Politecnico di Milano
V Facoltà di Ingegneria
Corso di laurea in Ingegneria Informatica
Dipartimento di Elettronica e Informazione

Studio empirico delle relazioni tra Garbage Collector, consumi e utilizzo della memoria

Relatore: Prof.sa Chiara FRANCALANCI
Correlatori: Prof. Eugenio CAPRA
Prof. Giovanni AGOSTA

Tesi di Laurea di:
Fabio BRAGA Matr. 735552

Anno Accademico 2009-2010

Ringrazimenti a ...

Abstract

This thesis work places itself in a larger project which addresses green IT energy efficiency issues; this project is called Green Software and is developed by Politecnico di Milano. The aim of Green Software Project is to set the guidelines to produce software with a very high energy efficency.
The softwares dimension growth leads companies to use high-level programming languages to simplify the development; many of this languages use a garbage collector to manage the heap.
In particular, this thesis analyzes the impact of garbage collector and its configurations on the energy efficency of various applications. The softwares tested include different versions, written in Java and C, of a memory allocation and deallocation benchmark and two Enterprise Resource Planning systems. All the applications are tested on two operating systems: Windows Server 32bit and Fedora 13 64bit. The goals of the thesis are to understand how much a configuration of the garbage collector influences the application's energy requirements and how this changes with different operating systems. In addition, the thesis provides guidelines on how to configure a Java Virtual Machine garbage collector and related knowledge to choose the best energy efficent configuration. To support this metodology was developed an application to find the best configurations for the JVMs, present on a server, to obtain the best energy efficency posssible.
From the tests results was observed that not always, increasing the amount of memory used by an application, it's possible to reduce its energy consumption; besides, setting with differents values the garbage collector parameters always

causes changes in applications energy consumption. Observing the tested applications behaviour and consumption, it's possible to conclude that doesn't exist the same optimal configuration for every software but everyone posses a different best one.

From values, noted on different systems, it's possible to observe that they use different amounts of energy mainly due the use of different architectures. In fact, 64bit architecture of Linux cause an important energy consumption overhead.

Riassunto

Questo lavoro di tesi si va ad inserire in un progetto più grande riguardante l'efficienza energetica; il progetto prende il nome di Green Software ed ha come scopo la definizione di linee guida per la produzione di software con un'alta efficienza energetica.
La crescita delle dimensioni dei software ha spinto le aziende ad utilizzare linguaggi di programmazione di alto livello per semplificarne lo sviluppo; gran parte di questi linguaggi utilizzano un garbage collector per gestire la memoria.
In particolare questa tesi analizza l'impatto del garbage collector e delle sue configurazioni sull'efficienza energetica di diverse applicazioni. I software testatati includono differenti versioni, scritte in Java e C, di un benchmark di allocazione e deallocazione della memoria e due Enterprise Resource Planning. Tutte le applicazioni sono state testate su due sistemi operativi diversi: Windows Server 32bit and Fedora 13 64bit.
Gli scopi della tesi sono capire quanto una configurazione del garbage collector influisce sulle richieste di energia di un'applicazione e quanto questo cambia tra diversi sistemi operativi.
In questo lavoro di tesi viene anche descritto come impostare correttamente le configurazioni del garbage collector in una Java Virtual Machine e viene fornito un know-how su come individuare la configurazione più efficiente dal punto di vista energetico. A supporto di questa metodologia è stata sviluppata un'applicazione per ricavare le configurazioni delle varie JVM, presenti su un server, in modo da ottenere la migliore efficienza energetica possibile.

Dai risultati dei test si è notato che non sempre, impostando il garbage collector per utilizzare un maggiore quantitativo di memoria, i consumi dell'applicativo si riducono; inoltre impostando differenti parametri di funzionamento si sono riscontrati ogni volta cambiamenti significativi nei consumi energetici. Osservando i comportamenti e i consumi delle applicazioni testate, si può concludere che non esiste una configurazione ottimale identica per tutti gli applicativi ma ognuno possiede la propria.

Dai valori rilevati, confrontando i due sistemi operativi testati, è stato possibile osservare che essi ottengono consumi differenti soprattutto a causa della diversa architettura utilizzata. Infatti l'architettura a 64 bit di Linux comporta un overhead in termini di consumo non trascurabile.

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Introduzione

1.1 Motivazione e obiettivi del lavoro

La crescita dell'impatto che l'informatica ha nella vita di ogni persona o impresa è cresciuto esponenzialmente negli ultimi anni. Ormai quasi la totalità delle aziende utilizza l'IT per operare in modo più efficiente ed efficace nel proprio ambiente di business.

I costi di un'architettura informatica si possono suddividere in tre parti fondamentali: i costi dell'hardware, i costi del software e i costi di gestione. All'interno di un'azienda gran parte della spesa informatica risiede nei costi di gestione e, in particolare, nei costi relativi all'energia elettrica utilizzata.

Mentre il costo di acquisto dell'hardware negli ultimi decenni è solo debolmente cresciuto, il costo per alimentare e raffreddare i sistemi è quadruplicato. Al giorno d'oggi il costo per l'energia e il raffreddamento rappresenta circa il 60% della spesa, con un impatto più che significativo sul Total Cost of Ownership. In aggiunta, la potenza elettrica richiesta da un data center cresce annualmente dell'8-10% e i gestori della rete elettrica rischiano di non essere più in grado di fornire una così elevata quantità di energia.

Tutto ciò pone le basi per la nascita di una disciplina, il *GreenIT*, che si occupa dei problemi legati all'impatto ambientale e al consumo energetico dei sistemi informativi.

In particolare il *GreenIT* risulta molto importante nel contesto odierno grazie principalmente a tre motivazioni:

- il costo energetico dell'IT è elevato
- il fabbisogno energetico di un'infrastruttura diviene un limite alla sua scalabilità
- l'IT ha un impatto significativo sull'inquinamento

La ricerca sul *GreenIT* si è da sempre focalizzata principalmente sull'efficienza energetica dell'hardware. Di conseguenza, in questi anni, tale obiettivo ha permesso di ottenere prestazioni molto più elevate a parità di consumi energetici. Si calcola che negli ultimi 30 anni, il valore di *MIPS/W* dei sistemi mainframe è aumentato di un fattore 28.000.
Tuttavia, benché l'hardware sia stato costantemente migliorato per essere energeticamente efficiente, la stessa strada non è stata intrapresa per il software applicativo. La disponibilità di hardware a basso costo e sempre più efficiente, ha portato i progettisti a trascurare l'efficienza energetica delle applicazioni lasciando ancora molti margini di miglioramento.

Questo lavoro si pone come obiettivo quello di esplorare l'influenza sui consumi di un particolare attore nella scena delle applicazioni moderne: il *Garbage Collector*. In moltissimi linguaggi di programmazione moderni, primo fra tutti Java, grazie ad esso il programmatore viene sollevato dal compito di gestire la memoria utilizzata.
Durante la fase di ricerca sono stati presi in considerazione degli applicativi Java sviluppati ad hoc o già esistenti, per testare le diverse configurazioni del *Garbage Collector* e verificarne gli impatti sui consumi energetici. Gli applicativi considerati sono un benchmark, chiamato *GCTest*, e due ERP. Inoltre è stata sviluppata una versione in linguaggio C dello stesso benchmark.

Per la misurazione del consumo energetico è stata utilizzata una scheda di acquisizione che permette di ottenere diversi dati relativi alle caratteristiche della richieste di energia.
Gli script e tutti i test effettuati possono essere impostati tramite un'interfaccia grafica, sviluppata appositamente, che gestisce in modo automatizzato tutto il sistema di acquisizione. Per i due ERP, tramite quest'ultima, è possibile specificare i parametri di creazione degli script che permettono di effettuare via browser le diverse operazioni desiderate.

Lo scopo ultimo della tesi è quello di produrre dei dati e delle linee guida che aiutino, al momento della creazione di un sistema informativo, ad identificare la soluzione e la configurazione che permette di ottenere la maggior efficienza energetica. Più precisamente si vuole studiare l'influenza sui consumi energetici di diversi fattori: il quantitativo massimo di memoria utilizzata, la configurazione del garbage collector, il sistema operativo e l'architettura software di quest'ultimo. Inoltre si intende effettuare un confronto fra i consumi energetici nel caso di gestione della memoria con garbage collector e nel caso senza garbage collector.

1.2 Descrizione dei capitoli

Nel Capitolo 2 viene mostrato lo stato dell'arte dei garbage collector esistenti focalizzandosi in particolare su quello implementato nella Java Virtual Machine. Successivamente vengono mostrati alcuni studi effettuati sul rapporto tra prestazioni e utilizzo del garbage collector.

Nel Capitolo 3 sono descritti gli studi già effettuati all'interno del progetto di *GreenSW* in cui si andrà ad inserire anche questa tesi.

Nel Capitolo 4 viene effettuato uno studio del tutto teorico sull'influenza del garbage collector sui costi di sviluppo, sul rapporto tra memoria e tempo d'esecuzione e infine sui parametri di configurazione del garbage collector Java. In seguito vengono definite le ipotesi di ricerca.

Nel Capitolo 5 viene illustrata la metodologia per la verifica delle ipotesi di ricerca, analizzando la struttura dell'ambiente di test sia a livello hardware che software e descrivendo i dettagli relativi ad ogni componente. In seguito sono illustrate le caratteristiche dell'applicazione sviluppata per l'analisi del comportamento del garbage collector.

Nel Capitolo 6 vengono illustrati i test effettuati tramite il benchmark *GCTest* e commentati i relativi risultati. In seguito sono approfondite le caratteristiche dei test sugli ERP Adempiere e Openbravo e vengono tratte alcune conclusioni basandosi sui risultati ottenuti. Infine sono riassunte le ipotesi di ricerca specificando per ognuna le conclusioni tratte.

Nel Capitolo 7 è descritto il piccolo tool creato per la configurazione di un server al fine di ottenere la miglior efficienza energetica.

Nel Capitolo 8 vengono tratte le conclusioni finali sul lavoro svolto.

Nelle Appendici A, B e C sono descritti rispettivamente i dettagli relativi alla configurazione di tutto il software analizzato, ai collegamenti della scheda di acquisizione e infine quelli riguardanti le applicazioni sviluppate.

Capitolo 2

Stato dell'Arte

In questo capitolo si è cercato di mostrare la situazione dello stato dell'arte delle tecniche di *garbage collection* utilizzate e lo stato della ricerca odierna e passata. Dopo una descrizione dei compiti e delle tecniche utilizzate dalla maggior parte dei *garbage collectors* verranno mostrati degli esempi pratici di quest'ultimi: il *Boehm-Demers-Weiser Garbage Collector* e quelli implementati nelle ultime versioni della *Java Virtual*
Machine. Infine verranno riassunti alcuni studi effettuati sull'influenza del garbage collector sulle prestazioni del sistema in termini di tempo d'esecuzione e occupazione di memoria.

2.1 Garbage Collector

Il *garbage collector* è una funzionalità introdotta in diversi linguaggi di sviluppo allo scopo di sollevare il programmatore dal compito di allocare e deallocare la memoria delle variabili in modo esplicito.
D'ora in poi verrà usata l'abbreviazione GC per indicare il garbage collector.
Nei linguaggi che non possiedono questa funzionalità, come il C o il C++, è lo sviluppatore che ha il compito di allocare memoria quando crea delle variabili e deallocarla quando non sono più necessarie. Con questo processo manuale è facile incorrere in errori di programmazione a per cui delle porzioni di memoria, che

di fatto possono essere riutilizzate, rimangono invece allocate. Questi errori sono chiamati *memory leaks*.
Il GC, quindi, ha il compito di individuare le locazioni di memoria appartenenti a variabili create, ma che non sono più necessarie, per deallocarle. Per effettuare questa operazione sono state sviluppate negli anni diverse tecniche utili all'individuazione delle variabili dette *morte* [21]. Una variabile viene considerata *morta* quando non è più referenziata da qualche altra variabile.
Le tecniche di più famose sono:

- mark and sweep: ogni oggetto in memoria ha un flag che rappresenta il suo stato (in uso o no). L'algoritmo partendo dalle variabili del *root set* comincia a marcare come in uso tutte le variabili raggiungibili tramite riferimenti da esso. Alla fine del primo step ripercorre tutta la memoria per eliminare quelle che risultano non più necessarie. Uno dei problemi di questa tecnica è rappresentato dal fatto che per portare a termine il processo è necessario percorrere l'intera memoria minimo due volte. Nella Figura 2.1 è presente un esempio di questa tecnica.

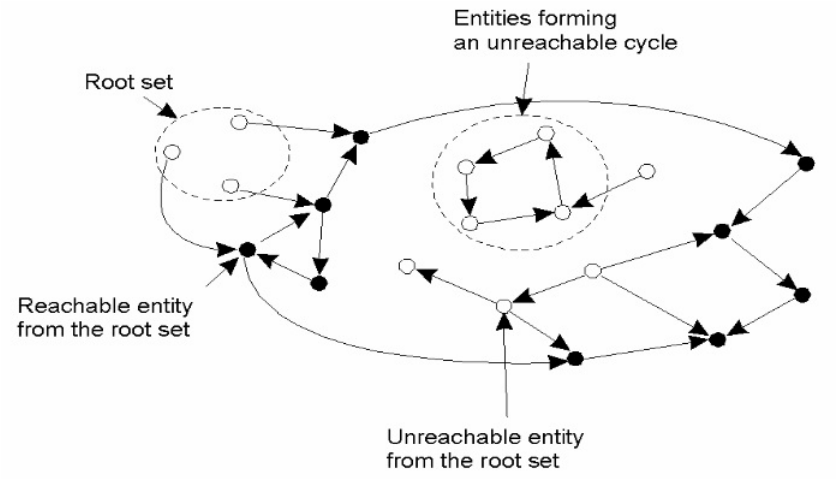


Figura 2.1: *Esempio di un'esecuzione della tecnica MarkSweep*

- tri-color marking: è una variante del mark and sweep nella quale viene aggiunto uno stato oltre ai due già esistenti che rappresenta gli oggetti che

non possiedono riferimenti ad altri oggetti. Viene chiamata tri-colour perché si considerano come neri gli oggetti in uso, grigi quelli appartenenti al nuovo stato e bianchi quelli da eliminare. In questo modo è possibile, una volta settati tutti i flag, evitare di analizzare l'intera memoria partendo prima dagli oggetti di colore grigio e poi se necessario procedere con quelli neri che fanno riferimento ad essi.

- reference counting: durante la creazione di oggetti e la loro distruzione si tiene traccia del numero di riferimenti a cui sono associati. Quando il numero raggiunge zero possono venire eliminati. Il vantaggio di questa tecnica è che riduce il tempo di analisi della memoria però, come contro, è necessaria un maggior quantitativo di memoria e istruzioni macchina per tenere aggiornato il contatore durante l'esecuzione.

Le implementazioni attuali dei GC utilizzano versioni modificate o ibride per migliorare le prestazioni della fase di individuazione delle locazioni di memoria da liberare.
Lo step successivo consiste nel rendere disponibili le celle di memoria non più utili e, nel caso sia previsto, riorganizzare la memoria per contenere le variabili *vive*. A questo scopo vengono utilizzate alcune metodologie differenti:

- semi-spaces: vengono usati due spazi di memoria nella heap di uguali dimensioni. Le variabili vengono create in uno spazio; quando questo è pieno, dopo aver definito gli oggetti vivi, li copia nell'altra area di memoria. Una volta finito il processo vengono scambiati i ruoli dei due spazi della heap. Il tempo di esecuzione è proporzionale al numero degli oggetti vivi e quindi diventa molto pesante nel caso di molte variabili con vita lunga.

- free-list: viene mantenuta una lista contenente le locazioni di memoria che sono state liberate perché precedentemente occupate da oggetti morti. Con

questa tecnica non avviene un compattamento della memoria in cui vengono salvati gli oggetti sopravvissuti con conseguente rischio di frammentazione.

- generational: vengono create due aree di memoria di dimensioni differenti, una per l'immagazzinamento delle variabili dette *young* e l'altra per le variabili *old*. Quando un oggetto viene creato è allocato nell'area di memoria young e se sopravvive a una o più garbage collection viene spostato nell'area tenuered. Dopo di che verrà analizzata più frequentemente l'area young rispetto quella tenuered con conseguente risparmio di risorse; solo dopo che lo spazio per le variabili old sarà pieno verrà effettuata una garbage collection sull'intera heap. Questa tecnica si basa sull'ipotesi che normalmente nei programmi gli oggetti si dividono in due categorie: giovani che muoiono presto e vecchi che sopravvivono a lungo (alta mortalità infantile).

Quasi la totalità dei GC che sono stati implementati fino ad ora utilizzano una o più delle tecniche presentate, eventualmente modificate, per cercare di sfruttare i vantaggi di una riducendone gli svantaggi. Successivamente verranno mostrati in dettaglio due GC in cui le politiche scelte sono degli ibridi di quelle mostrate.

2.2 Boehm-Demers-Weiser GC

In questa implementazione [12], sviluppata inizialmente per C e C++, viene utilizzato l'algoritmo *mark and don't sweep* per l'individuazione e la cancellazione degli oggetti morti. Il root set è composto dalle variabili presenti nello stack e da quelle statiche. Il BDW viene definito conservativo perché nel C e C++ il collector non può sapere a runtime le posizioni in memoria che rappresentano puntatori e perciò, ogni locazione che contiene un bit pattern che potrebbe essere interpretato come puntatore alla heap, viene considerata come tale. La compattazione della memoria non è prevista in questo algoritmo. Quando serve una posizione di memoria per l'allocazione di una nuova variabile viene utilizzata la tecnica *lazy*

sweep nella quale viene analizzata la memoria fino a trovare uno spazio marcato come libero e utile per le dimensioni del nuovo oggetto; per questo motivo l'algoritmo è chiamato *mark and don't sweep*. Perciò, la durata della garbage collection dovrebbe dipendere dal numero di oggetti vivi e non dalla dimensione della heap. Il BDW sfrutta anche il meccanismo degli algoritmi generational dato che concentra la ricerca di spazi liberi o riallocabili nei blocchi di memoria più recenti.
Il BDW è open source e viene fornito come libreria statica o dinamica. Può essere richiamato ed eseguito sostituendo alle funzioni classiche del C e C++ per l'allocazione e la deallocazione di memoria quelle presenti nella libreria (es. *malloc() diventa GC_malloc()*). I metodi per fare ciò nel C sono principalmente 3 [13]:

- sostituire i nomi delle funzioni richiamate tramite la direttiva di *define*
- creare un wrapper che intercetti le chiamate alle funzioni standard
- abilitare in fase di configurazione il redirect delle funzioni standard (*configure –redirect-malloc*). Questa opzione però al momento supporta solo eseguibili single-thread.

2.3 Java Virtual Machine

2.3.1 Default GC

La tecnica utilizzata dalla JVM 1.2 è di tipo generational e sfrutta entrambi i meccanismi di *copia* e *mark-compact* [11]. Il meccanismo di copia delle variabili sopravvissute non è performante se il loro numero è elevato ma lo è se il numero di oggetti morti è elevato. Il meccanismo di mark-compact, che è una variante del mark and sweep, effettua una compattazione della memoria contenente le variabili vive shiftando in base alle locazioni precedentemente occupate da quelle morte; in questo modo è possibile risolvere il problema della frammentazione della memoria. Quest'ultima tecnica risulta ottimale nel caso di variabili *long-lived* (non devono essere copiate continuamente) ma non funziona altrettanto bene con

quelle *short-lived*. L'obiettivo dell'approccio *generational* è quindi quello di impiegare le due tecniche nell'area di memoria (young o tenuered) dove sono più efficienti.Attualmente, oltre a queste due aree di memoria, è presente una terza detta *permanent space* dove vengono caricati le *loaded class* e i metodi degli oggetti.

Studi empirici hanno dimostrato che la maggior parte degli oggetti giovani muore presto mentre quelli che superano almeno una garbage collection tendono ad avere una vita abbastanza lunga. Dato che il meccanismo di copia funziona bene con un numero di oggetti short-lived elevato, viene usato per effettuare la garbage collection nell'area di heap riservata alle variabili young. Al contrario, la tecnica mark-compact, che funziona bene con oggetti long-lived, viene applicata alla parte di memoria relativa alle variabili old. In questo modo è possibile sfruttare i vantaggi di entrambe le tecniche cercando di limitarne gli svantaggi. E' da notare che la scelta delle dimensioni dell'heap e delle sottoparti relative alle due tipologie di oggetti influenzerà sicuramente le performance del GC.

Il vantaggio di suddividere in due aree di memoria le variabili permette di analizzarle individualmente e con cadenze differenti. Il meccanismo di collection per l'analisi dell'area young infatti parte dal root set per identificare i riferimenti con le variabili giovani, ma non analizza i riferimenti contenuti negli oggetti old. A causa di ciò si presenta però un problema: cosa succede se un oggetto giovane è referenziato solo da un oggetto vecchio?

Per trattare correttamente questo caso è necessario tenere traccia dei riferimenti intergenerazionali detti *old-to-young* e inserirli nel root set della generazione giovane.

Il GC tiene traccia della creazione dei riferimenti intergenerazionali che si vengono a creare dopo la promozione di un oggetto allo stato di old. Il *mutator* (il programma vero e proprio) quando effettua operazioni su puntatori può creare nuovi riferimenti old-to-young e quindi il collector deve riuscire a notarli per ag-

giornare la lista. Per fare questo la JVM utilizza un algoritmo di *card marking* che consiste nel suddividere la heap in *cards* e mantenere una lista delle cards modificate tramite il settaggio di un bit. In fase di collection vengono analizzati solo gli oggetti all'interno delle cards modificate.

