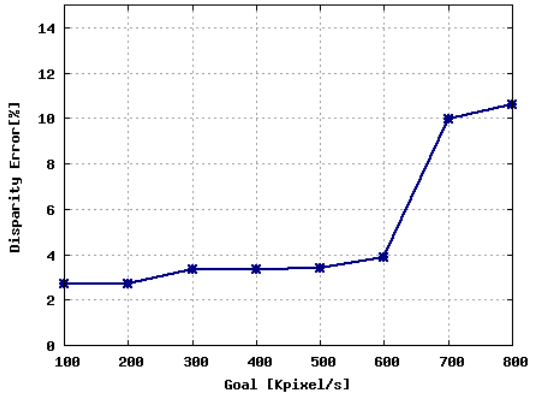
(a) Observed throughput using up to 2Hw threads



(b) Observed throughput using up to 4Hw threads



(c) Disparity error using 2Hw threads



(d) Disparity error using 4Hw threads

Figure 7: Observed throughput and disparity error by varying the throughput goal and the number of available CPU resources

vided experiment) the right Operating Point even when the model given from design-time analysis becomes incorrect.

Moreover, what is worth to notice is that for a run of around $33s$ the total overhead brought by the goal-checking and decision-making phases consist only of $1.31ms$, meaning less than 0.005%. For the case of selecting among 241 operating points, the resulting overhead is on average $13\mu s$ each frame and varies from a minimum of $8\mu s$ to a maximum of $113\mu s$ (see Figure 8(b)). Reducing the number of OPs, hence the search-space of the decision algorithm, we can lead (especially for the worst case execution) to significant lower overheads as well.

6. CONCLUSION AND FUTURE WORKS

This paper introduced our framework ARGO, a goal-driven application-specific run-time managing framework that make use of both design-time and run-time techniques. This strategy allows to offload most of the computation offline, making possible the use of a lightweight run-time manager.

A further evolution and validation of ARGO is in our plans. The first step to be taken is a deeper study of decision-making techniques in order to improve, hence decrease, the overall overhead given by our framework (especially in worst-case executions). Moreover, we are planning to port some benchmark suites such as PARSEC [24] to extensively test our framework with different sets of applications and workloads, considering also a multiprogrammed context. This will give us the possibility to better understand the possible interactions between the SYS-RTRM and the AS-RTM (eventually more than one) running on the platform.

7. REFERENCES

[1] An architectural blueprint for autonomic computing . Technical Report June, IBM, 2006.

[2] Henry Hoffmann, Stelios Sidiroglou, Michael Carbin, Sasa Misailovic, Anant Agarwal, and Martin Rinard. Dynamic knobs for responsive power-aware computing. *ACM SIGARCH Computer Architecture News*, 39(1):199–212, March 2011.