Nella Figura 2.2 si possono osservare degli esempi di collegamenti fra oggetti

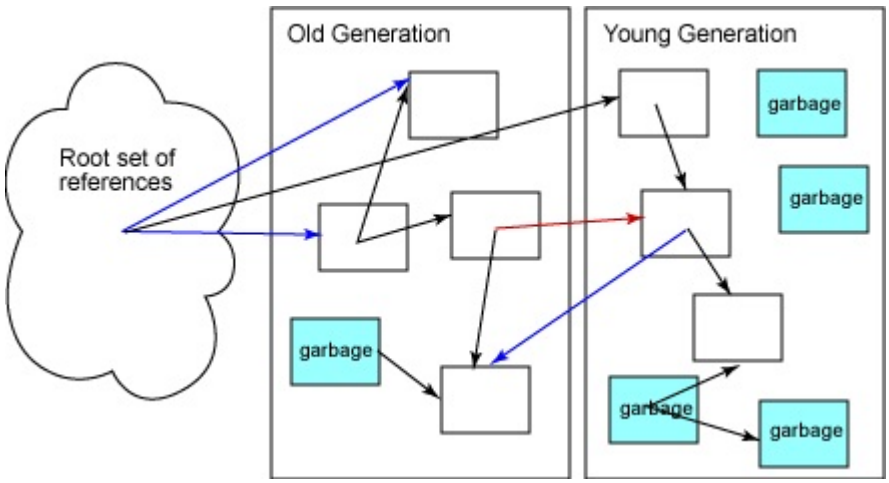


Figura 2.2: *Riferimenti tra oggetti young e old*

old e young; in rosso sono rappresentati i collegamenti old-to-young che devono essere aggiunti al root set della young generation, in blu quelli verso oggetti old che non devono essere presi in considerazione al momento dell'analisi sulla parte di memoria contenente gli oggetti giovani.
Per lanciare il garbage collection per l'intera heap è presente il comando *System.gc()* ma non ne esiste uno per lanciarlo esclusivamente per una sola parte di memoria.

2.3.2 Altri GCs

In aggiunta al GC di default che utilizza l'algoritmo di copia e il mark-compact, dalla versione 1.4.1 della JDK, sono presenti altri 4 tipi di GC ognuno realizzato per differenti obiettivi. Gli altri algoritmi sono l'*incremental collector*, il *parallel copying collector*, il *parallel scavenging collector* e il *concurrent mark-copy collector*. Questi ultimi tre sono stati introdotti per sfruttare al meglio le funzionalità

offerte dai sistemi multiprocessore.

Nella Figura 2.3 sono mostrate alcune linee guida da seguire nella scelta del

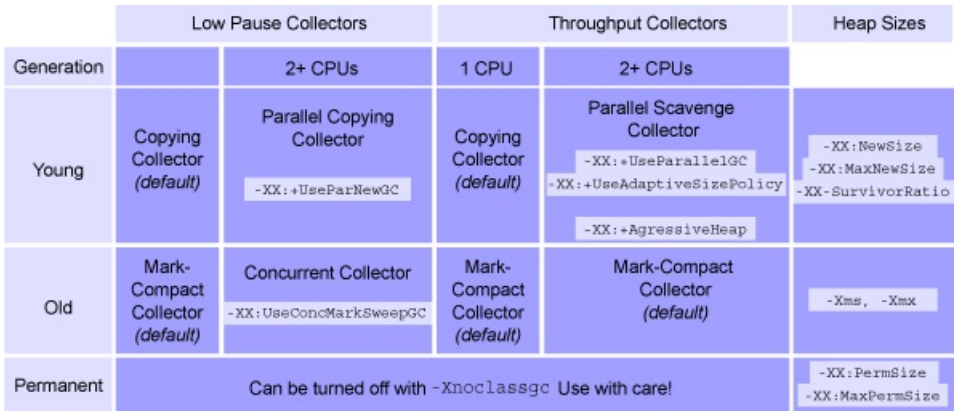


Figura 2.3: *Tipi di GC nella JVM di Sun*

garbage collection più adatto in base alle necessità [18].

L'*incremental collection* riduce le pause dovute alla garbage collection a scapito del throghput rendendolo utile nel caso l'obiettivo sia ridurre al minimo la durata di quest'ultime come in sistemi simil real-time. L'algoritmo usato da questo collector e detto *train* e crea una nuova area di memoria tra la generazione young e quella tenuered. Quest'area è divisa in sezioni di heap dette *trains* che a loro volta sono divise in *cars*. Ogni car può essere analizzata separatamente e rappresenta una generazione a sé stante con tutte le conseguenze del caso. In questo modo è necessario più lavoro per effettuare la collection ma comporta pause più brevi.

Dato che il GC standard blocca il processo del mutator quando viene eseguito, può risultare essere un collo di bottiglia soprattutto in sistemi multiprocessore. In questi sistemi difatti è possibile sfruttare il processore in idle per eseguire la collection mentre l'altro continua ad eseguire il programma.

Il *parallel copying collector* si occupa della regione di memoria young e divide i compiti di copia in tanti threads quanti sono i processori disponibili.

Il *concurrent mark-sweep* si occupa della regione di heap tenuered e blocca inizialmente il programma per eseguire una fase iniziale di mark e poi lo sblocca nella fase di sweep eseguita da diversi thread in parallelo.

Il *parallel scavengin collector* si occupa anch'esso della generazione giovane ed è ottimizzato per heap molto grandi (un gigabyte o più) su sistemi multiprocessore.

2.4 GC e Prestazioni

2.4.1 Impatto del GC sulle performance

Sono stati svolti diversi lavori di ricerca con lo scopo di identificare come diverse tecniche di garbage collection possono influire sulle prestazioni globali di un sistema in termini di tempo d'esecuzione, allocazione di memoria ecc.
Nello studio *Myths and Realities: the performance impact of GC* [22] sono stati analizzati diversi metodi di garbage collection col fine di identificare i loro effetti, oltre che sul throghput di istruzioni eseguite, anche sui vantaggi derivanti dalla creazione di località fra gli oggetti in memoria.
I collectors analizzati sono stati:

- SemiSpace: usa due spazi di memoria di dimensioni identiche, continua allocare in uno e quando questo è pieno traccia e copia gli oggetti vivi nell'altro. I nuovi oggetti saranno allocati sempre nella parte dove sono state copiate le variabili vive. Le performance di questa tecnica sono inversamente proporzionali al numero di variabili sopravvissute e quindi soffre in caso di oggetti long-lived e heap grande.

- MarkSweep: in questo caso le variabili sopravvissute non vengono copiate ma viene tenuta una *free-list* contenente le locazioni di memoria dove è possibile scrivere dopo l'esecuzione del GC. Anche questa tecnica soffre in caso di variabili long-lived dato che deve comunque analizzarle ed effettuare il tracing sull'intera heap.

- GenCopy: è un generational collector che alloca le variabili in un spazio di memoria detto *nursery* e poi quando sopravvivono a una collection le sposta

nell'area di memoria matura. Quando l'area matura è piena esegue una collection sull'intera heap. I vantaggi di questa tecnica consistono nel non dover controllare ogni volta tutta la memoria. Questo vantaggio è ancora più consistente nel caso di programmi in cui molti oggetti giovani muoiono velocemente senza essere spostati nell'area matura.

- GenMS: il concetto è uguale a quello del GenCopy ma in questa tecnica la parte matura di memoria viene analizzata con l'algoritmo MarkSweep. Il rischio è creare frammentazione della memoria ma è possibile adottare il MarkCompact invece che il MarkSweep.

- GenRC: utilizza un meccanismo di copia per la nursery e uno di reference counting per la parte matura. Entrambe le parti di memoria sono analizzate ogni volta permettendo di avere una minore footprint. Il tempo di esecuzione di questo collector è proporzionale alle dimensioni della nursery e al numero di oggetti morti.

Nella Figura 2.4 vengono mostrati i tempi di esecuzione totale, del GC, del mutator e le statistiche della cache dei vari benchmark utilizzando le tecniche per il GC descritte prima. Sull'asse y sono indicati i tempi di esecuzione mentre sull'asse x la grandezza di heap per quella determinata esecuzione.
Dalla figura si può notare che spesso aumentando la grandezza della heap si diminuisce il numero di entrate in funzione del GC e quindi anche il tempo di esecuzione. Alcune volte però, anche con piccole variazioni sulla dimensione della memoria, è possibile assistere a grandi cambiamenti nei tempi d'esecuzione. Questo perché le performance del GC dipendono molto anche dalla situazione in cui si trova ad operare e quindi dal punto nel programma in cui viene richiamato. Sempre osservando la figura si nota che le tecniche generazionali superano in quasi tutti i casi le alternative che agiscono sull'intera heap. I casi in cui questa considerazione risulta meno forte sono quelli dove c'è una bassa mortalità degli oggetti.

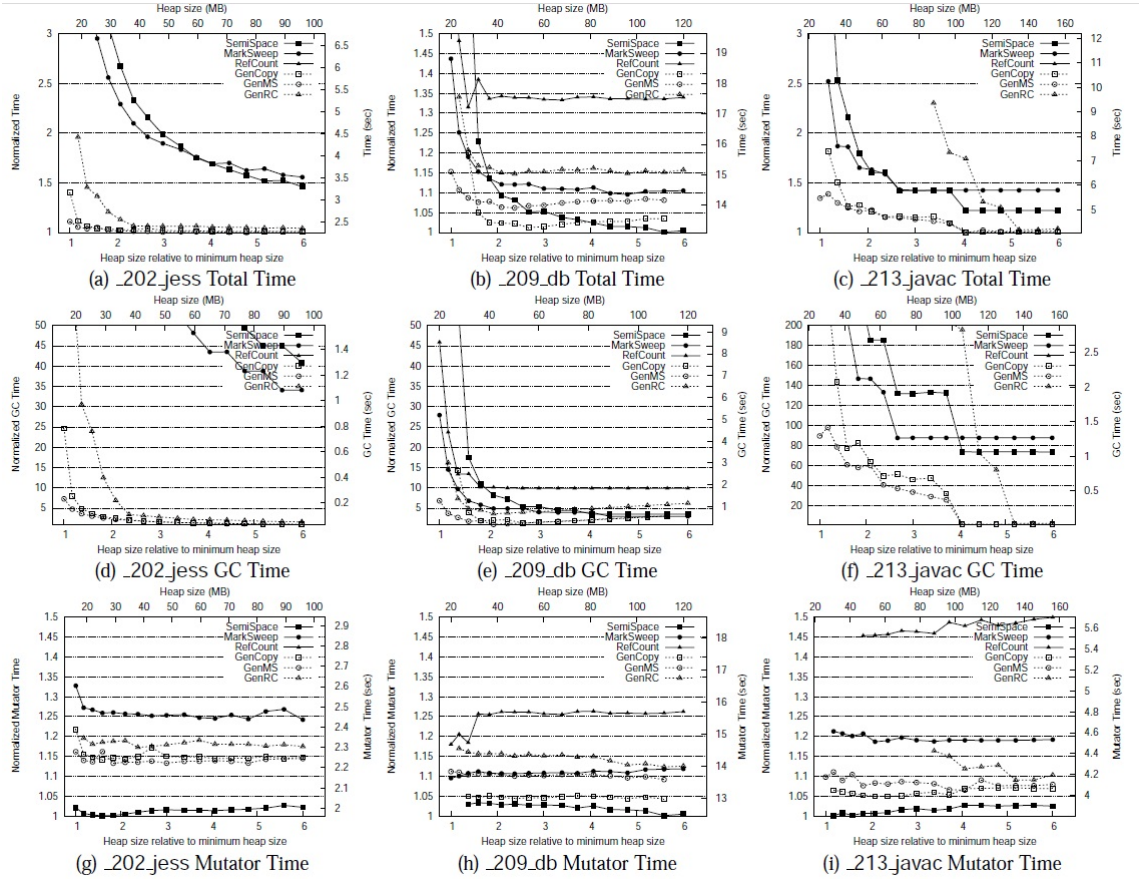


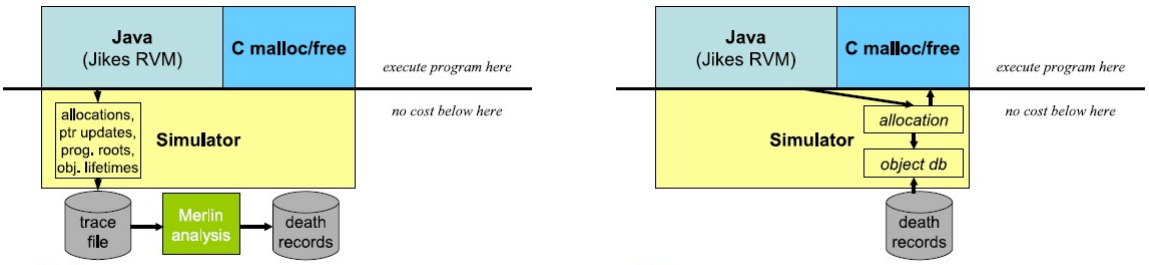
Figura 2.4: *Tempi d'esecuzione totali, del mutator e del GC per i sei collectors esaminati [22]*

In seguito sempre nello stesso studio è stata mostrata l'importanza, in determinati contesti, della località delle variabili raggiunta tramite l'allocazione contigua di certi algoritmi. Questo ragionamento ha portato gli autori a reputare come un vantaggio la presenza del GC nei casi analizzati.
La conclusione del lavoro può essere schematizzata in due punti. Primo, anche se il programma non segue l'ipotesi generazionale (alta mortalità infantile), l'allocazione contigua derivante dai meccanismi di copia per la nursery offre diversi benefici, indicando come da preferire le tecniche generazionali. Secondo, la scelta della tecnica di GC per l'area di memoria degli oggetti long-lived non deve essere dettata esclusivamente dall'efficienza spaziale che ne deriva, che porterebbe a preferire sempre il MarkSweep, ma deve tenere conto anche del rate di morte fra gli oggetti old e il rate di accesso e modifica dell'area di memoria matura. Se questi rates sono alti, un meccanismo di copia per l'area matura permetterebbe di ottenere vantaggi per il mutator tramite la località superando alla lunga gli svantaggi derivanti dai più alti tempi di collection.

2.4.2 GC vs Gestione Esplicita della Memoria

Verificare le differenze di performance tra allocazione e deallocazione esplicita e l'uso di un GC risulta abbastanza semplice per linguaggi, come il C o il C++, sprovvisti di GC. In questo caso difatti è sufficiente linkare la libreria del GC (es. Boehm-Demers-Weiser collector) e osservare le variazioni fra i due casi.
In linguaggi in cui è già presente un meccanismo di GC diventa molto più difficile ottenere risultati altrettanto veritieri perché non è possibile sostituire direttamente il meccanismo di collection con una gestione esplicita della memoria. In alternativa si potrebbe misurare il costo delle attività di garbage collection e sottrarlo a quello totale ottenendo così il costo del solo mutator. In questo modo si trascurerebbero però i vantaggi derivanti dalla riorganizzazione della memoria da parte del GC, come la località dei dati, o la possibilità di riutilizzare le parti di memoria liberate appena non sono più utili.

Nello studio *Quantifying the Performance of GC vs Explicit Memory Management* [15] gli autori hanno creato un programma che, analizzando l'esecuzione di un applicativo Java, permette di identificare dove devono essere liberate determinate variabili sollevando il GC dall'effettuare il tracing degli oggetti morti. Nella Figura 2.5 viene mostrata l'architettura generale di questo programma detto oracolo che permette di inserire in modo off-line delle chiamate alle funzioni di liberazione della memoria. Per off-line si intende prima dell'esecuzione del programma.



(a) Step one: collection and derivation of death records.

(b) Step two: execution with explicit memory management.

Figura 2.5: *Architettura dell'Oracolo*

Nei test sono state usate due tipologie di oracolo:

- lifetime-based: chiama la *free* non appena la variabile può essere deallocata con sicurezza ovvero quando non è più utilizzabile. Questo approccio è più aggressivo del secondo metodo.
- reachability-based: è più conservativo, chiama la *free* nell'ultimo istante disponibile per farlo ovvero quando sta per diventare irraggiungibile.

E' stato osservato che la versione dell'oracolo reachability-based in modalità online incrementa il tempo di esecuzione dal 2% fino al 33%. Per analizzare meglio gli effetti della deallocazione esplicita si è scelto di effettuare i test utilizzando la modalità off-line in modo da escludere il costo delle chiamate all'oracolo utili a sapere quando deallocare.

Nei test sono stati confrontati diversi GC e diversi meccanismi di allocazione per

Garbage collectors	
MarkSweep	non-relocating, non-copying single-generation
GenCopy	two generations with copying mature space
SemiSpace	two-space single-generation
GenMS	two generations with non-copying mature space
CopyMS	nursery with whole-heap collection
Allocators	
Lea	combined quicklists and approximate best-fit
MSExplicit	MMTk's MarkSweep with explicit freeing

Figura 2.6: *Memory Manager e GC utilizzati*

la versione esplicita, in Figura 2.6 sono mostrate le loro principali caratteristiche. Comparando il tempo d'esecuzione, il consumo di spazio in memoria e la grandezza della memoria virtuale occupata dai vari applicativi utilizzati, gli autori hanno notato che le performance del miglior GC sono competitive rispetto a quelle dell'allocazione esplicita quando viene utilizzato un quantitativo di memoria sufficiente. Difatti le performance del GC degradano di molto nel caso si sia costretti ad utilizzare una heap dalle dimensioni ridotte. Più precisamente quando viene utilizzato un quantitativo di memoria cinque volte superiore al necessario, le performance del GC sono comparabili e a volte superano quelle della gestione esplicita. Con tre volte la memoria necessaria, invece, si riscontra in media un peggioramento del 17% e con due volte addirittura del 70%. I due GC risultati più efficienti sono il GenMS e il GenCopy con un leggero vantaggio del primo. Analizzando l'andamento delle prestazioni rispetto l'aumento della heap è stato possibile definire un rapporto tra la grandezza della memoria e il tempo d'esecuzione. Di seguito è mostrata la formula che lega con buona approssimazione le performance dei due migliori GC alla dimensione della heap e i relativi coefficienti.

$$ExecTimeFactor = \frac{a}{b - HeapSizeFactor} + c$$

$$GenMS : a = -0.246, b = 0.59, c = 0.942$$

$$GenCopy : a = -0.297, b = 0.784, c = 0.1.031$$

Dai risultati dello studio si può concludere che se non si ha sufficiente memoria (almeno 3 volte la necessaria) l'uso del GC è sconsigliato perché porterebbe a un

peggioramento delle performance soprattutto in ambiti dove si fa uso intenso della RAM, come nei *DB in-memory* e nei *motori di ricerca*, e di conseguenza il GC risulterebbe sottoposto a un carico maggiore.

2.4.3 Dimensioni della Heap e Performance

Nei sistemi forniti di GC c'è sempre il rischio di settarlo in modo che entri in funzione troppo o non abbastanza frequentemente. Se viene richiamato troppo frequentemente c'è il rischio di introdurre un overhead non necessario dato che non avrà abbastanza memoria da liberare per giustificarne l'esecuzione. Invece, se richiamato a intervalli di tempo troppo lunghi, è probabile che il programma userà molta memoria virtuale e ciò causerà un incremento del tempo d'esecuzione a causa del paging.
A tal proposito è stato effettuato uno studio [23] per esaminare come la frequenza di chiamata impatta sul tempo d'esecuzione, la footprint e le pause derivanti dal processo di garbage collection. Nello studio sono stati analizzati diversi applicativi Java abbinati al GC Boehm-Demers-Weiser (BDW).
Nel BSD, quando non è disponibile uno spazio di memoria grande a sufficienza

```
if (alloced_since_last_GC >= (heap / FSD)) {
  collect_garbage()
} else {
  grow_heap_by((heap / FSD) + request_size)
}
```

Figura 2.7: *Pseudo-codice dell'algoritmo di scheduling del BDW*

per contenere il nuovo oggetto, viene eseguito un algoritmo di cui una versione semplificata è rappresentata in Figura 2.7. Questo algoritmo serve per cercare di richiamare il collector solo se esso potrà liberare sufficiente memoria da giustificarne la chiamata. Nel caso ciò non sia possibile verrà aumentata la dimensione della heap. Le decisioni derivanti dall'algoritmo saranno fortemente influenzate

dal *free space advisor* (FSD), per esempio se FSD=2 verrà lanciata la collection solo se dall'ultima chiamata è stata allocata almeno la metà della heap altrimenti verranno aumentate le dimensioni della heap di circa la metà della dimensione attuale.

Nella Figura 2.8 vengono mostrati i risultati ottenuti con diversi valori di FSD

Alg	Run Time			GCs	GC Time	Avg Pause	Max Pause	Avg Foot	Faults	GC Faults
fred										
Off	1654	+/-	4.4	1	2	2	2	12199	7547	24
FSD 1	1671	+/-	9.1	1	1	1	1	12199	7547	24
FSD 2	2174	+/-	3.9	7	502	71	237	3456	5756	33
FSD 4	3035	+/-	6.1	16	1334	83	302	1978	5248	35
FSD 8	4227	+/-	6.4	26	2509	96	323	1456	5155	35
FSD 16	6240	+/-	4.7	41	4472	108	320	1138	5051	35
Tholds	2200	+/-	67.1	2	478	176	314	8190	6150	32
db.100										
Off	68823	+/-	1145.8	1	19	19	19	56245	35709	24
FSD 1	71294	+/-	2111.1	1	20	20	20	56245	35653	24
FSD 2	55480	+/-	9.2	14	2268	161	224	6862	7682	42
FSD 4	57044	+/-	6.1	24	3927	163	220	4330	6355	45
FSD 8	62727	+/-	7.8	52	9509	182	223	2709	5462	45
FSD 16	69201	+/-	22.5	89	15988	179	221	2169	5170	45
Tholds	55443	+/-	5.2	11	2253	204	231	6853	7265	41
javac.100										
Off	463472	+/-	40770.6	1	19	19	19	137315	183910	24
FSD 1	452722	+/-	26595.3	2	16034	8017	16012	88581	162761	2989
FSD 2	323418	+/-	14149.3	17	92082	5329	53188	17074	77897	16470
FSD 4	74770	+/-	7267.6	36	24165	663	10012	8818	21387	2537
FSD 8	61744	+/-	265.0	65	27549	417	1126	5935	13677	277
FSD 16	87443	+/-	1606.7	121	55480	454	943	4320	11525	63
Tholds	64130	+/-	4696.4	25	23147	911	3501	8776	17535	1894

Figura 2.8: *Impatto della frequenza di chiamata del GC (BDW) con 64MB di RAM sull'esecuzione, il tempo è in millisecondi e le dimensioni in KB. Tholds sta a indicare il nuovo algoritmo introdotto dagli autori dello studio*

durante l'esecuzione di tre applicazioni java di benchmark. Dalla figura si può notare che la frequenza con cui viene chiamato il GC influenza sostanzialmente il tempo d'esecuzione e che non esiste un valore di FSD corretto per tutte le applicazioni. Inoltre è osservabile che aumentando la frequenza di chiamata al GC non è detto che il tempo medio di durata della collection sia inferiore. Chiamate più frequenti, però, portano ad avere footprint di dimensioni minori permettendo di minimizzare il numero di page faults durante l'esecuzione del programma. Nella Figura 2.9 gli stessi test sono eseguiti in un sistema con un quantitativo di

Alg	Run Time			GCs	GC Time	Avg Pause	Max Pause	Avg Foot	Faults	GC Faults
fred										
Off	1646	+/-	11.1	1	4	4	4	12199	7546	24
FSD 1	1681	+/-	17.6	1	5	5	5	12199	7546	24
FSD 2	2205	+/-	22.3	7	497	70	236	3456	5755	33
FSD 4	3034	+/-	11.7	16	1331	82	302	1978	5249	35
FSD 8	4228	+/-	8.2	26	2502	96	322	1457	5154	35
FSD 16	6233	+/-	8.7	41	4468	108	320	1137	5050	35
Tholds	1627	+/-	6.0	1	1	1	1	12199	7546	23
db.100										
Off	56198	+/-	197.1	1	36	36	36	56245	31755	24
FSD 1	56138	+/-	198.2	1	26	26	26	56245	31788	24
FSD 2	55349	+/-	4.6	14	2260	161	224	6862	7681	42
FSD 4	56921	+/-	5.0	24	3917	163	219	4330	6354	45
FSD 8	62597	+/-	7.4	52	9503	182	226	2709	5461	45
FSD 16	69053	+/-	21.7	89	15960	179	219	2169	5169	45
Tholds	53114	+/-	7.6	2	272	136	269	38101	25103	41
javac.100										
Off	405112	+/-	7328.3	1	24	24	24	137327	190956	24
FSD 1	268804	+/-	4031.4	2	438	219	401	88593	133641	42
FSD 2	38039	+/-	146.8	17	7008	394	1013	15683	19248	65
FSD 4	45687	+/-	272.5	37	14598	390	1000	8836	16346	65
FSD 8	58679	+/-	229.7	65	27342	417	976	5961	13376	74
FSD 16	87488	+/-	1499.5	122	55669	454	944	4322	11570	68
Tholds	35419	+/-	187.1	4	4570	921	1380	35173	25060	62

Figura 2.9: *Impatto della frequenza di chiamata del GC (BDW) con 128MB di RAM sull'esecuzione*

RAM maggiore e da questa si può osservare che i valori ottimali di FSD relativi ai benchmark variano rispetto al caso precedente. Per questo motivo gli autori hanno deciso di sviluppare un nuovo algoritmo che tiene in considerazione anche la memoria disponibile in modo da eseguire un'applicazione il più vicino possibile alla frequenza di chiamata del GC migliore.
Nella costruzione del nuovo algoritmo sono state seguite le seguenti linee guida:

- se c'è sufficiente memoria disponibile il GC non dovrebbe essere eseguito ma dovrebbe essere fatta crescere la heap
- quando la memoria disponibile si riduce si dovrebbe far iniziare una fase con GC più aggressivo e crescita della heap meno aggressiva in modo da minimizzare lo spreco di tempo dovuto ai page fault
- quando la memoria disponibile diventa veramente bassa è necessario creare un meccanismo che non permetta al GC di essere richiamato troppo fre-

quentemente, magari considerando prima se esso nelle chiamate precedenti ha effettivamente liberato un quantitativo di memoria sufficiente per soddisfare la necessità attuale

Di seguito è descritto il funzionamento dell'algoritmo:

Quando il *memory allocator* non riesce a trovare un blocco di memoria sufficiente per la nuova richiesta deve iniziare una garbage collection o aumentare la heap. Questa decisione è presa in base alla memoria disponibile, al quantitativo di memoria recuperata recentemente tramite il GC e infine se dall'ultima garbage collection è stata superata o meno una soglia di memoria allocata.
Finché la memoria allocata non super la prima soglia il GC non entra in funzione; quando una soglia è superata per la prima volta il GC viene sempre attivato. Durante l'esecuzione del collector, causata dal superamento della soglia T_i, viene memorizzata la quantità di memoria liberata (R_i) per decidere se quando verrà superata di nuovo la stessa soglia verrà chiamato il GC o verrà aumentata la heap. Verrà lanciata la garbage collection se in quelle recenti è stato liberato un certo ammontare di memoria ($S_i = T_i - T_{i-1}$).
Quando la heap viene aumentata la sua dimensione è posta pari alla soglia successiva (T_{i+1}).

Nella Figura 2.10 si può notare l'andamento di crescita della heap e quando viene innescato il GC. Un *cerchio* denota una crescita della heap (punti A,B ed E), un *quadrato* indica le operazioni di garbage collection (punti C, F e H) e il *rombo* che non è effettuata nessuna operazione (punto J).
Sempre dalla Figura 2.10 si può notare che la scelta delle soglie T_i, sempre più vicine man mano che si raggiunge il massimo valore di heap, permette di gestire il GC e la crescita della heap dinamicamente in base a quanta memoria è ancora disponibile. In questo modo vengono limitati gli overhead in fasi dove sono

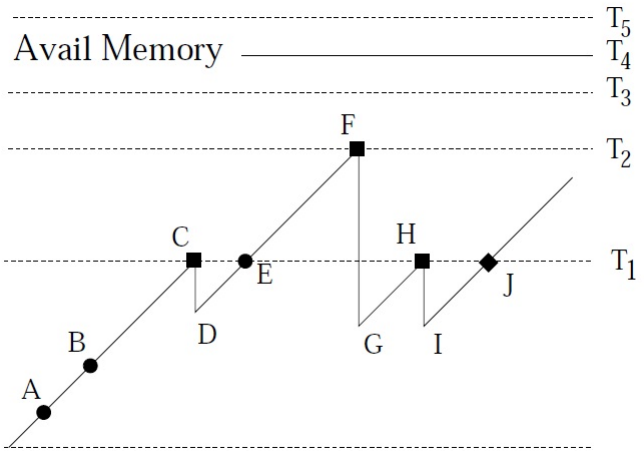


Figura 2.10: *Esempio delle operazioni di garbage collection e di crescita della heap*

evitabili: nella fase iniziale la heap cresce in modo aggressivo e il GC è eseguito poco, man mano che la memoria si riduce il GC sarà più aggressivo e la heap crescerà sempre più lentamente.

Dalla Figura 2.11 si può osservare che l'algoritmo basato sulle soglie è sempre ragionevolmente vicino al risultato ottenuto con il BDW settato con il FSD più adatto al caso.

I concetti dell'algoritmo sviluppato potrebbero essere applicati ad altri GC ma non è possibile garantire l'ottenimento di un effettivo miglioramento dato che sono basati sulle caratteristiche del BDW.

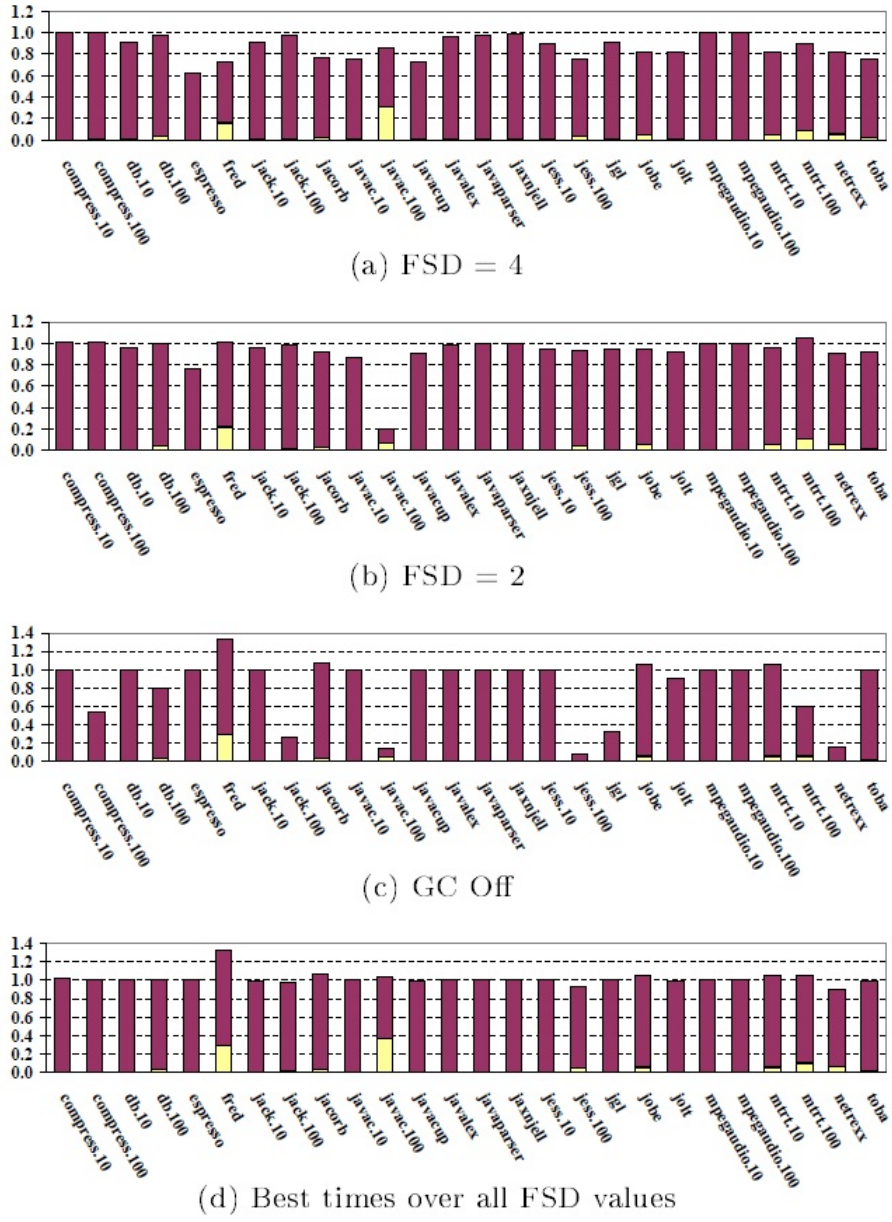


Figura 2.11: *Confronto fra tempi d'esecuzione con GC spento, BDW con FSD differenti e algoritmo a soglie in un sistema a 64MB di RAM*

2.4.4 Altri studi

Diversi confronti fra garbage collection e gestione esplicita della memoria sono stati effettuati con GC conservativi per C e C++. Nella sua tesi, Detlefs confronta le performance tra i due casi di tre programmi C++ [6]. Egli osserva che l'uso del GC in genere porta a prestazioni inferiori (dal 2% al 28%).
Zorn effettua lo stesso confronto in programmi C [24] osservando che il Boehm-Demers-Weiser [12] è a volte più veloce della gestione esplicita ma l'occupazione di memoria utilizzandolo è generalmente più alta (dal 21% al 228%).
Sono stati svolti anche diversi studi per identificare con precisione l'overhead sulle performance di un applicazione causato dal GC rispetto la gestione esplicita della memoria [7, 8, 16, 5, 24]. Steele osserva che l'overhead relativo al tempo d'esecuzione del GC in Lisp è di circa il 30% [14].
Usando un modello di costo per l'accesso in memoria, Appel presenta un analisi che mostra come, nel caso ci sia abbastanza memoria, la garbage collection può essere molto più veloce della deallocazione esplicita [1]. Egli osserva che la frequenza delle collections è inversamente proporzionale alla dimensione della heap, mentre il costo della collection rimane essenzialmente costante (valore derivante dal massimo set di variabili vive). Quindi, aumentando la dimensione della heap, si riduce il costo dell'esecuzione della collection.

Capitolo 3

Il Progetto Green Software

In questo capitolo vengono riportati gli studi effettuati fino ad ora nel progetto di *GreenSW* del Politecnico di Milano. Di seguito sono illustrate le conclusioni a cui hanno portato quattro tesi svolte dal 2008 al 2010 riguardanti lo studio dell'efficienza energetica di diversi ambienti IT.

3.1 Un approccio alla valutazione dei consumi energetici del software basato sull'analisi statica del codice

La tesi di Galli [9] si focalizza sulla ricerca di una metrica in grado di fornire un'indicazione dell'efficienza energetica delle applicazioni, in modo statico, cioè senza eseguirle.

Per studiare l'efficienza energetica del software è necessario, prima di tutto, avere a disposizione uno strumento che permetta di misurare le caratteristiche delle applicazioni che siano significative allo scopo: questa ricerca può spaziare tra le metriche già esistenti e metriche di nuova generazione, facendo leva su strumenti statistici per comporne i risultati.

Dopo averle create, le nuove metriche, sono state applicate ad un set di 84 applicativi campione, per poi operare una validazione di esse con un approccio statistico. La ricerca ha permesso di capire quale tipo di correlazioni esistono tra le carat-

teristiche dei software calcolabili staticamente e i consumi energetici degli stessi. Riassumendo, sono stati trovati almeno quattro risultati importanti:

- le metriche classiche di qualità del design del software non sono correlate in alcun modo coi consumi energetici;
- la qualità del software ha un impatto sui consumi energetici totali delle macchine;
- la metrica *Green* definita in questo lavoro, applicata al bytecode Java, è un buon indicatore del consumo energetico;
- la metrica di *entropia dell'espressività*, indice dell'utilizzo completo delle funzionalità del linguaggio di programmazione, è utile per categorizzare il comportamento delle applicazioni a seconda delle loro caratteristiche;

Se definitivamente provati, questi risultati potrebbero avere un notevole impatto sul modo di produrre software delle aziende moderne. Si aprirebbe tutta una serie di scenari organizzativi interessanti e suggestivi, che permetterebbero alle imprese anche di creare nuovi indirizzi di business.

Galli conclude affermando che nonostante i primi risultati siano molto incoraggianti, non è ancora possibile essere certi di aver trovato la correlazione per almeno due motivi:

- il campione di software, anche se considerato nella sua interezza, non è ancora sufficientemente abbastanza numeroso per fare affermazioni di tipo statistico
- il campione di software utilizzato è per natura limitato: le applicazioni erano tutte open source, e scritte in un unico linguaggio di programmazione (Java)

3.2 A methodology to evaluate empirically software energy consumption and its impact on Total Cost of Ownership

Il lavoro descritto in questo paragrafo riguarda la tesi di Formenti e Gallazzi [10]. La ricerca svolta ha dimostrato che il software a un impatto significativo sul costo totale dell'infrastruttura IT e in particolare sul Total Cost of Ownership.

Per prima cosa è stata definita una metodologia per comparare diverse applicazioni dal punto di vista energetico ed è stato implementato un kit per misurare il reale consumo del software.

Successivamente è stata validata la metrica (*Green Metric*) introdotta da Galli [9] per la valutazione del consumo energetico del software tramite l'analisi statica del codice.
Ciò è stato fatto misurando il consumo reale di 61 diverse applicazioni. L'analisi ha portato a due importanti conclusioni:

- la *Green Metric* rappresenta un buon indicatore dei fabbisogni energetici di un software
- l'*Expressivity Metric* indica effettivamente quanto le applicazioni sfruttano le funzionalità e l'espressività offerta dal relativo linguaggio di programmazione

In seguito sono stati analizzati differenti applicativi; in particolare degli Enterprise Resource Planning, dei Customer Relationship Management e dei Database Management Systems.
Per ogni applicazione sono stati definiti degli scenari tipici di utilizzo da cui sono stati creati dei workflows; dopo di che quest'ultimi sono stati eseguiti monitorandone il consumo energetico.
Formenti e Gallazzi con il loro studio hanno provato che, software differenti ma

funzionalmente comparabili, possono portare a consumi differenti fino al 34%.

Infine è stato quantificato l'impatto che l'efficienza energetica del software può avere sul costo totale di un data center analizzando il sistema tramite un modello di Total Cost of Ownership. Da quest'analisi è risultato che l'adozione di una soluzione software efficiente rispetto a una non efficiente può ridurre i costi del 20%.

3.3 Un'analisi esplorativa sull'ottimizzazione del consumo energetico del software

Nel lavoro di tesi svolto da Brianza e Alberio [2] è stata analizzata l'influenza delle diverse modalità di realizzazione del software sui consumi energetici e sulle performance, effettuando un confronto tra prodotti con le medesime funzionalità ma implementazioni differenti.
Lo sviluppo del software è stato analizzato a diversi livelli di dettaglio, dalle scelte architetturali, all'implementazione degli algoritmi, alla realizzazione a basso livello delle singole funzioni.
Dall'analisi architetturale si è mostrato come, sia la scelta delle librerie esterne che della tecnologia utilizzata siano rilevanti ai fini dell'efficienza energetica. Tali fattori possono incidere per oltre il 40% dei consumi.

Successivamente sono stati esaminati differenti implementazioni di algoritmi di utilizzo corrente presso istituti di credito per il calcolo dell'indice di redditività finanziaria di un investimento.
Si è osservato come una buona conoscenza del dominio applicativo consenta risparmi energetici significativi, fino a due ordini di grandezza. Nel caso in esame tali differenze sono dovute al diverso approccio matematico utilizzato.

Basandosi sui risultati precedenti [4], che hanno individuato il processore come principale responsabile dei consumi, è stata testata la tecnica della tabulazione per spostare il carico di lavoro verso altri componenti di un sistema.
Nel caso di tabulazioni realizzate in RAM, il risparmio ottenuto è stato superiore all'80%.
Per tabulazioni di grandi dimensioni è necessario ricorrere all'utilizzo di supporti di memorizzazione di massa. In questa direzione sono state testate diverse modalità di accesso a supporti di storage di tipo differente.

L'ultimo passo effettuato è stato l'analisi dell'implementazione a basso livello di una singola funzione, in particolare è stata confrontata l'operazione di elevamento a potenza con diverse operazioni elementari. Da questa analisi è emerso che per alcuni algoritmi, sotto determinate condizioni, l'esecuzione di moltiplicazioni ripetute risulta conveniente rispetto alla funzione nativa di elevamento a potenza offerta da Java. In tali condizioni è possibile ottenere un risparmio superiore al 40%.

3.4 Una metodologia per ridurre il consumo energetico del software, ed il suo impatto sul Total Cost of Ownership

Nel lavoro di tesi [3] è stata analizzata l'influenza, di diverse tecniche d'implementazione software, sui consumi energetici e le performance di un intero sistema informatico. Per farlo è stato eseguito un confronto diretto tra applicazioni con differenti strategie di realizzazione e medesime funzionalità. Il lavoro si pone anch'esso, come gli studi precedenti, all'interno del più ampio progetto di green software.

Sono state esaminate nel dettaglio differenti implementazioni di algoritmi *computational intensive*, sia per funzioni matematiche di uso comune, che per applicazioni utilizzate maggiormente presso istituti di credito; per esempio il calcolo dell'indice di redditività finanziaria di un investimento o il calcolo dell'importo delle rate di un contratto di mutuo a tasso fisso. Una prima analisi architetturale ha mostrato come, sia la scelta delle librerie esterne impiegate che la tecnologia utilizzata, siano rilevanti ai fini dell'efficienza energetica delle applicazioni, con un'incidenza di oltre il 40% sui consumi.

In seguito è stata realizzata e testata una tecnica di programmazione basata sulla memorizzazione. Lo scopo è ridurre i carichi computazionali destinati ai processori del sistema, principali responsabili dei dispendi energetici [9, 10], spostando il carico di lavoro verso altri componenti hardware dell'architettura.

La tecnica consiste nell'individuare i calcoli più onerosi presenti all'interno degli algoritmi analizzati, studiando nel dettaglio il loro impiego, e valutando la varianza ed il riuso dei valori in ingresso richiesti dalle funzioni interne. Successivamente occorre tabulare in dispositivi di memoria i dati richiesti e realizzare nuovi software applicativi che, durante l'esecuzione, effettuino delle letture di valori precalcolati, se disponibili, riducendo i carichi di lavoro destinati alla macchina. Negli algoritmi testati è emerso che molte operazioni eseguivano svariati calcoli impiegando dati in input con range limitati di valori, o con elevato riuso dei medesimi dati. In tal caso la tecnica è risultata vantaggiosa, permettendo di ottenere considerevoli risparmi energetici.

Per memorizzare tabulati di grandi dimensioni è necessario ricorrere all'utilizzo di supporti di memorizzazione di massa. In questa direzione sono state testate diverse modalità di accesso a supporti di storage di tipo differente. Nel caso, in-

vece, di tabulazione in banchi di memoria RAM, caratterizzati da elevate velocità di accesso ai dati memorizzati, con consumi energetici contenuti, la tecnica di memorizzazione ha permesso di ottenere un vantaggio energetico di oltre l'80% durante l'esecuzione degli algoritmi testati.