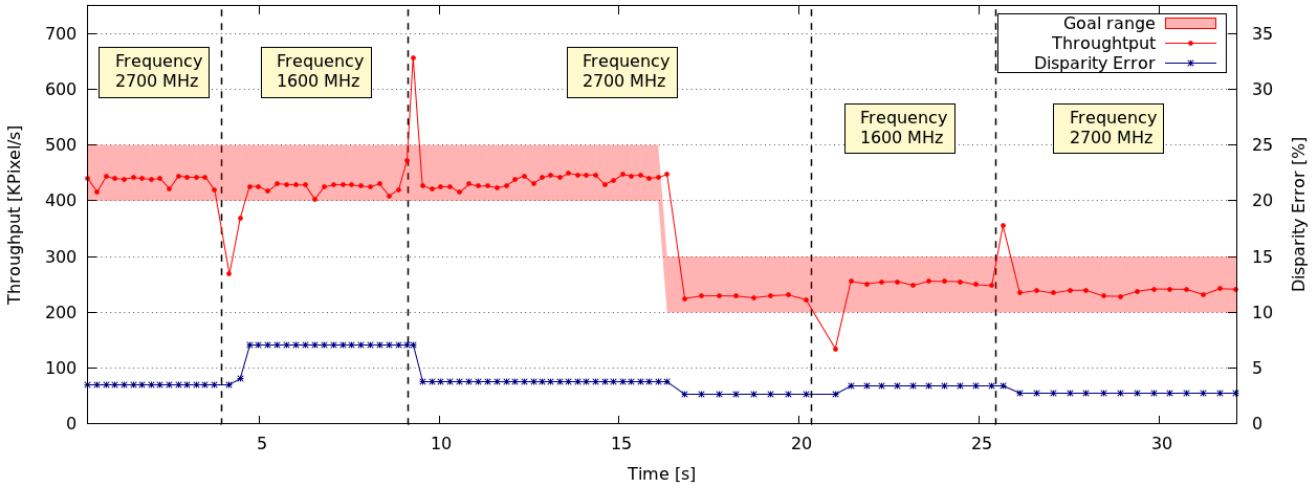
[3] Tommaso Cucinotta, Luca Abeni, Luigi Palopoli, and Giuseppe Lipari. A Robust Mechanism for Adaptive Scheduling of Multimedia Applications. *ACM Transactions on Embedded Computing Systems*, 10(4):1–24, November 2011.

[4] L. Palopoli, T. Cucinotta, L. Marzario, and G. Lipari. AQuoSA-adaptive quality of service architecture. *Software: Practice and Experience*, 39(1):1–31, January 2009.

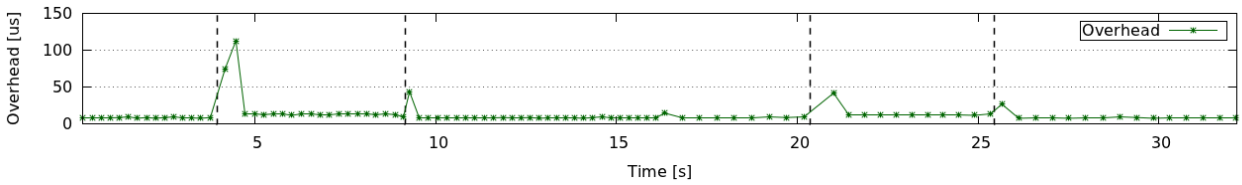
[5] G. Almeida, Rémi Busseuil, Luciano Ost, Florent Bruguier, Gilles Sassatelli, Pascal Benoit, Lionel Torres, and Michel Robert. PI and PID Regulation Approaches for Performance-Constrained Adaptive Multiprocessor System-on-Chip. *Embedded Systems Letters, IEEE*, 3(99):77–80, September 2011.

[6] Ch. Ykman-Couvreur, V. Nollet, Th. Marescaux, E. Brockmeyer, Fr. Catthoor, and H. Corporaal. Design-time application mapping and platform exploration for MP-SoC customised run-time management. *IET Computers & Digital Techniques*, 1(2):120–128, 2007.

[7] C. Ykman-Couvreur, P. Avasare, G. Mariani, G. Palermo, C. Silvano, and V. Zaccaria. Linking run-time resource



(a) Application Behavior



(b) AS-RTM overhead

Figure 8: Execution of the application Stereo Matching with different goals and frequency perturbations

management of embedded multi-core platforms with automated design-time exploration. *IET Computers & Digital Techniques*, 5(2):123–135, 2011.

[8] Gianluca Palermo, Cristina Silvano, and Vittorio Zaccaria. 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*, 28(12):1816–1829, December 2009.

[9] Giovanni Mariani, Gianluca Palermo, Cristina Silvano, and Vittorio Zaccaria. ARTE: An Application-specific Run-Time management framework for multi-core systems. In *2011 IEEE 9th Symposium on Application Specific Processors (SASP)*, pages 86–93. IEEE, June 2011.