L'ultimo passo è stato la valutazione dei risparmi energetici ed economici ottenuti, in rapporto ai costi di un'infrastruttura IT, dimostrando che le applicazioni software hanno un'influenza rilevante sul Total Cost of Ownership dell'architettura informatica. È stato realizzato inoltre un modello di ripartizione dei costi che mostra in dettaglio l'impatto percentuale dell'efficienza energetica sui vari fattori che compongono il TCO complessivo di un'azienda. Infine sono stati calcolati i trade-off tra i risparmi ottenuti, impiegando la tecnica di ottimizzazione oggetto di questo lavoro, e i costi di acquisto e attivazione dei supporti di memoria necessari al salvataggio dei dati richiesti dagli algoritmi.

Capitolo 4

Definizione del Problema e Ipotesi di Ricerca

In questo capitolo verranno valutati i vantaggi e successivamente le problematiche dovuti alla presenza di un GC. Saranno analizzati gli aspetti riguardanti sia la fase di sviluppo che quelli successivi di esecuzione dell'applicativo approfondendo il funzionamento del GC contenuto nella Java Virtual Machine.
Infine verranno illustrate le ipotesi di ricerca relative al lavoro di tesi.

4.1 Guadagni diretti tramite GC

Con l'ausilio del GC si solleva il programmatore dal compito di deallocare gli oggetti che vengono istanziati durante l'esecuzione dell'applicativo. In questo modo, durante la fase di scrittura del codice, non è necessario preoccuparsi di ragionare su quando queste variabili non sono più utili, su quando è meglio deallocarle ma soprattutto non è necessario scrivere alcuna riga di codice per effettuare tale operazione.
In questo paragrafo si vuole indagare sugli effettivi vantaggi che comporta la presenza del GC durante la fase di stesura del codice. Quindi, non saranno analizzati parametri atti a valutare le performance a runtime dell'applicativo, ma verranno presi in considerazione i parametri di produttività dello sviluppatore. In questo

modo si vuole giungere a legare i vantaggi derivanti dalla semplificazione della fase di programmazione con gli aspetti strettamente economici concernenti questa fase.
Il programmatore viene agevolato sotto diversi aspetti nella creazione del software, di seguito sono elencati i più importanti:

- non è necessario preoccuparsi di quando è possibile deallocare un'area di memoria: si occuperà il GC, tramite apposite tecniche, di individuare quando una variabile non è più utilizzabile *(es. raggiungibile)*. In questo modo si evita un classico tipo di bug detto *dangling pointer* nel quale si cerca di accedere tramite puntatore a un oggetto che è stato deallocato.

- un altro classico tipo di errore evitabile, anche se meno grave per l'esecuzione del programma, è detto *memory leak* che indica la mancata deallocazione di una variabile non più utile.

- il programmatore oltre a non dover individuare quando è possibile deallocare non dovrà scrivere nessuna riga di codice per farlo

Come conseguenza di queste agevolazioni si potranno avere dei vantaggi diretti in termini di minor tempo di stesura del codice. Supponendo che, come è logico pensare non utilizzando un GC, per ogni operazione di allocazione ne deve esistere una di deallocazione per l'oggetto corrispondente, è stato ritenuto interessante indagare sull'influenza che ha il numero di linee di codice aggiuntive sul numero totale di quelle che compongono il sorgente.
Per analizzare il numero di statements di allocazione presenti in un programma, sono stati presi in considerazione diversi applicativi Java open source cercando di considerare un vasto campo di tipologie SW.
Per effettuare il conteggio di linee di codice totali che compongono un applicativo e il numero di quelle che contengono uno statement di creazione di un oggetto, è stato sviluppato un modulo *python* che automatizzasse tale procedura. La fun-

zione contenuta nel modulo prende in ingresso il percorso principale del software e analizza tutti i file con estensione *.java*, mantenendo un contatore per ogni riga di codice analizzata e uno per il numero di *new* osservate. Non vengono presi in considerazione dal programma i files con estensione *.jar* anche se essi contengono codice sorgente. Nel conteggio delle linee di codice totali o contenenti istruzioni di allocazione, non sono considerate le parti di commento, riconosciute dalla funzione tramite l'utilizzo di espressioni regolari.
I programmi analizzati sono elencati di seguito suddivisi per categoria d'appartenenza:

- Data Bases: Apache-Derby, Ozone, BerkleyDB
- ERP: Compiere ERP, OpenBravo ERP, Adempiere
- J2EE Frameworks: AppFuse, Expresso, Glassfish
- EJB Servers: OpenEJB, JBoss, Geronimo
- Search Engines: Compass, Egothor
- Personal Financial: JMoney ,GFP
- Mail Client: Pooka
- Office Utility: OpenOffice

Analizzando, con il software python creato, uno ad uno i vari applicativi presi in considerazione i risultati ottenuti sono quelli mostrati in Figura 4.1.
Per avere una visione meno dispersiva, nella Figura 4.2, i risultati sono stati raggruppati per categoria e sintetizzati in una tabella che rappresenta in numero medio di linee totali per tipologia di applicativo, le linee medie contenti la parola chiave per l'allocazione e la relativa percentuale.

In Figura 4.2 vengono mostrati i risultati medi dai quali si può osservare che, nonostante le dimensioni e il numero medio di linee totali varia di molto in base

Categoria:	Programma:	Linee di codice totali:	Linee con allocazione:	Percentuale:
Data Bases:	Apache-Derby	586028	12674	2,16
	Ozone	138383	4703	3,4
	BerkleyDB	286882	12329	4,3
ERP	Compiere ERP	435961	13251	3,04
	OpenBravo ERP	184982	8057	4,36
	Adempiere	922053	27683	3
J2EE Frameworks	AppFuse	20726	799	3,86
	Expresso	142256	6036	4,24
	Glassfish	993026	35805	3,61
EJB Servers	OpenEJB	299900	14290	4,76
	JBoss	619962	24462	3,95
	Geronimo	305776	16665	5,45
Search Engines	Compass	150950	6803	4,51
	Egothor	112711	3979	3,53
Personal Financial	JMoney	60356	2989	4,95
	GFP	70505	4215	5,98
Mail Client	Pooka	61396	3058	4,98
Office Tools	OpenOffice	557095	29402	5,28

Figura 4.1: *Linee totali, linee con statement new e percentuali relative per ogni applicativo*

Categoria:	Linee di codice totali:	Linee con allocazione:	Percentuale:
Data Bases	337097,67	9902	2,94
ERP	514332	16330,33	3,18
J2EE Frameworks	385336	14213,33	3,69
EJB Servers	408546	18472,33	4,52
Search Engines	131830,5	5391	4,09
Personal Financial SW	65430,5	3602	5,51
Mail client	61396	3058	4,98
Office Tools	557095	29402	5,28

Figura 4.2: *Linee totali, linee con statement new e percentuali relative per ogni tipologia di applicativo*

alla tipologia di applicativo e qualche volta anche all'interno dello stesso gruppo di software, la percentuale di linee di codice che effettuano un'allocazione rimane sempre abbastanza simile nei vari casi. La percentuale varia da un minimo di 2.94% a un massimo di 5.51%. Si osserva che le percentuali più alte si trovano per l'applicativo *OpenOffice* e per gli applicativi di piccole dimensioni, come quelli di *Personal Financial* o i *Mail Client*. Questo risultato potrebbe dipendere da diversi aspetti dell'applicazione:

- l'utilizzo di interfacce grafiche elaborate
- la forte correlazione tra vita degli oggetti e breve durata delle operazioni effettuate dall'utente
- presenza di molte funzionalità attivabili e disattivabili durante l'esecuzione
- preponderanza di operazioni semplici che comportano poche linee di codice per la logica applicativa
- scarsa ottimizzazione del processo di scrittura del codice
- scarsa attenzione nel riutilizzo di oggetti
- molto altro...

Osservando i risultati ottenuti si può supporre che generalmente un programmatore, in linguaggi privi di GC, è costretto a scrivere il 4% circa di linee di codice in più. Come già mostrato in precedenza, oltre al tempo aggiuntivo di scrittura, ciò comporta anche un aumento del tempo globale di creazione dell'applicazione. Questo a causa dei ragionamenti necessari per individuare gli istanti corretti per la deallocazione delle variabili. Quindi, molto probabilmente, il carico di lavoro aggiuntivo dello sviluppatore sarà di qualche punto percentuale più alto del 4%.

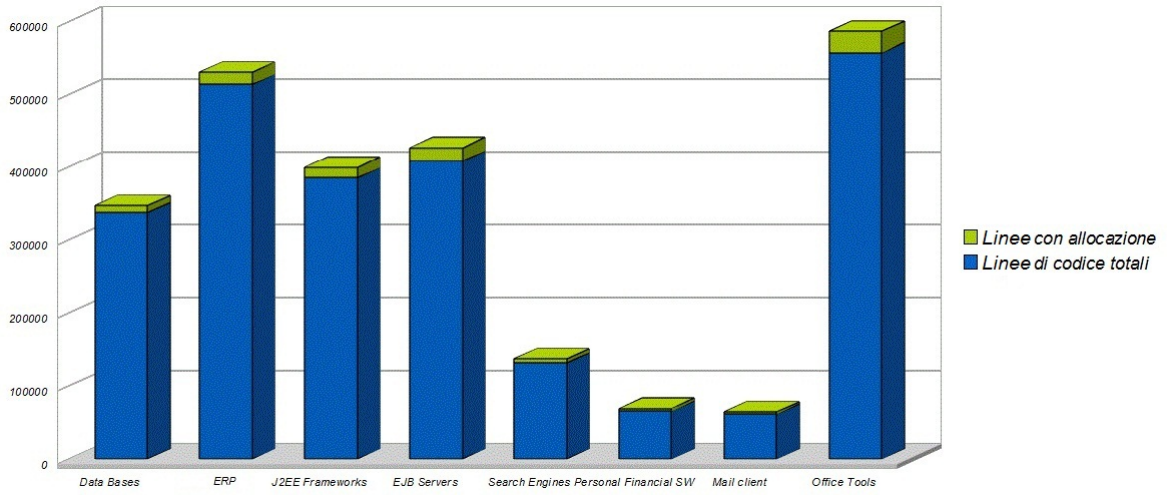


Figura 4.3: *Numero di linee totali e linee con statement new per ogni tipologia di applicativo*

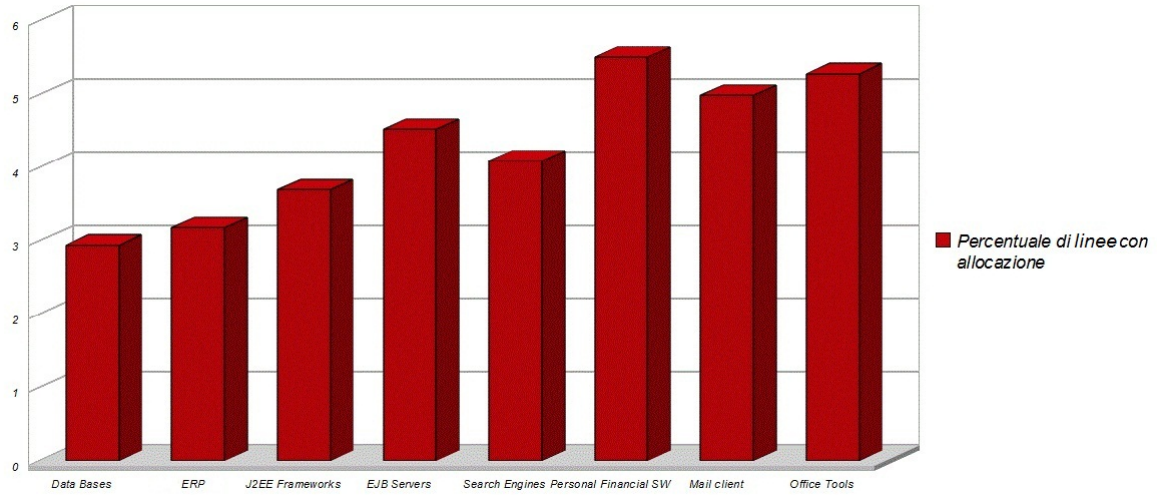


Figura 4.4: *Percentuale di linee con statement new per ogni tipologia di applicativo*

4.1.1 Introduzione di ulteriori errori

Oltre a considerare il maggior tempo necessario per la stesura del codice va considerata anche la più alta probabilità di errori nel software. Come errori del software si intendono quelli mostrati precedentemente: dangling pointers e memory leaks. E' da sottolineare che l'obiettivo di creare codice libero da questi bug diventa ancor più difficile da raggiungere nel caso lo sviluppo è affidato a un team di più persone. Supponiamo che ad uno sviluppatore venga affidata la creazione di una determinata funzionalità, di una determinata parte di codice o di un modulo. Il codice prodotto verrà poi utilizzato da altri programmatori considerando il suo funzionamento, per obiettivi di tempo, ai morsetti o comunque poco approfonditamente. In questo modo è possibile che le persone che riusano la parte di software non conosceranno la gestione delle variabili interna e come conseguenza è probabile che essi utilizzino in modo erroneo determinati oggetti (es. tengo il puntatore a un oggetto che viene deallocato invece che crearne una nuova copia).
Si può portare come esempio uno studio fatto da G. Phipps [19] nel quale vengono presi in considerazione due linguaggi per osservare in quale il programmatore riesce a scrivere più facilmente un programma funzionante e privo di bug. I linguaggi analizzati nello studio sono *Java*, che possiede un GC, e *C++*, che invece non lo possiede.
In questo esperimento vengono sviluppati due progetti sw reali e poi analizzati per individuare eventuali *bugs* o *defects*. Un *bug* è definito nello studio come un problema individuato durante la fase di testing o deployment. Un *defect* è anch'esso un bug ma può essere anche un errore osservato durante la fase di compilazione.
Dai risultati del test Phipps conclude che:

- un tipico programma C++ ha due o tre volte più bug per linea di codice rispetto a un tipico programma Java
- in C++ vengono generati dal 15% al 50% defects in più per riga di codice e per risolverli è necessario circa sei volte più tempo

- considerando la produttività come linee di codice corrette al minuto, Java risulta essere più produttivo dal 30% al 200%

Nella Figura 4.5, nella Figura 4.6 e nella Figura 4.7 sono rappresentati in sintesi i dati da cui l'autore dello studio ha tratto le sue conclusioni.

	Defects		Bugs		
	Mean	Std Dev	Mean	Std Dev	No. of samples
C++	82	25	18	8	7
Java	61	11	6	2.5	5

Figura 4.5: *Defects e bugs per migliaio di linee di codice (kLoc) osservate [19]*

Phase fixed	C++	Java
Compile	6	1.5
Unit Test (Bugs)	30	15
Post-deployment	240	?

Figura 4.6: *Minuti per risolvere i defects in base alla fase e al linguaggio [19]*

Data Set	Mean	StdDev	No. samples
C++	0.93	0.46	7
Java	1.55	0.78	5

Figura 4.7: *Statistiche di produttività, l'unità di misura è linee di codice per minuto [19]*

Va considerato che l'esperimento è stato svolto su un campione ridotto di applicativi e soprattutto da un solo sviluppatore, perciò, i risultati non possono essere considerati certi. Nonostante questo i dati forniscono un'idea di massima sul peso che può avere in termini di errori possibili un linguaggio come il C++, rispetto a uno più moderno e fornito di GC come il Java. Certamente non tutti i bug

aggiuntivi presenti nell'applicativo scritto in C++ sono dovuti a memory leaks o dangling pointers ma sicuramente una parte è probabile che lo siano. Va sottolineato anche che l'autore dell'articolo ha dichiarato che lo sviluppatore possedeva una buona esperienza in C++ mentre stava ancora imparando ad utilizzare Java; ovviamente questo fa presumere che il divario di errori tra i due linguaggi, per programmatori di pari esperienza, può essere molto maggiore di quello osservato. Alla luce di questo studio, si può confermare che con buona probabilità, il tempo aggiuntivo necessario a scrivere un programma funzionante in un linguaggio privo di GC sarà superiore del 4% come supposto precedentemente.

4.1.2 Minore manutenibilità del SW

Un altro aspetto da non sottovalutare è la maggiore complessità del sw a causa della gestione esplicita di allocazioni e deallocazioni. Ciò comporta che, all'atto di operazioni di manutenzione ordinaria o straordinaria, il programmatore incontrerà molte più difficoltà. Ciò contribuisce all'innalzamento dell'overhead di costo e impegno derivante dall'assenza del GC. Questo fattore sicuramente influenza maggiormente la fase di manutenzione del software che quella di sviluppo, ma è da considerare anche in quest'ultima. La motivazione di ciò è che anche durante la fase di debug, precedente al deploy a alla manutenzione, può avvenire di doversi confrontare con errori derivanti da un utilizzo inappropriato delle primitive per la gestione della memoria. In conclusione si può dire che la minore manutenibilità del software comporta costi aggiuntivi anche in fase di sviluppo e quindi porterà ad avere un overhead più alto di quello supposto per le sole linee di codice aggiuntive.

4.1.3 Influenza sui costi di sviluppo

Applicando il modello Basic CoCoMo (*Basic COnstructive COst MOdel*) si può osservare l'influenza che può avere l'overhead, dovuto alle sole deallocazioni, sul tempo e il costo di sviluppo.

Il Basic CoCoMo è il modello più semplice della famiglia di modelli CoCoMo e può essere applicato a tre diverse tipologie di progetti:

- *progetti Organic*: sono relativamente piccoli, rappresentano semplici progetti software nei quali piccoli team con buone esperienze lavorano a un insieme di rigidi requisiti.

- *progetti Semi-Detached*: sono nel mezzo delle tre tipologie (sia per dimensione che per complessità) e si adattano a progetti software nei quali team con esperienze medie lavorano su requisiti di livello medio.

- *progetti Embedded*: sono progetti software che devono essere sviluppati con un hardware ristretto e con costrizioni software e operazionali.

A ognuna di queste tipologie di sw sono stati assegnati dei parametri. I progetti presi in esame precedentemente rientrano o nella categoria Organic o in quella Semi-Detached.
Il modello calcola partendo dalle migliaia di linee di codice non commentate il costo, il tempo necessario per la consegna di un progetto sw e altri parametri.
Le formule utilizzate dal modello per stimare lo sforzo in mesi persona (E), il tempo necessario alla consegna (D), la produttività mensile in linee al mese per ogni sviluppatore (P) e il costo totale di sviluppo (C) sono quelle mostrate di seguito.

$$E = a(KNCSS)^b$$

$$D = cE^d$$

$$P = \frac{E}{D}$$

$$C = (AvgMonthCost)(E)$$

$$KNCSS \ = \ migliaia\ di\ linee\ non\ commentate$$

$$AvgMonthCost \ = \ 5000$$

$$Organic: \ a = 2.4 \ b = 1.05 \ c = 2.5 \ d = 0.38$$

$$Semi - Detached: \ a = 3.0 \ b = 1.12 \ c = 2.5 \ d = 0.35$$

Applicando per ogni software esaminato le formule e i parametri corretti si possono calcolare i vari overhead che si vengono ad aggiungere. Nella Figura 4.8 e nella Figura 4.9 sono riportati i risultati supponendo un costo mensile medio di uno sviluppatore pari a 5000 euro; i valori riportati indicano la differenza tra il caso con e il caso senza GC.

Tipologia	Programma:	Linee di codice totali:	Linee Aggiuntive:	E+ [PM]	D+ [Months]
S-D	Apache-Derby	586028	12674	91,61	0,38
S-D	Ozone	138383	4703	28,61	0,33
S-D	BerkleyDB	286882	12329	81,9	0,56
S-D	Compiere ERP	435961	13251	92,49	0,47
S-D	OpenBravo ERP	184982	8057	50,78	0,48
S-D	Adempiere	922053	27683	211,4	0,62
O	AppFuse	20726	799	3,87	0,18
S-D	Expresso	142256	6036	36,86	0,42
S-D	Glassfish	993026	35805	275,96	0,77
S-D	OpenEJB	299900	14290	95,46	0,63
S-D	JBoss	619962	24462	178,21	0,7
S-D	Geronimo	305776	16665	111,63	0,73
S-D	Compass	150950	6803	41,85	0,46
S-D	Egothor	112711	3979	23,62	0,32
O	JMoney	60356	2989	16,47	0,35
O	GFP	70505	4215	23,68	0,45
O	Pooka	61396	3058	16,89	0,35
S-D	OpenOffice	557095	29402	211,63	0,89

Figura 4.8: *Overhead sui mesi uomo necessari e sul tempo di consegna nel caso senza GC*

Si deve ricordare che il modello Basic CoCoMo rappresenta un metodo molto grezzo per il calcolo del costo di sviluppo di un'applicazione e ormai da qualche anno non è più utilizzato così frequentemente come in passato. La bassa precisione dei suoi risultati è dovuta al fatto che non prende in considerazione tutti gli aspetti che potrebbero influire sullo sviluppo di un sw come la maturità del

Tipologia	Programma:	Linee di codice totali:	Linee Aggiuntive:	C+ [€]	C+ [%]
S-D	Apache-Derby	586028	12674	458071,04	2,43
S-D	Ozone	138383	4703	143050,77	3,81
S-D	BerkleyDB	286882	12329	409511,53	4,83
S-D	Compiere ERP	435961	13251	462461,95	3,41
S-D	OpenBravo ERP	184982	8057	253896,95	4,89
S-D	Adempiere	922053	27683	1056982,97	3,37
O	AppFuse	20726	799	19356,65	4,33
S-D	Expresso	142256	6036	184296,79	4,76
S-D	Glassfish	993026	35805	1379802,78	4,05
S-D	OpenEJB	299900	14290	477311,19	5,35
S-D	JBoss	619962	24462	891046,45	4,43
S-D	Geronimo	305776	16665	558160,09	6,12
S-D	Compass	150950	6803	209231,63	5,06
S-D	Egothor	112711	3979	118094,25	3,96
O	JMoney	60356	2989	82374,33	5,56
O	GFP	70505	4215	118419,11	6,72
O	Pooka	61396	3058	84450,27	5,59
S-D	OpenOffice	557095	29402	1058156,82	5,93

Figura 4.9: *Overhead sui costi totali nel caso senza GC*

progetto, l'esperienza del team, il linguaggio utilizzato ecc.

Fatta questa premessa, osservando i risultati si può notare che è presente un forte overhead per i mesi uomo necessari e conseguentemente per il costo totale; nei casi presi in considerazione il costo totale viene incrementato dal 2% fino al 7%. Quindi il costo che si viene ad aggiungere, per il fatto di utilizzare un linguaggio privo di GC, non è trascurabile in fase di scelta dell'ambiente di sviluppo.

Al contrario, il tempo di consegna non aumenta di molto perché il modello CoCoMo modifica automaticamente le dimensioni del team al crescere della grandezza (in KNCSS) del progetto. In questo modo, nonostante la crescita di E, il valore di D varia di poco; nei programmi analizzati la variazione non arriva a toccare neanche il mese.

4.1.4 Ulteriori analisi

Finora abbiamo osservato che esistono vantaggi economici diretti derivanti dall'utilizzo di un linguaggio fornito di GC. Sorge però spontanea una domanda: quanto mi costa in più eseguire un programma più pesante, come sono tipicamente quelli che utilizzano GC?
I costi aggiuntivi possono derivare dalla necessità di avere più memoria RAM disponibile, di avere un processore più potente e non ultimo di consumare più energia elettrica per ottenere le stesse performance di un programma scritto in un linguaggio privo di GC.
Nei paragrafi successivi si cercherà di trovare i legami tra GC e i parametri appena citati per dare un'idea di massima sull'influenza che un suo utilizzo può avere sui costi d'esecuzione.

4.2 Relazione tra Memoria e Tempo d'Esecuzione

Già nel capitolo riguardante lo stato dell'arte si è discusso del rapporto che esiste fra il tempo d'esecuzione di un programma e il quantitativo di memoria disponibile. In questo paragrafo sarà approfondito questo aspetto osservando i risultati in modo critico per ottenere delle linee guida per la scelta del quantitativo di memoria ottimale per l'esecuzione di un determinato applicativo.
Nello studio *Quantifying the Performance of GC vs Explicit Memory Management* [15], di cui abbiamo già parlato, gli autori hanno analizzato l'influenza delle dimensioni della heap sul tempo d'esecuzione di un'applicazione. I software presi in considerazione riguardano dei benchmark contenuti nel tool *SpecJVM98* che simula dei contesti applicativi di vario genere.
Gli autori hanno realizzato un programma detto oracolo che identifica secondo due tecniche differenti quando le variabili possono essere deallocate in modo esplicito;

in seguito hanno utilizzato queste informazioni per confrontare la deallocazione esplicita con l'uso del GC. Le tecniche utilizzate sono chiamate *liveness based* e *reachability based*. Mentre la prima tecnica permette di identificare il primo punto possibile per liberare le locazioni di memoria in modo sicuro, la seconda le libera nell'ultimo punto utile per farlo. In questo modo vengono identificati i due istanti di tempo estremi per effettuare l'operazione di *free*; generalmente i programmi reali che utilizzano la deallocazione esplicita andranno a posizionarsi fra questi due estremi.

In Figura 4.10 sono elencati i GC utilizzati nei test e i memory manager presi in

Garbage collectors	
MarkSweep	non-relocating, non-copying single-generation
GenCopy	two generations with copying mature space
SemiSpace	two-space single-generation
GenMS	two generations with non-copying mature space
CopyMS	nursery with whole-heap collection
Allocators	
Lea	combined quicklists and approximate best-fit
MSExplicit	MMTk's MarkSweep with explicit freeing

Figura 4.10: *Memory Manager e GC utilizzati*

considerazione per l'allocazione e la deallocazione esplicita della memoria. Tutti i GC presi in considerazione sono del tipo *stop-the-world* e il più performante è risultato essere il GenMS. Confrontando le prestazioni dei due memory manager gli autori hanno notato che nonostante i due portino a tempi d'esecuzione molto simili il *Lea* è molto più performante in quanto permette di utilizzare molta meno memoria rispetto al *MSExplicit*.

Nella Figura 4.11 si possono osservare i risultati ottenuti dal GenMS rispetto la gestione esplicita della memoria utilizzando il Lea. Nella tabella vengono mostrate le medie dei tempi d'esecuzione e della footprint, al variare della grandezza della heap e della tecnica utilizzata per l'identificazione degli istanti per la deallocazione. I fattori presenti nella colonna heap size sono multipli della minima quantità di memoria possibile per utilizzare il GenMS. La footprint indica il numero massimo

	GenMS			
	vs. Lea w/ Reachability		vs. Lea w/ Liveness	
Heap size	Footprint	Runtime	Footprint	Runtime
1.00	210%	169%	253%	167%
1.25	252%	130%	304%	128%
1.50	288%	117%	347%	115%
1.75	347%	110%	417%	109%
2.00	361%	108%	435%	106%
2.25	406%	106%	488%	104%
2.50	419%	104%	505%	102%
2.75	461%	103%	554%	102%
3.00	476%	102%	573%	100%
3.25	498%	101%	600%	100%
3.50	509%	100%	612%	99%
3.75	537%	101%	646%	100%
4.00	555%	100%	668%	99%

Figura 4.11: *GenMS vs. Lea: dimensioni della footprint e del tempo d'esecuzione*

di pagine di memoria utilizzate durante l'esecuzione; una pagina è in uso solo se è stata allocata dal kernel e acceduta in qualche istante.

In Figura 4.12 sono rappresentati gli stessi dati della tabella ma in forma grafica per rendere più visibile l'andamento delle curve che rappresentano la dimensione della footprint e del tempo d'esecuzione. Si può osservare che la dimensione della footprint aumenta considerevolmente con l'introduzione del GC (+110% minimo) e continua a crescere più la dimensione della heap massima viene aumentata. Invece, per quanto riguarda il tempo d'esecuzione, si ottengono risultati già molto buoni con l'utilizzo di tre volte la heap minima possibile; ulteriori incrementi portano a miglioramenti molto bassi. Questo risultato probabilmente è da imputare al fatto che, con un quantitativo di memoria disponibile così alto, il GC non effettua mai un'operazione di collection. Questo perché di fatto non si manifesta la necessità di liberare memoria per terminare l'esecuzione del benchmark.

Le osservazioni confermano la formula proposta dagli autori per identificare la percentuale di overhead sul tempo d'esecuzione nel caso del GC GenMS nella quale si mostra che esso dipende in modo inversamente proporzionale al quadrato

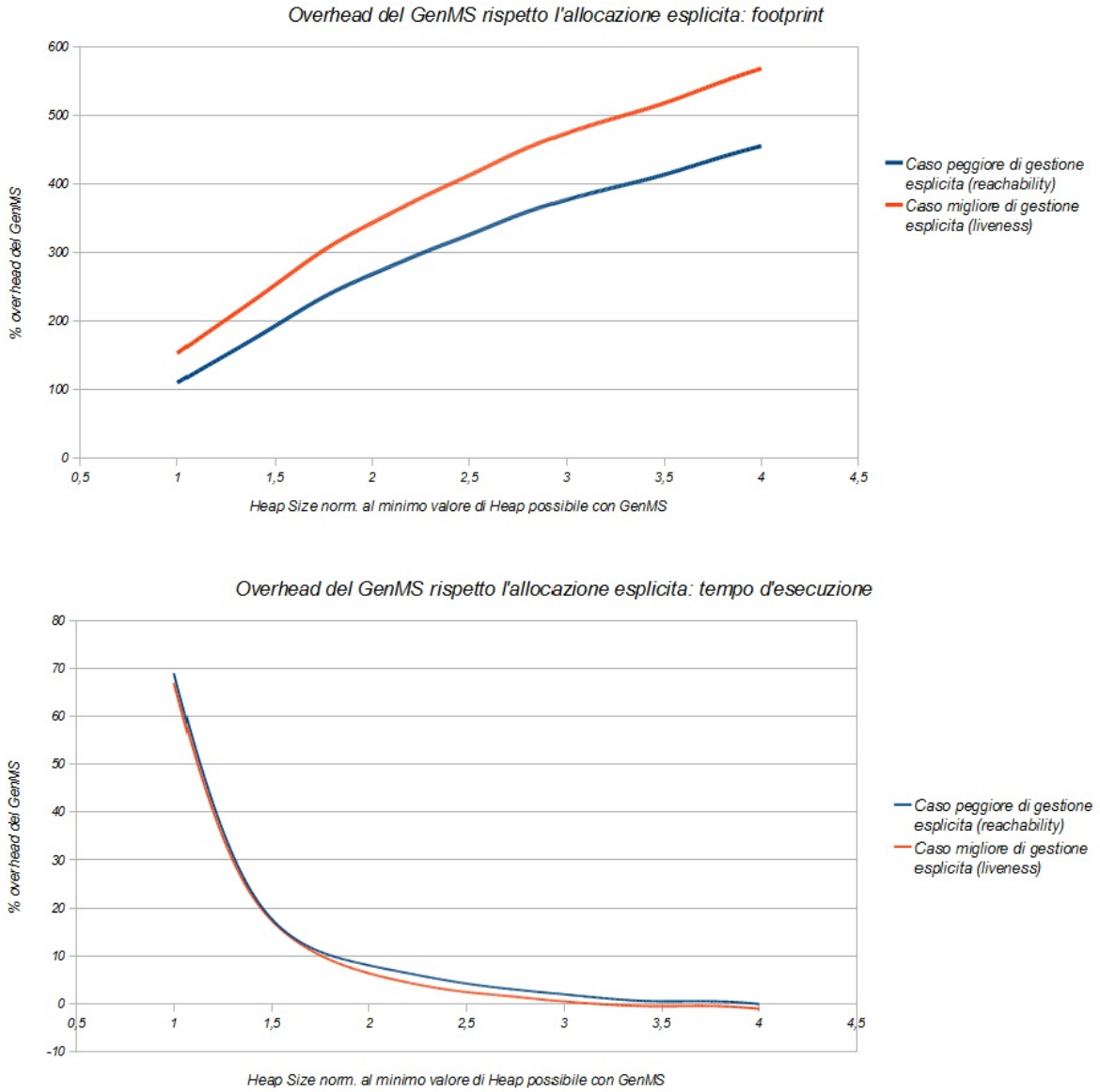


Figura 4.12: *GenMS vs. Lea: overhead della footprint e del tempo d'esecuzione*

della heap size. La formula identificata è:

$$ExecTimeFactor = \frac{a}{b - HeapSizeFactor} + c$$

$$GenMS : a = -0.246, b = 0.59, c = 0.942$$

Gli autori infine concludono che se non si possiede sufficiente memoria (almeno 3 volte la necessaria) l'uso del GC è sconsigliato perché porterebbe a un peggioramento delle performance soprattutto in ambiti dove si fa uso intenso della RAM, come nei *DB in-memory* e nei *motori di ricerca*, e di conseguenza il GC è sottoposto a un carico maggiore.

4.3 Parametri JVM GC e loro influenza

La Java Virtual Machine permette all'utente di impostare numerosi parametri relativi al funzionamento e alla tipologia del GC [18]. Per quanto riguarda il GC standard, quello descritto nel capitolo relativo allo stato dell'arte, vengono forniti diversi strumenti per settare le caratteristiche del suo funzionamento in modo da adattarlo alla tipologia di applicazione per cui sarà utilizzato. Gran parte dei parametri configurabili si riferiscono alle impostazioni della heap utilizzata dalla JVM. Questo paragrafo ha come scopo quello di illustrare in che modo ogni parametro influisce sul funzionamento del GC e il significato dei valori ad essi attribuibili.

Per comprendere al meglio a cosa si riferiscono i parametri che verranno descritti di seguito è bene comprendere come è strutturata la memoria all'interno della Java Virtual Machine.

Nella Figura 4.13 è possibile osservare la suddivisione della memoria nelle varie parti:

- *Eden*: area di memoria dove vengono allocati inizialmente gli oggetti. Appartiene all'area young della heap.

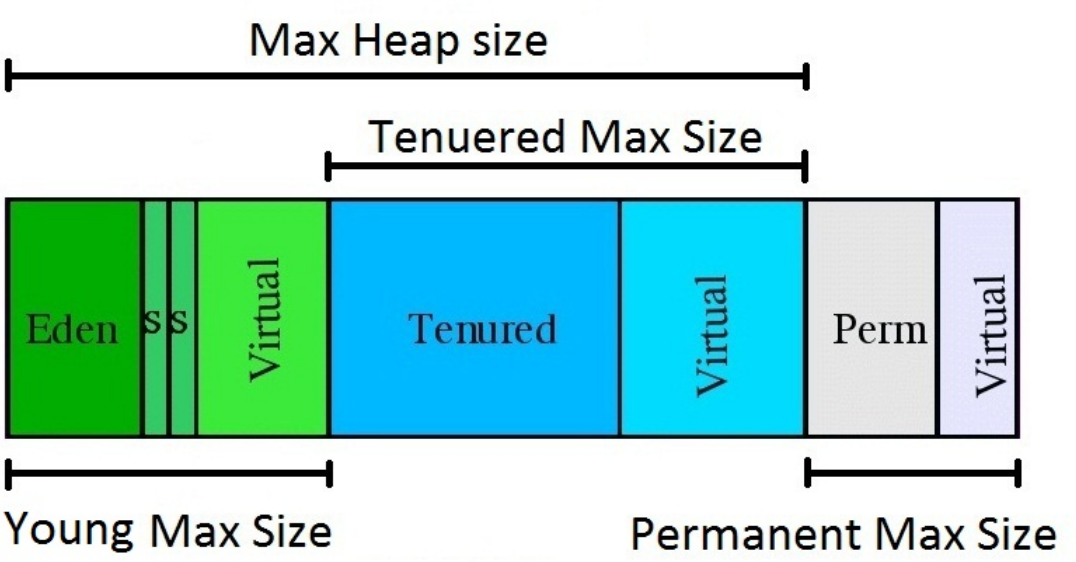


Figura 4.13: *Suddivisione della memoria di un'applicazione java*

- *Survivor Spaces*: sono due parti di memoria che entrambe appartengono all'area young della heap. In ogni momento un solo *survivor space* rimane vuoto e verrà utilizzato per ospitare gli oggetti sopravvissuti contenuti nell' *eden* e nell'altro *survivor space*, finché essi non saranno pronti per essere copiati nell'area *tenuered* della memoria.

- *Virtual*: è la parte di memoria young ancora disponibile per l'espansione.

- *Tenuered Space*: rappresenta l'area di memoria contenente gli oggetti long lived promossi da un *survivor space*.

- *Virtual*: è la parte di memoria tenuered ancora disponibile per l'espansione.

- *Permanent Space*: contiene gli oggetti definiti permanenti come le definizioni delle classi e i metodi statici.

- *Virtual*: è la parte di memoria permanent ancora disponibile per l'espansione.

4.3.1 Heap totale: dimensioni, espansione e contrazione

Innanzi tutto è bene descrivere i comandi relativi alle caratteristiche della memoria totale utilizzata dalla JVM in termini di dimensioni e modo in cui essa cresce o decresce.

Il parametro che viene utilizzato per settare il volume massimo di heap occupabile dall'applicazione è *-XmxHeapSizeM*, dove al posto di *HeapSize* deve essere inserito il valore desiderato. La lettera finale M indica che il quantitativo specificato è da considerare in megabyte; è possibile utilizzare anche le lettere K e G per specificare valori rispettivamente in kilobyte e gigabyte. Inizialmente, l'applicazione non utilizzerà tutta la memoria specificata nella dimensione massima, ma effettuerà un percorso di espansione della memoria occupata solo se ne sarà presente la necessità, occupando al massimo il quantitativo di memoria specificato.

Il parametro per settare la quantità di memoria minima dalla quale la JVM inizierà il suo percorso di espansione è *-Xms* seguito, come nel caso precedente, dal valore e dall'unità di misura. Questo parametro è bene sia settato vicino alla memoria minima utilizzata dall'applicazione, in modo che esso non comporti overhead inutili derivanti da un processo di espansione praticamente certo. Questo parametro, oltre ad indicare la dimensione di partenza della heap, indica anche la dimensione minima che può raggiungere durante il processo di contrazione della memoria utilizzata; ciò rappresenta un'altra motivazione valida per cui esso debba essere settato vicino alla memoria minima utilizzata dall'applicazione.

Per decidere quando espandere o contrarre la memoria disponibile la Java Virtual Machine si basa sui valori di due parametri: *-XX:MinHeapFreeRatio* e *-XX:MaxHeapFreeRatio*.

Il primo parametro specifica quando espandere la heap: se il suo valore è maggiore della percentuale di memoria libera la heap viene espansa. Al contrario il secondo parametro specifica quando contrarre la heap: se il suo valore è minore della percentuale di memoria libera essa verrà contratta. Entrambi i parametri,

per essere settati, devono essere seguiti da un simbolo di uguaglianza e poi da un valore senza nessuna unità di misura dato che rappresentano delle percentuali (*es. -XX:MinHeapFreeRatio=60*). Il valore di default per il *MinHeapFreeRatio* è 40 mentre per il *MaxHeapFreeRatio* è 70. La heap viene espansa o contratta in base ai casi di una quantità di memoria tale da permettere il raggiungimento delle percentuali di memoria libera specificate in *MinHeapFreeRatio* e *MaxHeapFreeRatio*. In sistemi a 64bit i valori di default sono innalzati del 30% per compensare la maggiore grandezza degli oggetti a causa delle dimensioni doppie dei puntatori.

4.3.2 Dimensioni sezioni interne della heap

Tramite altri parametri è possibile settare le dimensioni di partenza (e minime) o quelle massime delle aree di memoria young, tenuered e permanent applicando gli stessi concetti presentati nel paragrafo precedente.
I comandi principali sono:

- *-XX:NewSize* e *-XX:MaxNewSize*: per settare la memoria iniziale e minima o la memoria massima della parte di heap young.
- *-XX:PermSize* e *-XX:MaxPermSize*: per settare la memoria iniziale e minima o la memoria massima della parte di heap permanent.
- *-XX:SurvivorRatio=K*: il valore K sta ad indicare la dimensione della parte di memoria associata all'*eden* rispetto a quella per i *survivor spaces*. Un valore K indica che

  $$K/(K+2)$$

  parti di memoria young sono associate all'*eden* mentre la rimanente parte è suddivisa fra i due *survivor spaces* (*es. con K=6, 1/8 della heap young a ciascun SS mentre 6/8 all'eden*).

Le dimensioni minima e iniziale o massima della parte di memoria tenuered dipendono direttamente dai valori impostati coi comandi precedenti.

4.3.3 Rapporto tra young e tenuered

Tramite il parametro -XX:NewRatio è possibile specificare il rapporto desiderato tra le dimensioni della memoria young e della memoria tenuered. Settandolo a un valore K la parte tenuered di memoria andrà ad occupare

$$K/(K+1)$$

parti della memoria disponibile al netto di quella occupata dalla *permanent generation*, le restanti parti verranno occupate dall'area young.
Settando questo parametro vengono automaticamente settati tutti i parametri mostrati nel paragrafo precedente in modo da mantenere le proporzioni standard tra *eden* e *survivor spaces*.
Il valore di questo parametro influenza fortemente il comportamento del GC e le conseguenze sulle prestazioni dell'applicazione possono variare in modo molto profondo. L'importanza di un buon settaggio del rapporto tra le due aree di memoria principali è importante quasi quanto un buon settaggio della memoria massima associata alla JVM.

4.3.4 Consigli sulla configurazione del GC

Come già detto i parametri che influenzano maggiormente le prestazioni del GC sono la dimensione della memoria massima e il valore del *NewRatio*.
In genere si tende ad associare un quantitativo di memoria il più alto possibile, in modo da creare le condizioni tali per cui l'esecuzione della garbage collection è necessaria ad una frequenza abbastanza bassa rispetto al caso con poca memoria disponibile. In questo modo è possibile ridurre il tempo totale di funzionamento del GC. L'effetto collaterale di optare per un quantitativo molto alto è il rischio

di incorrere in pause molto lunghe dovute al tempo maggiore impiegato dal GC per analizzare tutta la memoria occupata.
Se non si presentano pause molto lunghe durante il funzionamento dell'applicazione o esse non sono ritenute un problema il consiglio è di utilizzare un quantitativo alto di memoria.
Ogni volta che la memoria young viene occupata totalmente viene effettuata una GC su di essa mentre se è la parte tenuered ad essere occupata completamente viene effettuata una garbage collection sull'intera heap (*FullGC*).
Un altro fattore da considerare è che le applicazioni generalmente creano oggetti di due tipi: short lived e long lived.
La frequenza di chiamata del GC non dipende quindi solo dalla dimensione totale dalla memoria ma anche da come viene suddivisa la memoria interna e dalla tipologia dell'applicazione.
Il parametro *NewRatio* deve essere utilizzato per settare correttamente la suddivisione della memoria in base al tipo di applicazione. Nel caso di applicazioni di tipo server (es. web server, ERP, etc.) sono presenti in maggiore percentuale oggetti di tipo long lived, mentre in applicazione di tipo client (es. GUI, etc.) sono presenti in quantità maggiore oggetti di tipo short lived.
Un *NewRatio* basso tende a privilegiare applicazioni di tipo server, perché aumentando la parte young di memoria sarà maggiore la probabilità che un oggetto muoia nelle prime GC senza essere copiato nell'area tenuered. Una dimensione troppo piccola della parte young non permetterebbe al GC di effettuare una scrematura efficiente degli oggetti data la loro tendenza ad avere una vita abbastanza lunga. Ciò introdurrebbe un overhead significativo all'area di memoria contenente gli oggetti long lived.
Un valore di *NewRatio* alto invece, comporta un'area young piccola rispetto a quella tenuered e privilegia applicazioni di tipo client in cui la maggior parte degli oggetti hanno una vita molto breve. Con una configurazione simile e con

applicazioni di questo tipo sarà possibile, anche con un'area di memoria giovane ristretta, eliminare abbastanza oggetti da giustificare la chiamata al GC. Inoltre, essendo il numero di oggetti long lived basso e la memoria ad essi dedicata alta, la frequenza delle *FullGC* sarà molto bassa. Con una configurazione della memoria opposta vengono introdotti overhead causati dal maggior carico derivante dall'analisi di una quantità alta di memoria young.
Quindi per trovare il NewRatio più preformante per la nostra applicazione è bene effettuare alcuni test e osservare il comportamento del GC.