[10] Martina Maggio, Henry Hoffmann, Marco D Santambrogio, Anant Agarwal, and Alberto Leva. Controlling software applications via resource allocation within the heartbeats framework. In *49th IEEE Conference on Decision and Control CDC*, pages 3736–3741. IEEE, 2010.

[11] Maggio, Martina and Hoffmann, Henry and Santambrogio, Marco D and Agarwal, Anant and Leva, Alberto. Decision making in autonomic computing systems: Comparison of approaches and techniques. In *Proceedings of the 8th ACM international conference on Autonomic computing - ICAC '11*, pages 201–204, New York, New York, USA, June 2011. ACM Press.

[12] Henry Hoffmann, Jonathan Eastep, Marco D. Santambrogio, Jason E. Miller, and Anant Agarwal. Application heartbeats. In *Proceeding of the 7th international conference on Autonomic computing - ICAC '10*, pages 79–88, New York, New York, USA, June 2010. ACM Press.

[13] Vincent Nollet, Diederik Verkest, and Henk Corporaal. A Quick Safari Through the MPSoC Run-Time Management Jungle. In *2007 IEEE/ACM/IFIP Workshop on Embedded Systems for Real-Time Multimedia*, pages 41–46. IEEE, October 2007.

[14] H. Shojaei, A.-H. Ghamarian, T. Basten, M. Geilen, S. Stuijk, and R. Hoes. A parameterized compositional multi-dimensional multiple-choice knapsack heuristic for cmp run-time management. In *Design Automation Conference, 2009. DAC '09. 46th ACM/IEEE*, pages 917 –922, july 2009.

[15] Ch. Ykman-Couvreur, V. Nollet, Fr. Catthoor, and H. Corporaal. Fast multi-dimension multi-choice knapsack heuristic for mp-soc run-time management. In *System-on-Chip, 2006. International Symposium on*, pages 1 –4, nov. 2006.

[16] Cornel Klein, Reiner Schmid, Christian Leuxner, Wassiou Sitou, and Bernd Spanfelner. A Survey of Context Adaptation in Autonomic Computing. In *Fourth International Conference on Autonomic and Autonomous Systems (ICAS'08)*, pages 106–111. IEEE, March 2008.

[17] J.O. Kephart and D.M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, January 2003.

[18] A.R. Alameldeen and D.A. Wood. IPC Considered Harmful for Multiprocessor Workloads. *IEEE Micro*, 26(4):8–17, July 2006.

[19] Erez Perelman, Greg Hamerly, Michael Van Biesbrouck, Timothy Sherwood, and Brad Calder. Using SimPoint for accurate and efficient simulation. *ACM SIGMETRICS Performance Evaluation Review*, 31(1):318–319, June 2003.

[20] Christopher A. Mattson, Anoop A. Mullur, and Achille Messac. Smart pareto filter: obtaining a minimal representation of multiobjective design space. *Engineering Optimization*, 36(6):721–740, 2004.

[21] J B MacQueen. Some methods for classification and analysis of multivariate observations. In L M Le Cam and J Neyman, editors, *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability*, volume 1, pages 281–297. California, USA, University of California Press, 1967.

[22] Peng Yang and Francky Catthoor. Pareto-optimization-based run-time task scheduling for embedded systems. In *Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, CODES+ISSS '03, pages 120–125, New York, NY, USA, 2003. ACM.

[23] G. Lafruit. Cross-Based Local Stereo Matching Using Orthogonal Integral Images. *IEEE Transactions on Circuits and Systems for Video Technology*, 19(7):1073–1079, July 2009.

[24] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC benchmark suite. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques - PACT '08*, pages 72–81. ACM Press, 2008.

Bibliografia

[1] An architectural blueprint for autonomic computing . Technical Report June, IBM, 2006.

[2] Henry Hoffmann, Stelios Sidiroglou, Michael Carbin, Sasa Misailovic, Anant Agarwal, and Martin Rinard. Dynamic knobs for responsive power-aware computing. *ACM SIGARCH Computer Architecture News*, 39(1):199–212, March 2011.