4.3.5 Analisi comportamento GC

Per comprendere al meglio lo stato di funzionamento del GC è possibile osservare la frequenza delle sue chiamate, la loro tipologia e la loro efficacia.
Aggiungendo delle opzioni alla chiamata della JVM è possibile ottenere l'output relativo al funzionamento del GC:

- *-verbose:gc* : viene stampato sullo standard output tutte le chiamate al GC e i relativi dati di funzionamento
- *-Xloggc=nomefile* : ridireziona l'output del GC sul file specificato in modo che possa essere analizzato in modo più efficiente
- *-XX:+PrintGCDetails* : restituisce un output più ricco di informazioni.

Quello che segue è un esempio di output ottenuto con l'utilizzo dei primi due parametri:

```
49.049: [GC 115536K->97241K(135808K), 0.0201240 secs]
50.366: [GC 115479K->97377K(135808K), 0.0172490 secs]
50.383: [Full GC 97377K->88932K(145664K), 0.5612850 secs]
```

Ogni riga rappresenta i dati relativi a una chiamata del GC e la parte iniziale di ognuna rappresenta il tempo trascorso in secondi tra l'avvio dell'applicazione e la

stessa.
La scrittura *GC* o *Full GC* identifica rispettivamente una chiamata di garbage collection sulla parte young oppure una sull'intera area di memoria. Successivamente alla tipologia della chiamata è specificato lo spazio occupato prima e dopo essa e, tra parentesi, lo spazio di memoria libera senza contare quello nella parte permanent. Lo spazio prima e dopo l'esecuzione della garbage collection riguarda la sola parte young di memoria nel gaso di *GC* mentre l'intera area di memoria nel caso di *FullGC*. Infine è specificato il tempo di durata dell'operazione di garbage collection.
Come si può osservare le *Full GC* sono molto più pesanti delle *GC* dal momento che devono analizzare l'intera memoria occupata.
Generalmente per osservare l'efficienza del GC con un certo setting della memoria si osserva il tempo totale d'esecuzione delle garbage collection: se rappresenta un valore superiore al 5-10% del tempo d'esecuzione dell'applicazione è utile cercare di effettuare un analisi per migliorarne le prestazioni.
Di seguito sono descritti alcuni comportamenti che possono essere osservati tramite il log dettagliato (*XX:+PrintGCDetails*) del GC:

- memoria totale troppo bassa : l'efficienza del GC è scarsa nonostante i diversi settaggi del *NewRatio*.
- memoria totale troppo alta : le pause introdotte dal GC sono alte nonostante i diversi settaggi del *NewRatio*.
- area young troppo piccola rispetto quella tenuered : la memoria liberata nella parte young viene in gran parte ricopiata nella parte tenuered.
- area young troppo grande rispetto quella tenuered : le *GC* sulla parte di memoria young causano quasi sempre una *FullGC*.

Per osservare in modo più preciso i comportamenti adottati dal GC è necessario impostare la JVM per fornire un log dettagliato tramite il comando specificato in precedenza. Di seguito sono riportati due esempi di output dettagliato:

```
[GC [DefNew: 3968K->64K(4032K), 0.0460948 secs]
7451K->6186K(32704K), 0.0462350 secs]

[GC [DefNew: 16000K->16000K(16192K), 0.0000574 secs]
[Tenured: 2973K->2704K(16384K), 0.1012650 secs]
18973K->2704K(32576K), 0.1015066 secs]
```

Il primo esempio mostra i dati relativi all'esecuzione di una *GC* dove la sigla *DefNew* indica il tipo di GC utilizzato. Le dimensioni specificate tra le parentesi quadre più interne si riferiscono all'area di memoria young, mentre quelle tra le parentesi più esterne all'intera heap. Il secondo valore di tempo specificato indica la durata complessiva dell'operazione di garbage collection e può includere, se necessario, anche il tempo impiegato per effettuare un'analisi della parte tenuered di memoria.
Il secondo esempio riguarda un caso in cui un'operazione di *GC* ne scatena una di *FullGC*. Come si può osservare le prime parentesi interne racchiudono i valori relativi alla *GC* mentre le seconde quelli riguardanti la *FullGC*. I dati contenuti nella parentesi più esterna rappresentano gli stessi dati del caso precedente.
Una volta analizzato attentamente l'output si potrà riconoscere la situazione di funzionamento del GC ed eventualmente modificare le impostazioni per migliorarne l'efficienza.

4.4 Ipotesi di Ricerca

Dagli studi e dalle considerazioni esposte finora, sono state impostate delle ipotesi relative agli effetti che diversi fattori possono avere sui consumi energetici. Questo

lavoro di tesi ha lo scopo di valutare la veridicità delle ipotesi descritte di seguito.

La relazione presente tra il tempo d'esecuzione di un'applicazione che utilizza un GC e la quantità di memoria massima disponibile (paragrafo 4.2), pone le basi per la prima ipotesi: l'aumento della heap massima permette una riduzione dei consumi. La motivazione di questa ipotesi è basata sul fatto che, con un quantitativo maggiore di memoria disponibile, il lavoro effettuato dal GC è minore facendo quindi diminuire il tempo d'esecuzione e, di conseguenza, anche i consumi energetici. Questo perché il processore, che è il principale responsabile dei consumi [4], è sottoposto a un carico minore e quindi diminuisce il suo assorbimento di potenza.

Hyp 1 : l'aumento della heap massima diminuisce i consumi di un'applicazione che utilizza un GC

Dopo aver osservato come lavorano due GC, il Boehm (paragrafo 2.4.3) e quello della JVM (paragrafo 2.3), si ipotizza che il loro settaggio influisca fortemente sui consumi. Questo perché, in base alla suddivisione degli oggetti tra long-lived e short-lived, alcune configurazioni sono più performanti di altre. Gli oggetti o le variabili vengono chiamati long-lived se essi, una volta allocati, vengono utilizzati o mantenuti referenziati per un periodo di tempo sufficientemente lungo; essi vengono chiamati anche oggetti o variabili *old*. Gli oggetti e le variabili short-lived, che vengono chiamati anche *young*, si comportano nel modo opposto rimanendo referenziati per poco tempo e quindi anno un ciclo di vita molto breve.

Hyp 2 : la configurazione del GC influisce sui consumi energetici

Conoscendo i principi generali di funzionamento del Bohem GC si ipotizza che, nel caso di molti oggetti short-lived, esso riesce a essere energeticamente più efficiente della gestione esplicita della memoria, ovvero senza GC. L'ipotesi deriva dal fatto che tale GC è in grado di deallocare più efficientemente la memoria

allocata rispetto alla gestione della heap senza GC. Questo poiché utilizza una tecnica di deallocazione detta *lazy-sweep* già descritta nel paragrafo 2.2. Nel caso però, di molti oggetti long-lived, il GC dovrà analizzare diverse volte la memoria senza riuscire a liberare abbastanza spazio da giustificarne la chiamata.

Hyp 3 : il GC consuma meno della gestione esplicita se sono presenti molti oggetti short-lived

La quarta ipotesi che si vuole verificare riguarda l'influenza del sistema operativo sui consumi. Più in dettaglio, si ipotizza che la scelta di un determinato sistema operativo influisca sui consumi finali dell'applicazione. Questa ipotesi deriva dal fatto che sistemi operativi profondamente differenti, come possono essere una versione Windows e una Linux, possiedono concetti e tecniche diverse per la gestione della memoria. Si ipotizza perciò che le differenze presenti, come il tipo di file system, la politica di utilizzo della RAM e il tipo di spazio di swap ([17, 20]), influiscano direttamente o indirettamente sul comportamento generale del sistema e del GC modificando i consumi totali.

Hyp 4 : il sistema operativo ha impatto sui consumi energetici

L'ultima ipotesi riguarda le differenze architetturali dei sistemi operativi: si ipotizza che un sistema operativo a 64 bit comporti, a parità di memoria, un consumo energetico maggiore rispetto ad uno a 32 bit. Questo perché la presenza di puntatori di maggiori dimensioni, obbligano il GC ad analizzare una dimensione di memoria molto più estesa che comporta garbage collection più frequenti. Difatti, in Java, l'overhead di memoria dovuto all'utilizzo di un'architettura a 64 bit viene ritenuto essere del 30% circa (paragrafo 4.3.1).

Hyp 5 : un'architettura software a 64 bit comporta, a parità di memoria, consumi più elevati rispetto a quelli di una a 32 bit

Capitolo 5

Metodologia di Lavoro

In questo capitolo viene descritto in dettaglio l'ambiente in cui sono stati svolti i test. Inizialmente vengono illustrate le applicazioni testate e in seguito tutti gli elementi hardware che compongono il sistema. Dopo la descrizione fisica dei componenti vengono approfonditi i dettagli relativi alle applicazioni sviluppate per produrre i carichi di lavoro, per acquisire i dati e per poi analizzarne i risultati.

5.1 Ambiente e applicazioni di test

5.1.1 Applicazioni di test

Per comprendere al meglio il funzionamento del GC in contesti pratici sono state testate alcune applicazioni Java e C su due sistemi operativi differenti.
I sistemi operativi presi in considerazione per i test sono stati Windows Server 2003 e Fedora 13 64bit. Il sistema operativo linux preso in considerazione inizialmente era CentOS, che meglio rappresenta un sistema operativo server ma, per problemi di compatibilità con la versione di Openbravo testata, è stato necessario adottare Fedora.
La prima applicazione di testing, chiamata *GCTest*, ha lo scopo di mostrare coi suoi risultati come il GC lavora differentemente con diversi quantitativi di memoria massima. Il software è stato scritto sia in Java che in C in modo da poter operare un confronto tra i due linguaggi. La versione C è stata creata in altrettante

due versioni: una senza GC, che utilizza i classici meccanismi di allocazione e deallocazione, e l'altra con il Boehm-Demers-Weiser GC. GCTest alloca e dealloca in modo ciclico una lista di oggetti (o variabili in C) delle dimensioni specificate. Le altre applicazioni testate sono due sistemi ERP che utilizzano come web server Tomcat e come database PostgreSQL: Openbravo e Adempiere. I test su queste applicazioni permettono di creare un carico reale su cui osservare il funzionamento del GC. Nell'Appendice A viene descritta in dettaglio la procedure di installazione e configurazione per entrambi i sistemi operativi dei due ERP analizzati.

5.1.2 Ambiente di test

Durante i test sono stati utilizzati due calcolatori collegati tramite un router, uno per ricoprire il ruolo di server mentre l'altro di client.
Per l'acquisizione dei dati sono state utilizzate due schede di misurazione e una scheda d'acquisizione dati. Le prime schede vengono utilizzate per trasformare i segnali di corrente in segnali analogici tra 0 e 5 *Volt*, mentre l'ultima converte i segnali analogici in digitale comunicandoli ad un pc tramite cavo *USB*.

Al server è collegata la scheda di misurazione ed è un personal computer di tipo desktop con le seguenti caratteristiche:

- processore Intel Core2Duo E6400 a 2,13GHz
- scheda madre Asus P5B
- 2GB di ram DDR2 PC2-5300 in DualChannel
- scheda video ATI HD2600XT
- hard disk 120GB 7200rpm interfaccia IDE UltraATA
- alimentatore Enermax 620W

Ad esso sono collegate tramite appositi cavi le schede per la misurazione dei valori di corrente.

Il pc client è un laptop Sony Vaio SR21M con un processore Intel Centrino2 P8400, 4GB di ram DDR2 e un hard disk 5400rpm da 250GB. Ad esso è collegata tramite cavo *USB* la scheda di acquisizione che fa da tramite con le schede di misurazione.

Le schede di misurazione hanno lo scopo di consentire la misurazione del consumo totale e dei consumi dei componenti interni al personal computer. La prima scheda viene inserita in serie tra la presa di corrente della rete elettrica e la spina del cavo d'alimentazione del server (Figura 5.1). La seconda invece possiede diversi connettori per essere inserita, sempre in serie, tra i cavi di alimentazione dei componenti interni(Figura 5.2).

La prima scheda è equipaggiata con un trasduttore corrente-tensione (modello *Lem LTS 25-NP*) mentre la seconda con otto trasduttori. I trasduttori permettono di misurare il flusso di corrente che li attraversa e restituire un valore di tensione tra 0 e 5 Volt che lo rappresenta. Questo valore viene letto dalla scheda di acquisizione per poi calcolare il consumo istantaneo del sistema o del singolo componente relativo.

Il trasduttore che equipaggia la prima scheda viene utilizzato per leggere il flusso di corrente che scorre nell'alimentatore per poter calcolare il consumo totale del sistema. I trasduttori della seconda scheda leggono i valori dei diversi cavi di alimentazione collegati a processore, scheda madre e hard disk. Più in particolare i segnali misurati appartengono ai cavi:

- ATX +12V
- ATX -12V
- ATX 3.3V

(a) Interno della scheda

(b) Esterno della scheda

(c) Pins di collegamento della scheda

Figura 5.1: *Scheda di misurazione per il calcolo del consumo totale del sistema*

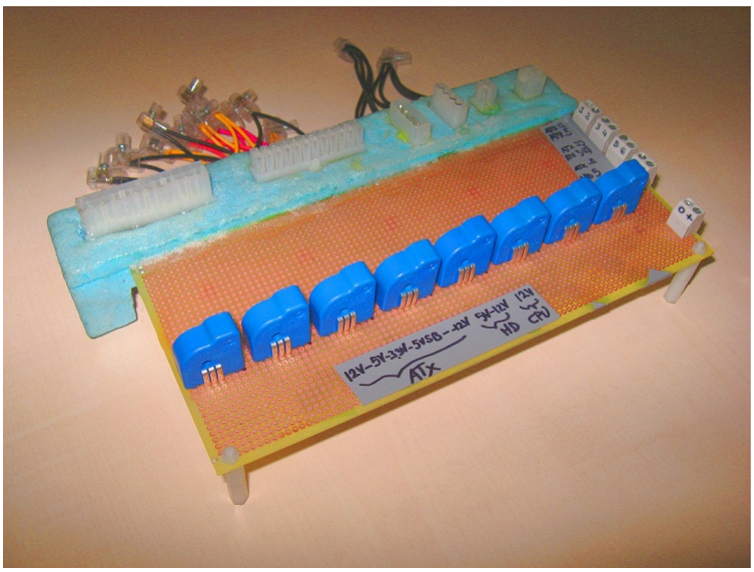


Figura 5.2: *Scheda di misurazione per il calcolo del consumo dei componenti interni*

- ATX 5V
- ATX 5V standby
- Hard Disk 5V
- Hard Disk 12V
- CPU 12V

La scheda d'acquisizione dati è il modello NI USB-6210 della National Instruments (16 ingressi, 16-bit, 250kS/s) la quale è stata collegata al laptop tramite cavo *USB*. Ad essa devono essere collegati i fili provenienti dalle due schede di misurazione.
I collegamenti tra schede di misurazione e scheda d'acquisizione sono descritti con precisione nell'Appendice B. Ulteriori dettagli sulla costruzione e sulle considerazioni tecniche relative all'intero sistema d'acquisizione possono essere trovati nella tesi dell'Ing. Gallazzi e dell'Ing. Formenti [10].

Nella Figura 5.3 viene illustrato lo schema dei collegamenti tra i vari componenti del sistema di test.

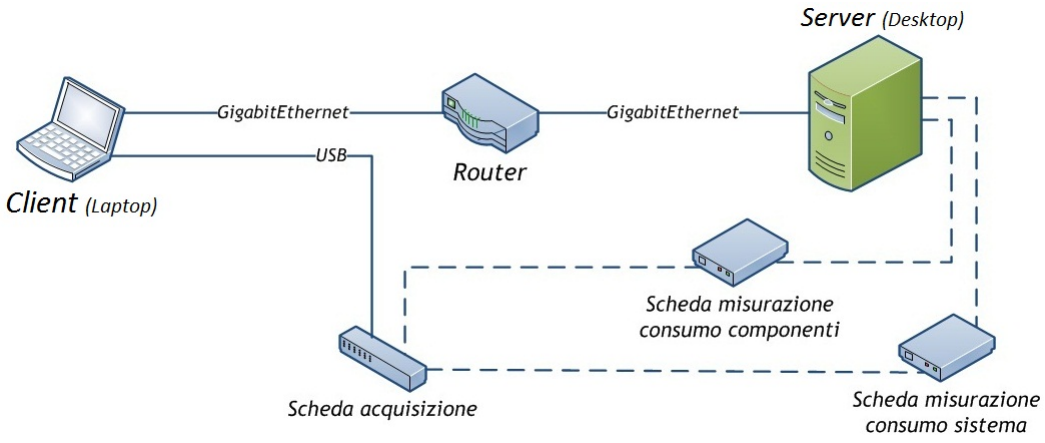


Figura 5.3: *Diagramma delle connessioni tra i componenti del sistema di test*

5.2 Applicazione per il testing e l'acquisizione dei dati energetici

Nel paragrafo precedente sono stati descritti l'ambiente di test e le applicazioni che sono state analizzate. Ora è necessario definire come questi software sono stati testati e come l'intero sistema di test si coordina. Le applicazioni che hanno permesso lo svolgimento dei test sono due:

- *Workload Manager*: è un software java di tipo client/server che permette di impostare il carico di lavoro a cui sottoporre la macchina di test.
- *Virtual Instruments*: è un programma LabVIEW che permette di acquisire i dati sui consumi energetici realtivi a un determinato test.

5.2.1 Workload Manager

L'architettura di questa applicazione è la stessa di quella utilizzata da Gallazzi e da Formenti [10] ma è stata riscritta per adattarsi alle nuove esigenze di test.
Il pacchetto di questa applicazione è suddiviso in due applicazioni: una che svolge il compito di server e una invece quello di client. La parte server possiede un interfaccia grafica tramite la quale è possibile scegliere la tipologia di test e le sue caratteristiche; la parte client ha il compito di eseguire il test scelto con le caratteristiche specificate. L'applicazione server può comandare più client contemporaneamente in modo da produrre un workload maggiore per cercare di porre la macchina di test sotto un carico di lavoro più alto.

I test che sono stati svolti riguardano due ERP e un programma di benchmark chiamato GCTest; per ognuna delle due tipologie è presente una scheda, nell'interfaccia grafica, che permette all'utente di settare le caratteristiche del workload. In Figura 5.4 e in Figura 5.5 si può vedere come sono composte le due schede, la

prima per gestire i GCTest mentre la seconda gli ERP.

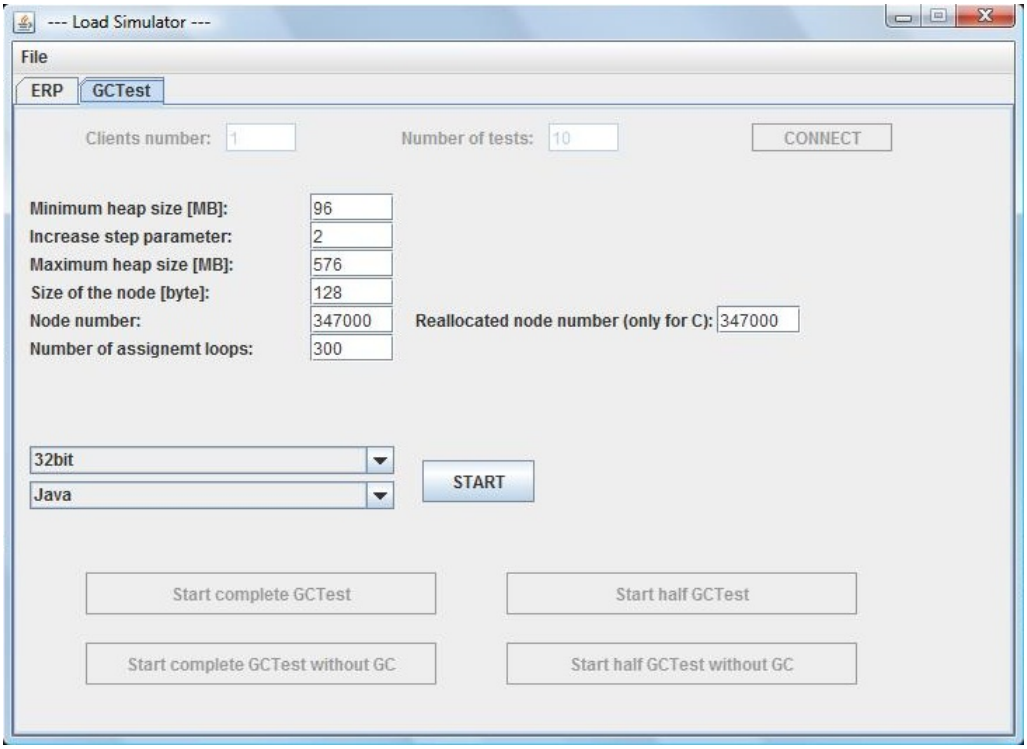


Figura 5.4: *Scheda dell'interfaccia grafica per il settaggio dei test con GCTest*

Workload Manager: ERP

Prima di eseguire l'applicazione server è necessario che siano in esecuzione una o più applicazioni client su una sola macchina o su diverse macchine. Le applicazioni client come già detto sono di tipo Java ma, oltre alla JVM, necessitano anche di un'applicazione che permetta loro di simulare le richieste via browser all'ERP scelto.

L'applicazione utilizzata, *iMacros*, permette di specificare utilizzando un linguaggio di scripting le operazioni da effettuare in una pagina web. Il programma Java, per comunicare con questa applicazione, utilizza una DLL chiamata *jawin.dll*. Essendo sia il programma che la DLL compatibili solo Windows è necessario che le macchine client abbiano installato un sistema operativo di questo tipo.