[3] Cornel Klein, Reiner Schmid, Christian Leuxner, Wassiou Sitou, and Bernd Spanfelner. A Survey of Context Adaptation in Autonomic Computing. In *Fourth International Conference on Autonomic and Autonomous Systems (ICAS'08)*, pages 106–111. IEEE, March 2008.

[4] Vincent Nollet, Diederik Verkest, and Henk Corporaal. A Quick Safari Through the MPSoC Run-Time Management Jungle. In *2007 IEEE/ACM/IFIP Workshop on Embedded Systems for Real-Time Multimedia*, pages 41–46. IEEE, October 2007.

[5] H. Shojaei, A.-H. Ghamarian, T. Basten, M. Geilen, S. Stuijk, and R. Hoes. A parameterized compositional multi-dimensional multiple-choice knapsack heuristic for CMP run-time management. *Design Automation Conference, 2009. DAC '09. 46th ACM/IEEE*, pages 917–922, 2009.

[6] Marc Geilen, Twan Basten, Bart Theelen, and Ralph Otten. An Algebra of Pareto Points. *Fundamenta Informaticae*, 78(1):35–74, September 2007.

[7] Ch. Ykman-Couvreur, V. Nollet, Fr. Catthoor, and H. Corporaal. Fast Multi-Dimension Multi-Choice Knapsack Heuristic for MP-SoC Run-Time Management. In *2006 International Symposium on System-on-Chip*, pages 1–4. IEEE, November 2006.

[8] R. Parra-Hernandez and N.J. Dimopoulos. A New Heuristic for Solving the Multichoice Multidimensional Knapsack Problem. *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on*, 35(5):708–717.

[9] BOSP: Barbeque Open Source Project. `https://bitbucket.org/bosp/barbeque/wiki/Home`.

[10] Reza Azimi, Michael Stumm, and Robert W. Wisniewski. Online performance analysis by statistical sampling of microprocessor performance counters. In *Proceedings of the 19th annual international conference on Supercomputing - ICS '05*, pages 101–110, New York, New York, USA, June 2005. ACM Press.

[11] A.R. Alameldeen and D.A. Wood. IPC Considered Harmful for Multiprocessor Workloads. *IEEE Micro*, 26(4):8–17, July 2006.

[12] D.H. Albonesi, R. Balasubramonian, 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, and S.E. Schuster. Dynamically tuning processor resources with adaptive processing. *Computer*, 36(12):49–58, December 2003.

[13] Ramazan Bitirgen, Engin Ipek, and Jose F. Martinez. Coordinated management of multiple interacting resources in chip multiprocessors:

A machine learning approach. In *2008 41st IEEE/ACM International Symposium on Microarchitecture*, pages 318–329. IEEE, November 2008.

[14] Seungryul Choi and Donald Yeung. Learning-Based SMT Processor Resource Distribution via Hill-Climbing. *ACM SIGARCH Computer Architecture News*, 34(2):239–251, May 2006.

[15] Mohammed El Shobaki. On-Chip Monitoring of Single- and Multiprocessor Hardware Real-Time Operating Systems. *In 8th International Conference On Real-Time Computing Systems and Applications. IEEE*, 2002.

[16] P. Kohout, B. Ganesh, and B. Jacob. Hardware support for real-time operating systems. In *First IEEE/ACM/IFIP International Conference on Hardware/ Software Codesign and Systems Synthesis (IEEE Cat. No.03TH8721)*, pages 45–51. ACM.

[17] Leandro Fiorin, Gianluca Palermo, and Cristina Silvano. A monitoring system for NoCs. In *Proceedings of the Third International Workshop on Network on Chip Architectures - NoCArc '10*, page 25, New York, New York, USA, December 2010. ACM Press.