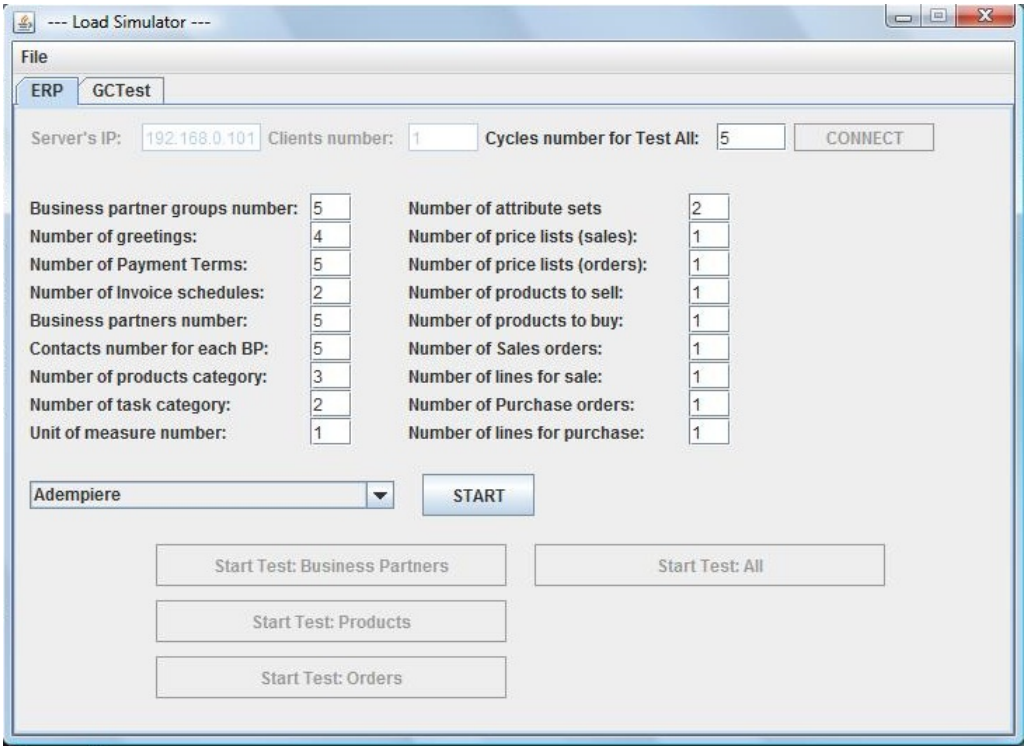


Figura 5.5: *Scheda dell'interfaccia grafica per il settaggio dei test sugli ERP*

Il client in base ai parametri scelti per il test dell'ERP crea una lista di comandi che poi invierà ad iMacros il quale li eseguirà; al termine della procedura iMacros comunicherà al client collegato che può procedere e il client a sua volta comunicherà al server di aver terminato con il test assegnato. Una volta che il server ha ricevuto il segnale di terminazione del test da tutti i client collegati procede, se impostato, con il ciclo di test successivo; altrimenti ne comunica la fine con un popup e riporta l'interfaccia grafica a disposizione dell'utente per nuovi test.
Per lanciare l'applicazione client si deve digitare il seguente comando:

```
java -jar client.jar ip_server os
```

Al posto di *os* si deve scrivere nel caso degli ERP la lettera *W* per indicare al programma che viene eseguito su una macchina con sistema operativo Microsoft. Sarà chiara più avanti l'utilità di quest'ultimo parametro. Una volta lanciato, il

client, tenterà ciclicamente una connessione via socket al server fino a quando non lo troverà disponibile ad accettare connessioni.

L'applicazione server, una volta aperta con il semplice comando *java -jar serverGUI.jar*, visualizzerà le due schede mostrate nelle Figure 5.5 e 5.4 che saranno selezionabili cliccando sul relativo segnalibro. Nella scheda riguardante i test sugli ERP è possibile impostare i seguenti parametri:

- *Server's IP* : imposta l'IP dell'application server su cui è presente l'ERP e a cui devono essere inviate le richieste (IP della macchina da testare)
- *Clients number* : imposta il numero di client da cui accettare connessioni e a cui inviare le impostazioni del test scelto
- *Cycles number for Test All* : imposta il numero di cicli di test da eseguire nel caso venga selezionata la modalità *Test All*.
- *Items Number Area* : comprende tutta l'area dove è possibile selezionare il numero di elementi che devono essere creati durante il test. Per ogni tipologia di oggetto è possibile specificarne il numero (*es. quattro business partners, due orders price lists, etc.*).
- *ERP Menu* : nel menu a tendina è possibile scegliere se l'ERP da testare è Adempiere od Openbravo.
- *Pulsante Start* : sblocca i tasti per lanciare i vari tipi di test con le impostazioni specificate in precedenza.
- *Pulsanti Start Test* : lanciano rispettivamente il test che esegue la parte relativa alla creazione dei business partners, dei prodotti, degli ordini oppure di tutte e tre le parti assieme.

Una volta che tutti i client hanno segnalato la terminazione del test viene visualizzato un popup di comunicazione. Alla pressione del tasto di conferma il popup

sparisce e il programma ritorna alla sitauzione precedente alla pressione del tasto *Start Test* selezionato.

Workload Manager: GCTest

Al contrario degli ERP, nel caso dei GCTest, il client deve essere lanciato direttamente sulla macchina che andrà testata, in modo che possa richiamare il programma GCTest corretto in base alla versione scelta. Dato che i percorsi e la sintassi del comando per eseguire un file sono differenti tra Linux e Windows, deve essere comunicato al client su che sistema operativo sta per essere eseguito. Per fare ciò basta digitare la lettera *W* o *L* al posto della parola *os* nel seguente comando:

```
java -jar client.jar ip_server os
```

In totale sono state scritte 4 versioni dei GCTest:

- *GCTest in Java*: come già spiegato crea una lista del numero di elementi scelto, la distrugge interamente e ripete queste operazioni per il numero di cicli specificato.
- *GCHalfTest in Java*: come il caso precedente ma la lista non viene distrutta completamente ma viene invece reallocata per metà, lasciando l'altra metà invariata dall'inizio del test alla fine.
- *GCTest in C senza GC*: esegue le stesse operazioni della versione Java ma, oltre a deallocare staticamente le variabili come succede normalmente in C, accetta in ingresso un parametro aggiuntivo per specificare la quantità di nodi da reallocare.
- *GCTest in C con GC*: identica all'applicazione precedente tranne per il fatto che si affida al Boehm-Demers-Weiser GC per la deallocazione delle variabili.

Come si può osservare dalla Figura 5.4 è possibile impostare alcuni parametri per il test:

- *Clients Number*: indica il numero di client da cui accettare connessioni e che eseguiranno il GCTest scelto.
- *Number of Tests*: indica il numero di cicli completi di test che saranno eseguiti.
- *Minimum e Maximum Heap Size*: indicano l'intervallo di quantità di memoria massima che verrà associata a ogni GCTest in Java.
- *Increase step parameter*: indica di quanto cresce rispetto alla *Minimum Heap Size* la memoria massima associata al GCTest ogni volta che viene eseguito. L'incremento di memoria massima ad ogni test si ottiene dividendo la *Minimum Heap Size* con l'*Increase step parameter*. Per esempio con la *Minimum Heap Size* impostata a 100MB, la *Maximum* a 300MB e l'*Increase step parameter* a 2 il GCTest, per ogni ciclo completo, sarà lanciato con una memoria massima di: 100, 150, 200, 250 e 300 MB.
- *Size of the Node*: specifica le dimensioni di ogni nodo della lista, più avanti verrà illustrato in dettaglio come ciò avviene all'interno del GCTest.
- *Node number*: indica la lunghezza in nodi della lista.
- *Reallocated Node number*: questo parametro è valido solo per le versioni C dei GCTest e indica il numero di nodi che saranno reallocati.
- *Number of Assignment Loops*: indica il numero di volte che la lista deve essere ricostruita in ogni singolo test.

Dopo aver impostato questi parametri è possibile scegliere se il sistema operativo sul quale si sta eseguendo il test possiede un architettura a 32 o 64 bit. Questo

parametro viene tenuto in considerazione per calcolare più precisamente le dimensioni di un nodo della lista.
Inoltre si può scegliere tra le versioni C o le versioni Java del GCTest.
Infine, per lanciare la sequenza di test coi parametri specificati finora, sono presenti quattro pulsanti che funzionano in modo differente in base al linguaggio di programmazione scelto:

- *Start Complete GCTest*: sia nel caso di Java che C viene lanciata la versione che dealloca ogni volta l'intera lista. La versione di GCTest in C è quella che utilizza il GC.

- *Start half GCTest*: nel caso di Java viene lanciato l'applicazione GCHalfTest mentre nel caso di C viene lanciato il realativo GCTest passandogli come parametro il numero di nodi da reallocare. La versione di GCTest in C è quella che utilizza il GC.

- *Start Complete GCTest without GC*: questo pulsante è attivo solo se il linguaggio selezionato è il C. Esso lancia la versione che dealloca ogni volta l'intera lista. In questo caso l'allocazione e la deallocazione della memoria avviene nel modo classico.

- *Start half GCTest without GC*: anche questo pulsante è attivo solo se il linguaggio selezionato è il C. Esso lancia la versione che realloca il numero di nodi specificato. L'allocazione e la deallocazione della memoria avviene nel modo classico.

Una volta terminati i cicli completi di test viene segnalata la loro fine con un popup come nel caso dei test sugli ERP.

5.2.2 Virtual Instruments

Per analizzare il consumo della macchina su cui viene testato l'applicativo vengono raccolti i diversi dati misurati dalle due schede collegate al server. Durante

l'esecuzione vengono raccolti i valori istantanei della potenza assorbita ad una frequenza minima di 100Hz. Il valore di frequenza deriva dal fatto che, secondo il teorema del campionamento di Nyquist Shannon, un segnale *x(t)* che non contiene frequenze superiori a B è completamente descrivibile da una serie di valori spaziati nel tempo di $\frac{1}{2B}$. Essendo la frequenza massima del segnale proveniente dalla rete elettrica 50Hz la frequenza di campionamento necessaria è di almeno 100Hz.
Nello strumento virtuale è possibile settare il consumo in idle per i vari componenti in modo da poter calcolare successivamente la potenza aggiuntiva necessaria ad eseguire l'applicazione.
Al termine del test i dati campionati vengono analizzati per calcolare i seguenti valori:

- *valore medio di potenza*: calcolato effettuando la media aritmetica dei vari valori di potenza acquisiti
- *integrale del consumo*: indica la potenza consumata durante l'esecuzione del test ed è calcolato tramite la formula

  $$PotenzaMediaAssorbita * DurataTest$$

  dove la durata del test è espressa in secondi.
- *consumo in Wh*: viene calcolato come

  $$\frac{PotenzaMediaAssorbita * DurataTest}{3600}$$

  dove la durata del test è espressa in secondi.
- *valore medio differenziale di potenza*: è calcolato come la potenza media assorbita al netto della potenza assorbita in idle.
- *integrale differenziale del consumo*: è calcolato come l'integrale dei valori istantanei ma sottratti della potenza media assorbita in idle.

- *consumo differenziale in Wh*: come nel caso non differenziale ma utilizzando come potenza media quella differenziale.

Tutti i dati istantanei, la durata del test e i valori calcolati vengono salvati in diversi file ognuno corrispondente al sistema complessivo e ai diversi componenti.

Tutte queste operazioni vengono svolte in modo automatizzato dal Virtual In-

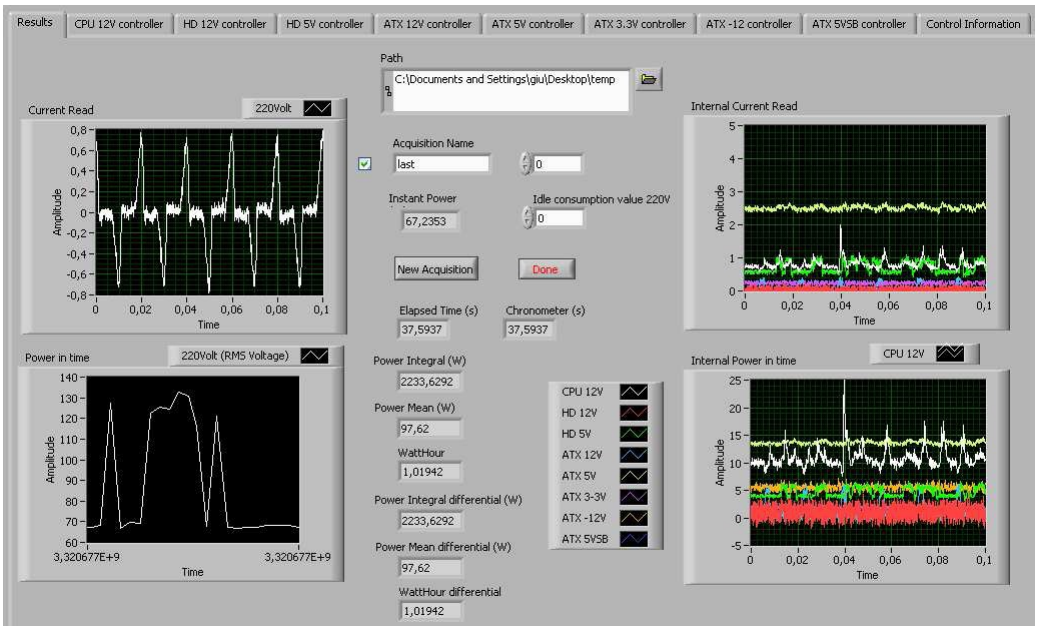


Figura 5.6: *Scheda principale dell'interfaccia grafica del virtual instrument*

strument di cui uno screenshot della relativa interfaccia grafica è visibile nella Figura 5.6.

L'inizio e la fine dell'acquisizione può essere comandata manualmente dall'utente utilizzando i vari pulsanti dell'interfaccia oppure, in modo automatizzato, segnalando i comandi di inizio e fine all'interno di alcuni file speciali. Questo secondo metodo viene descritto in modo più approfondito nel paragrafo successivo.

5.2.3 Comunicazione tra WM Server, WM Client e Virtual Instrument

Le comunicazioni che avvengono tra Workload Manager parte server e parte client sono già state illustrate nel paragrafo precedente mentre, la gestione automatizzata

del virtual instrument, verrà descritta con maggiore dettaglio ora.
Il Virtual Instrument controlla continuamente determinati file per capire quando può dare inizio all'acquisizione dei dati e quando invece la deve fermare. Inoltre scrivendo nell'apposito file è possibile impostare anche il nome del test attuale che andrà a dare il nome alla cartella contenente i file dei risultati.
Lo strumento virtuale è stato modificato per controllare la cartella *c:/log/* ed analizzare il contenuto di due file:

- *testStart.txt*: dove devono essere scritti i comandi *start* per far iniziare l'acquisizione e *stop* per interromperla.
- *testName.txt*: dove deve essere scritto il nome del test.

In questo modo il WM server può gestire in modo automatizzato il Virtual Instrument semplicemente modificando i due file nel modo appena descritto.
Con questa configurazione del sistema di test e di acquisizione dei dati è possibile suddividere i compiti ognuno su una macchina diversa a patto che soddisfi determinati requisiti:

- *Workload Manager server*: oltre a dover possedere una JVM non deve avere particolari requisiti.
- *Workload Manager client*: la macchina deve avere installato una JVM, iMacros e la DLL *jawin.dll*.
- *Virtual Instrument*: il computer deve aver installato LabVIEW e condividere i file per la gestione dello strumento virtuale con il WM server.
- *Server*: deve aver installato i sistemi operativi e i software descritti nell'Appendice A.

Ovviamente il requisito comune a tutte le macchine è che siano in connessione fra di loro.

Nella Figura 5.7 si può osservare lo schema delle macchine reali o virtuali che partecipano all'esecuzione e al monitoraggio di un test.

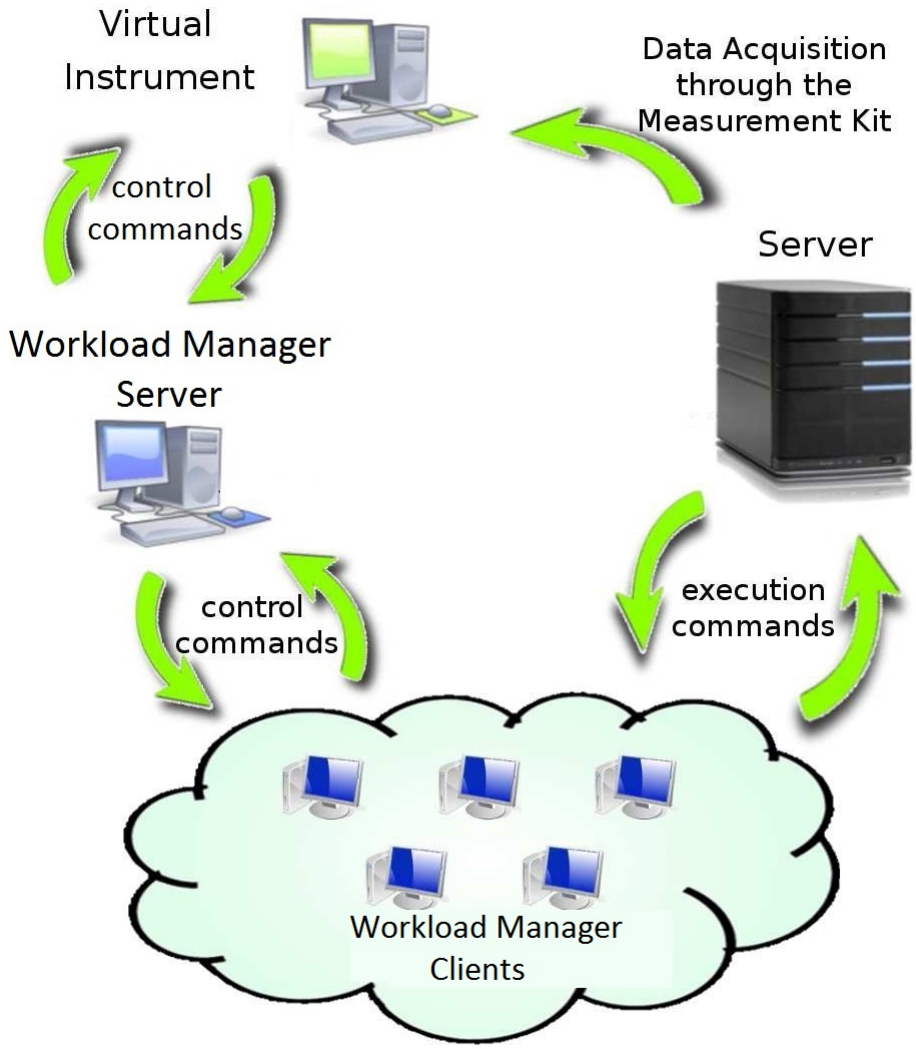


Figura 5.7: *Diagramma generale del sistema di test*

Come si può osservare dalla Figura 5.3, nel paragrafo riguardante l'ambiente di test, il sistema è stato configurato in modo da eseguire su una sola macchina (il laptop) i compiti di WM server, di WM client e di Virtual Instrument nel caso di test sugli ERP. Nel caso di GCTest, invece, i WM client andranno eseguiti direttamente sulla macchina da testare (desktop).

5.3 Applicazione per l'analisi dell'output del GC

Nel capitolo precedente è stato illustrato il modo in cui si possono ricavare informazioni relative al funzionamento del GC. D'altra parte i dati che si ricavano sono di difficile comprensione nella forma in cui vengono mostrati. Per avere una visione più immediata delle differenze di funzionamento del GC in diverse configurazioni di memoria, è stato scritto un programma che analizza l'intero file di log generato durante la collection e restituisce dei valori rappresentanti:

- *GC Total Time*: tempo totale delle pause dovute al GC. L'unità di misura per il tempo è il *secondo*.
- *GC Time (young)*: tempo totale delle pause dovute alle collection sulla parte young di memoria.
- *Full GC Time*: tempo totale delle pause dovute alle full garbage collection.
- *Total Released Memory*: memoria totale deallocata dal GC. I valori relativi alla memoria sono in *MegaBytes*.
- *GC (young) Released Memory*: memoria totale deallocata dal GC durante delle young collection.
- *Full GC Released Memory*: memoria totale deallocata tramite full garbage collection.
- *Medium Available Heap*: memoria mediamente libera durante l'esecuzione.
- *GC Total Number*: numero totale di garbage collection eseguite.
- *GC (young) Number*: numero totale di garbage collection sull'area young eseguite.
- *Full GC Number*: numero totale di full garbage collection eseguite.

L'applicativo è impostato per analizzare i file di log semplici (non dettagliati), ma è possibile modificarlo velocemente per renderlo compatibile anche con i file restituiti col comando *-XX:+PrintGCDetails*.
Lo scopo principale di questa applicazione non è tanto comprendere da dove provengono i problemi di funzionamento del GC, anche se è possibile trarre alcune conclusioni dai risultati, ma comprendere le differenze che si vengono a creare con setting differenti della memoria.
In questo modo è possibile effettuare una scelta migliore delle impostazioni di memoria, dato che è possibile osservare valori aggregati che tengono in considerazione un ampio tempo di esecuzione dell'applicazione testata. La granularità offerta dall'output del GC, invece, permette di comprendere le problematiche relative alle singole esecuzioni del GC, ma non riesce a dare una visione d'insieme sul comportamento durante l'intera vita dell'applicazione. Soluzioni che permettono di migliorare singoli comportamenti non è detto che portino ad un miglioramento del funzionamento globale.
Nei capitoli successivi sarà possibile comprendere meglio come questa analisi aggregata può essere utilizzata nei confronti tra diversi setting di memoria.