[18] L. Benini and G. De Micheli. Networks on chips: a new SoC paradigm. *Computer*, 35(1):70–78, 2002.

[19] G. Almeida, Rémi Busseuil, Luciano Ost, Florent Bruguier, Gilles Sassatelli, Pascal Benoit, Lionel Torres, and Michel Robert. PI and PID Regulation Approaches for Performance-Constrained Adaptive Multiprocessor System-on-Chip. *Embedded Systems Letters, IEEE*, 3(99):77–80, September 2011.

[20] Brian D. Noble, M. Satyanarayanan, Dushyanth Narayanan, James Eric Tilton, Jason Flinn, and Kevin R. Walker. Agile application-aware adaptation for mobility. *ACM SIGOPS Operating Systems Review*, 31(5):276–287, December 1997.

[21] NetBSD operating system, `http://www.netbsd.org/`.

[22] L. Palopoli, T. Cucinotta, L. Marzario, and G. Lipari. AQuoSA-adaptive quality of service architecture. *Software: Practice and Experience*, 39(1):1–31, January 2009.

[23] Tommaso Cucinotta, Luca Abeni, Luigi Palopoli, and Giuseppe Lipari. A Robust Mechanism for Adaptive Scheduling of Multimedia Applications. *ACM Transactions on Embedded Computing Systems*, 10(4):1–24, November 2011.

[24] Woongki Baek and Trishul M. Chilimbi. Green: a framework for supporting energy-conscious programming using controlled approximation. *ACM SIGPLAN Notices*, 45(6):198–209, May 2010.

[25] Sasa Misailovic, Stelios Sidiroglou, Henry Hoffmann, and Martin Rinard. Quality of service profiling. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE '10*, volume 1, pages 25–34, New York, New York, USA, May 2010. ACM Press.

[26] T.F. Abdelzaher and J.A. Stankovic. ControlWare: a middleware architecture for feedback control of software performance. In *Proceedings 22nd International Conference on Distributed Computing Systems*, pages 301–310. IEEE Comput. Soc.

[27] Ashvin Goel, David Steere, Calton Pu, and Jonathan Walpole. SWiFT: a feedback control and dynamic reconfiguration toolkit. page 23, August 1998.

[28] Ch. Ykman-Couvreur, V. Nollet, Th. Marescaux, E. Brockmeyer, Fr. Catthoor, and H. Corporaal. Design-time application mapping and platform exploration for MP-SoC customised run-time management. *IET Computers & Digital Techniques*, 1(2):120–128, 2007.

[29] C. Ykman-Couvreur, P. Avasare, G. Mariani, G. Palermo, C. Silvano, and V. Zaccaria. Linking run-time resource management of embedded

multi-core platforms with automated design-time exploration. *IET Computers & Digital Techniques*, 5(2):123–135, 2011.

[30] Cristina Silvano, William Fornaciari, 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, and Tang Shibin. MULTICUBE: Multi-objective Design Space Exploration of Multi-core Architectures. In *2010 IEEE Computer Society Annual Symposium on VLSI*, pages 488–493. IEEE, July 2010.

[31] V. Zaccaria, G. Palermo, F. Castro, C. Silvano, and G. Mariani. Multicube Explorer: An Open Source Framework for Design Space Exploration of Chip Multi-Processors. *Architecture of Computing Systems (ARCS), 2010 23rd International Conference on*, pages 1–7, 2010.

[32] Giovanni Mariani, Gianluca Palermo, Cristina Silvano, and Vittorio Zaccaria. ARTE: An Application-specific Run-Time management framework for multi-core systems. In *2011 IEEE 9th Symposium on Application Specific Processors (SASP)*, pages 86–93. IEEE, June 2011.

[33] Gianluca Palermo, Cristina Silvano, and Vittorio Zaccaria. 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*, 28(12):1816–1829, December 2009.

[34] K.I. Smith, R.M. Everson, J.E. Fieldsend, C. Murphy, and R. Misra. Dominance-Based Multiobjective Simulated Annealing. *Evolutionary Computation, IEEE Transactions on*, 12(3):323–342.

[35] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, April 2002.

[36] Maggio, Martina and Hoffmann, Henry and Santambrogio, Marco D and Agarwal, Anant and Leva, Alberto. Decision making in autonomic computing systems: Comparison of approaches and techniques. In *Proceedings of the 8th ACM international conference on Autonomic computing - ICAC '11*, pages 201–204, New York, New York, USA, June 2011. ACM Press.

[37] Henry Hoffmann, Martina Maggio, Marco D. Santambrogio, Alberto Leva, and Anant Agarwal. SEEC: A Framework for Self-aware Computing. Technical report, MIT-CSAIL, 2010.

[38] Henry Hoffmann, Martina Maggio, Marco D. Santambrogio, Alberto Leva, and Anant Agarwal. SEEC: A General and Extensible Framework for Self-Aware Computing. Technical report, MIT-CSAIL, 2011.

[39] Henry Hoffmann, Martina Maggio, Marco D. Santambrogio, Alberto Leva, and Anant Agarwal. SEEC: A Framework for Self-aware Management of Multicore Resources. Technical report, MIT-CSAIL, 2011.

[40] Martina Maggio, Henry Hoffmann, Marco D Santambrogio, Anant Agarwal, and Alberto Leva. Controlling software applications via resource allocation within the heartbeats framework. In *49th IEEE Conference on Decision and Control CDC*, pages 3736–3741. IEEE, 2010.

[41] Henry Hoffmann, Jonathan Eastep, Marco D Santambrogio, Jason E Miller, and Anant Agarwal. Application heartbeats for software performance and health. *Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming PPoPP 10*, 45(5):347–348, 2010.

[42] Henry Hoffmann, Jonathan Eastep, Marco D. Santambrogio, Jason E. Miller, and Anant Agarwal. Application heartbeats. In *Proceeding of the 7th international conference on Autonomic computing - ICAC '10*, pages 79–88, New York, New York, USA, June 2010. ACM Press.

[43] Kishore Kumar Pusukuri, Rajiv Gupta, and Laxmi N. Bhuyan. Thread reinforcer: Dynamically determining number of threads via OS level monitoring. In *2011 IEEE International Symposium on Workload Characterization (IISWC)*, pages 116–125. IEEE, November 2011.

[44] Erez Perelman, Greg Hamerly, Michael Van Biesbrouck, Timothy Sherwood, and Brad Calder. Using SimPoint for accurate and efficient simulation. *ACM SIGMETRICS Performance Evaluation Review*, 31(1):318–319, June 2003.

[45] J B MacQueen. Some methods for classification and analysis of multivariate observations. In L M Le Cam and J Neyman, editors, *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability*, volume 1, pages 281–297. California, USA, University of California Press, 1967.

[46] Specifiche del processore Quad-Core AMD Opteron 8378 `http://products.amd.com/en-us/opteroncpudetail.aspx?id=495&f1=&f2=&f3=yes&f4=&f5=&f6=&f7=&f8=&f9=&f10=&f11=&`.

[47] Specifiche del processore Intel i7 2620M `http://ark.intel.com/products/52231/intel-core-i7-2620m-processor-(4m-cache-2_70-ghz)`.

[48] G. Lafruit. Cross-Based Local Stereo Matching Using Orthogonal Integral Images. *IEEE Transactions on Circuits and Systems for Video Technology*, 19(7):1073–1079, July 2009.

[49] Vincenzo Consales and Andrea Di Gesare. Implementazione di un Network Intrusion Detection System in CUDA: Accelerazione dell'algoritmo Aho-Corasick. Tesi di laurea. Politecnico di Milano, 2006.

[50] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC benchmark suite. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques - PACT '08*, pages 72–81. ACM Press, 2008.