Capitolo 6

Risultati Operativi

In questo capitolo sono descritti più in dettaglio i test effettuati per indagare sulla veridicità o meno delle ipotesi di ricerca. Per ogni test vengono analizzati i risultati ottenuti ed effettuate alcune osservazioni. Infine vengono riassunte le ipotesi di ricerca e per ognuna di esse viene descritta e commentata la relativa validità.

6.1 Test su allocazione e deallocazione variabili

La prima serie di test è stata svolta sull'applicativo *GCTest* in tutte le versioni create, per comprendere il comportamento del GC al variare delle dimensioni di memoria e del numero di oggetti long-lived.
Il benchmark *GCTest* crea una lista contenente il numero di nodi specificato e delle dimensioni desiderate. Dopo aver creato la lista viene distrutto il numero di nodi passato come parametro per poi ricrearli. Anche il numero di cicli di creazione e distruzione può essere impostato prima del lancio del test.

Come già detto i sistemi operativi analizzati possiedono due architetture diverse: Windows Server 2003 adotta un'architettura a 32bit mentre Fedora 13 una a 64bit. Utilizzare un sistema operativo a 64 bit comporta un overhead di memoria dovuto alla maggior dimensione dei puntatori. Questo overhead si potrebbe

ripercuotere sulle prestazioni delle applicazioni e del GC. Il vantaggio di avere un sistema operativo a 64bit risiede nel fatto che è possibile gestire quantitativi di memoria ram molto maggiori (superiori a 3 GB, fino a 128 GB). Per l'ambiente di test analizzato, che possiede 2 GB di ram, non c'è nessun vantaggio a utilizzare un sistema operativo a 64bit. Si è scelto di effettuare test su architetture differenti per avere risultati che permettano di confrontare sistemi molto diversi fra di loro per capire quanto ciò influisca sui consumi energetici.

Per creare i nodi delle dimensioni desiderate il programma richiede in input, oltre alle dimensioni desiderate per ogni nodo, il tipo di architettura del sistema operativo (32 o 64 bit). In questo modo, tenendo conto delle dimensioni dei puntatori che indirizzano i vari elementi, può calcolare l'effettiva area di memoria da allocare per ciascuno di essi.

In Java ogni nodo è definito come una classe contenente un puntatore a un oggetto dello stesso tipo e un array di *byte*. Il puntatore viene utilizzato per collegare i vari elementi della lista mentre l'array per decidere il numero di byte da occupare per ottenere delle dimensioni in memoria pari a quelle scelte.
In Java generalmente un oggetto possiede due puntatori mentre un array utilizza delle strutture dati che comportano un overhead di memoria pari a quello di tre puntatori.
La dimensione del nodo è calcolata con la formula seguente:

$$DimTotale = OvhOggetto + OvhPuntatore + OvhArray + NumByteArray$$

$$DimTotale = 2 * ref + ref + 3 * ref + NumByteArray$$

$$32bit : ref = 4byte; 64bit : ref = 8byte$$

La sigla *ref* sta ad indicare le dimensioni dei puntatori.
Il programma calcola tramite formula inversa il numero di elementi dell'array di

byte:

$$NumByteArray = DimTotale – 6 * ref$$

In questo modo è possibile posizionare le due JVM sullo stesso piano di lavoro per osservare le differenze a livello di sistema operativo.

In C ogni nodo è definito tramite una struttura che contiene un puntatore a una struttura dello stesso tipo e un puntatore a *unsigned char* che serve per indirizzare l'area di memoria che andrà allocata. Per calcolare le dimensioni dell'area di memoria puntata viene utilizzata una formula simile alla precedente, dove però il numero dei riferimenti totali scende da sei a due: uno derivante dal puntatore alla struttura e uno derivante dal puntatore ad *unsigned char*. Quindi la formula inversa utilizzata nella versione C è:

$$NumByteArray = DimTotale – 2 * ref$$

Specificando per entrambi i sistemi operativi la stessa architettura, è possibile ottenere un confronto su un carico egualitario in modo da comprendere a quanto ammonta l'overhead in termini di consumi derivante dalla maggiore dimensione dei puntatori a 64bit.

I test eseguiti in Java sono stati effettuati al variare dei seguenti parametri:

- tipologia di architettura
- tipologia di JVM (client o server)
- numero di nodi reallocati (tutti o solo la metà)
- quantità massima di memoria

Per quanto riguarda i test in C sono stai eseguiti con diversi quantitativi di nodi reallocati e con entrambe le versioni del *GCTest*: con GC e senza.

In tutti i casi i test sono configurati con gli stessi parametri (tranne che per il tipo di architettura e il numero di nodi reallocati) e valori scelti sono:

- *lunghezza lista = 347000 nodi*
- *dimensione nodo = 128 byte*
- *cicli di creazione-distruzione = 300*

I parametri sono stati scelti per garantire un tempo d'esecuzione sufficientemente lungo per poter considerare trascurabili le eventuali variazioni casuali di carico derivanti dal sistema operativo.

Il numero di test eseguiti per ogni impostazione dei parametri è pari a cinque. Il numero di ripetizione scelto ha permesso di ottenere in tutti i casi un intervallo di confidenza per la media, al 99%, con ampiezza inferiore al 10% della media aritmetica.
Per poter calcolare l'intervallo di confidenza sono state calcolate prima la media e la varianza campionaria con le seguenti formule:

$$MediaCamp = \bar{X} = \frac{\sum_{i=1}^{n} X_i}{n}$$

$$VarianzaCamp = S^2 = \frac{\sum_{i=1}^{n} (X_i - \bar{X})^2}{n-1}$$

L'intervallo di confidenza per la media è stato calcolato tramite la formula che sfrutta la distribuzione di *Student* dato che media e varianza non sono note. Il risultato si ottiene dalla formula:

$$IC\ al\ 99\% :\ \bar{X} \pm\ t_{n-1}\left(\frac{1+0.99}{2}\right)\frac{\sqrt{S^2}}{\sqrt{n}}$$

$$t_{n-1}\left(\frac{1+0.99}{2}\right) = t_4(0.995) = 4.604$$

I valori dei test indicheranno sempre la media aritmetica dei singoli risultati.

6.1.1 GCTest Java: JVM Server e Client

Prima di proseguire con la descrizione del test è bene comprendere le differenze tra le due tipologie di JVM prese in considerazione, sia in termini di configurazione della memoria interna sia in termini di casi di utilizzo.
La versione client della JVM è più adatta per contesti in cui gli oggetti short-lived sono molto maggiori di quelli long-lived. Un esempio di questa tipologia di contesto è un'interfaccia grafica. Gli oggetti con vita breve verranno subito eliminati e perciò non sarà necessario passarli nella parte di heap contenente gli oggetti con durata della vita maggiore. In questa versione di JVM il *NewRatio*, che è il parametro che specifica la quantità di heap a disposizione degli oggetti maturi, è pari a 8/9. In questo modo si ottiene un quantitativo di memoria per gli oggetti young basso mentre per quelli old grande.
Strutturando così la memoria sarà difficile che la parte contenente le variabili old venga riempita del tutto, copiando gli oggetti young sopravvissuti, e rendendo inevitabile una GC dell'intera memoria (*Full GC*). In questo modo si evitano pause lunghe dovute all'analisi dell'intera memoria.
Di contro se questa versione viene utilizzata in contesti a cui non è adatta (tanti oggetti long-lived) si incorrerà in un overhead alto dovuto al frequente spostamento degli oggetti sopravvissuti verso la parte tenuered della heap.

La versione server punta tramite apposite tecniche ad avere una velocità di esecuzione e di picco alta, piuttosto che al veloce start-up dell'applicazione e alla bassa occupazione di memoria come avviene nella versione client.
La versione server ha un *NewRatio* pari a 2/3. In questo modo la quantità di memoria per gli oggetti short-lived risulta essere maggiore, diminuendo l'overhead dovuto allo spostamento degli oggetti sopravvissuti e avvantaggiando così contesti con molti oggetti long-lived.
Ovviamente se utilizzata con molti oggetti short lived sarà presente un overhead

dovuto al maggior carico di lavoro a cui è sottoposto il GC per analizzare un'area di memoria più grande (short). Nel caso la memoria disponibile sia poca il GC sarà comunque obbligato ad effettuare delle *Full GC* più frequenti creando pause maggiori rispetto alla versione client; questo deriva dal fatto che l'area di memoria young è più grande e quindi è in grado di riempire più velocemente l'area tenuered.

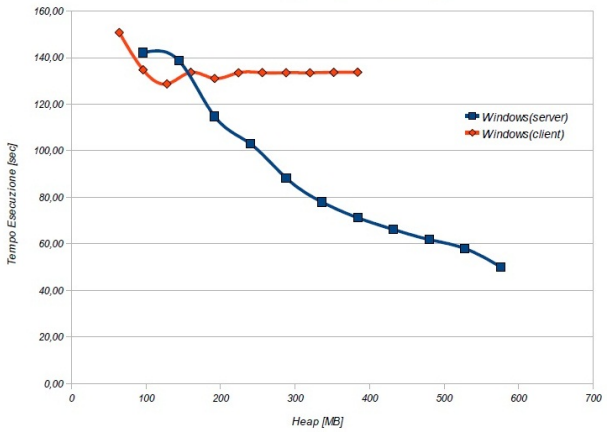
La JVM è disponibile in entrambe le versioni solo per sistemi operativi a 32 bit, quindi, per questo caso di test, verrà preso in considerazione solo Windows Server 2003.

Oltre ai test di misura dei consumi sono stati svolti anche dei test per comprendere lo stato di funzionamento del GC tramite l'applicazione descritta nel paragrafo precedente.

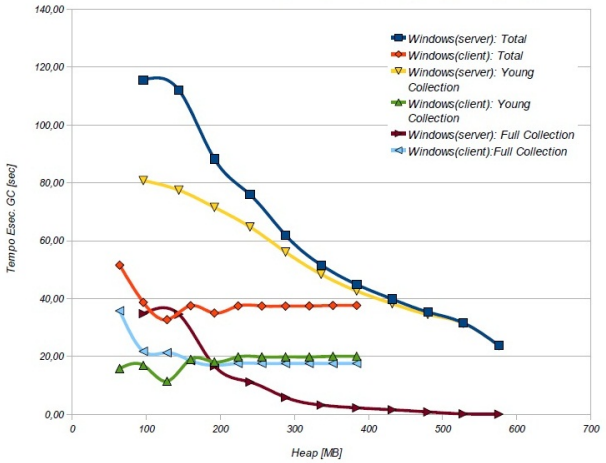
Analisi comportamento GC

Nel test per verificare il funzionamento del GC è stata impostata la JVM per salvare in un file di log i dettagli relativi alle garbage collection eseguite. Una volta terminati i test sono stati analizzati i risultati tramite l'applicazione creata. La memoria massima impostata durante i test è stata fatta variare da un valore pari al minimo possibile per terminare l'esecuzione, a un valore massimo pari a sei volte il valore minimo. Per la JVM versione client il valore minimo è risultato essere 64MB mentre per la versione server 96MB. Questa differenza è dovuta al fatto che la JVM Server utilizza un'impostazione della memoria che gli impedisce di effettuare una garbage collection tale da liberare sufficiente memoria durante il primo ciclo di allocazione della lista.
I valori ricavati dai test, di distruzione completa della lista (*GCTest*: Figura 6.1) e di distruzione di metà lista (*GCHalfTest*: Figura 6.2), al variare del tipo di Java Virtual Machine e della dimensione della memoria massima sono:

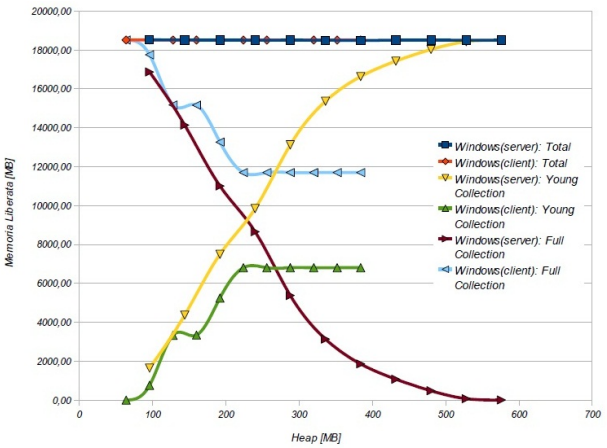
- tempo d'esecuzione



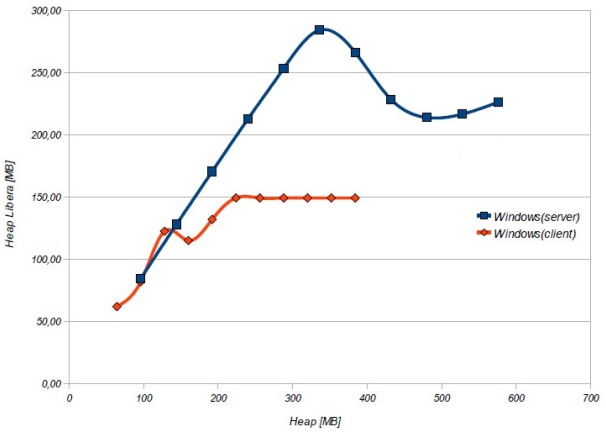
(a) Tempo d'esecuzione al variare della memoria massima



(b) Tempo impiegato dal GC al variare della memoria massima

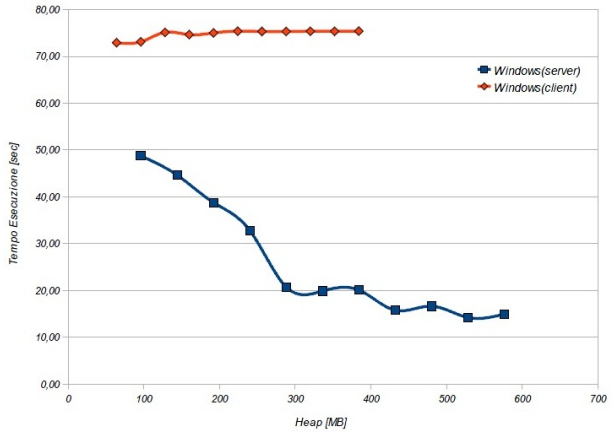


(c) Memoria liberata dal GC al variare della memoria massima

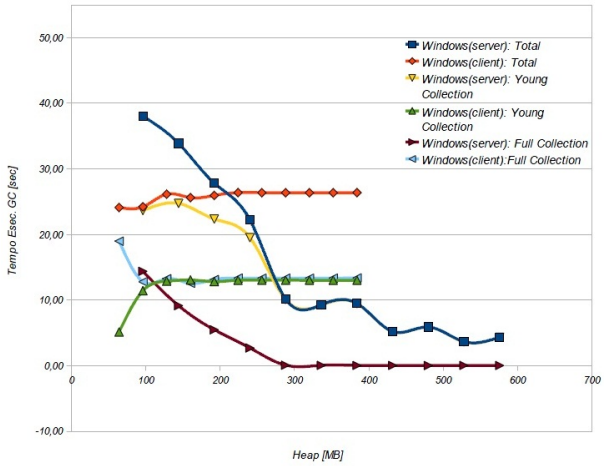


(d) Quantitativo medio di memoria libera al variare della memoria massima

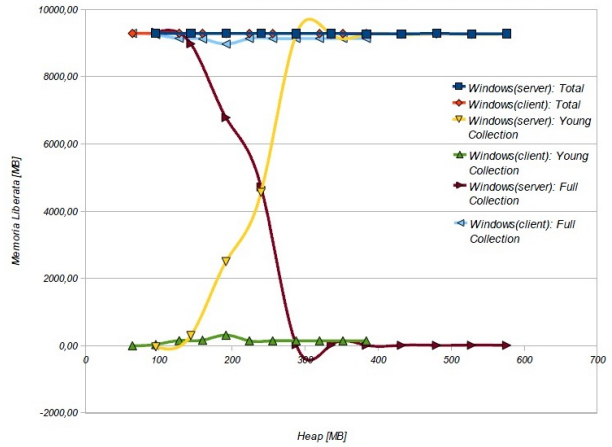
Figura 6.1: *GCTest: JVM Client e Server*



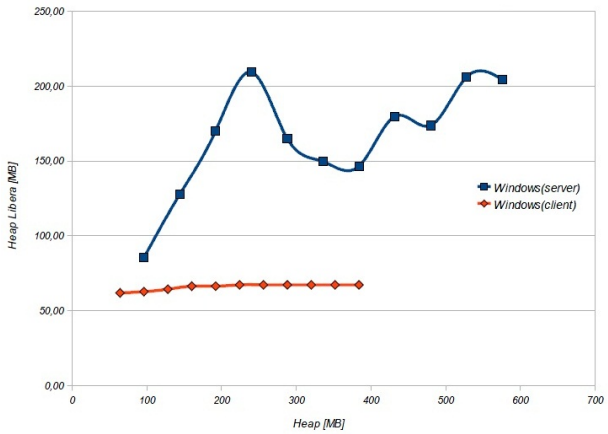
(a) Tempo d'esecuzione al variare della memoria massima



(b) Tempo impiegato dal GC al variare della memoria massima



(c) Memoria liberata dal GC al variare della memoria massima



(d) Quantitativo medio di memoria libera al variare della memoria massima

Figura 6.2: *GCHalfTest: JVM Client e Server*

- tempo totale per le garbage collection
- tempo impiegato per le garbage collection sull'area young
- tempo impiegato per le full garbage collection
- quantità di memoria liberata dal GC
- quantità di memoria liberata con le garbage collection sull'area young
- quantità di memoria liberata con le full garbage collection
- memoria mediamente libera

Come si può notare dai grafici nel GCTest è risultata come migliore la soluzione client nel caso venga assegnato un valore di heap basso, mentre se il valore di heap è alto è più efficiente la versione server.
Questo perché con poca memoria la parte contenente gli oggetti long-lived viene riempita molto più velocemente nella versione server. Nella versione client invece, avendo una parte di memoria assegnata agli oggetti short-lived minore, la parte di heap tenuered sarà riempita meno velocemente.
Perciò nel caso client saranno più frequenti le GC degli oggetti young e meno frequenti le FullGC. In questo modo, essendo le pause dovute alle FullGC molto maggiori, nel caso di scarsa memoria viene avvantaggiata la versione client.
Con una memoria disponibile molto alta, la versione server riesce a gestire quasi completamente la creazione e distruzione degli oggetti con la parte di heap young permettendo di avere pause molto brevi. Nella versione client invece si è costretti comunque ad utilizzare la parte di heap tenuered causando pause più lunghe dovute alle FullGC.

Nel GCHalfTest la metà dei nodi della lista appartengono alla categoria di oggetti long-lived e quindi la versione server riesce, anche con bassi quantitativi di heap, a gestire le nuove variabili (altra metà della lista) tramite la parte di

memoria young.
Al contrario la versione client si vedrà costretta ad analizzare più frequentemente la parte tenuered dato che verrà riempita più velocemente a causa della presenza di molti oggetti long-lived.
Perciò, con i benefici derivanti dalla riduzione delle pause dovute alle FullGC, la versione server è competitiva in termini di tempi d'esecuzione fin da valori di heap bassi.

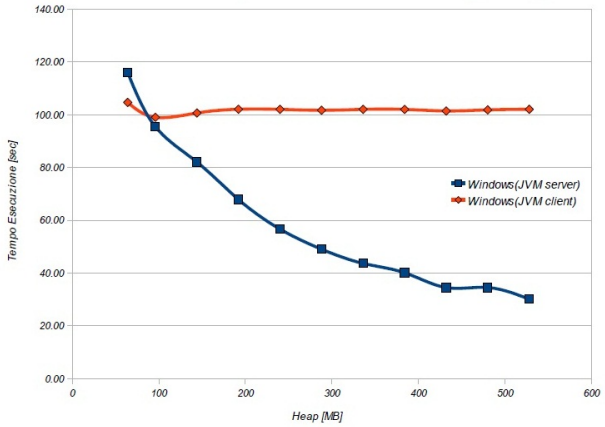
Da questi test si può comprendere come, differenti impostazioni della memoria (parametro *NewRatio*), portino ad avere risultati in termini di tempi d'esecuzione molto diversi tra loro e che questo sia dovuto a un comportamento non corretto del GC. Inoltre è possibile che con un rapporto errato tra area young e area tenuered della memoria i benefici derivanti da un aumento del quantitativo massimo di heap possono essere praticamente nulli.
In conclusione, è molto importante verificare lo stato di funzionamento del GC per diversi valori del *NewRatio* in modo da ottenere la maggior efficienza possibile per ogni quantitativo di memoria massima impostata.

Analisi consumi energetici

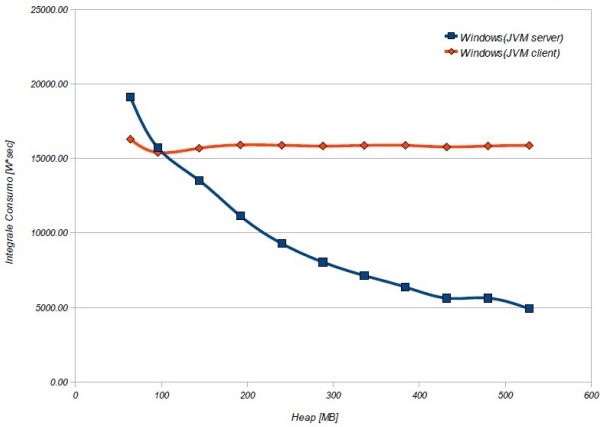
Nei test per l'analisi dei consumi energetici è stato scelto di allineare i valori di memoria settati per la JVM Client a quelli per la JVM Server, per avere un raffronto più diretto tra i due casi. Quindi entrambe le serie di test partono da un quantitativo di memoria massima pari a 96 MB fino ad arrivare a 6 volte quest'ultimo.

I valori analizzati riguardano i tempi d'esecuzione e i consumi totali della macchina monitorata (application server); i grafici che rappresentano tali risultati sono visibili in Figura 6.3 e 6.4. Più in specifico è stato osservato:

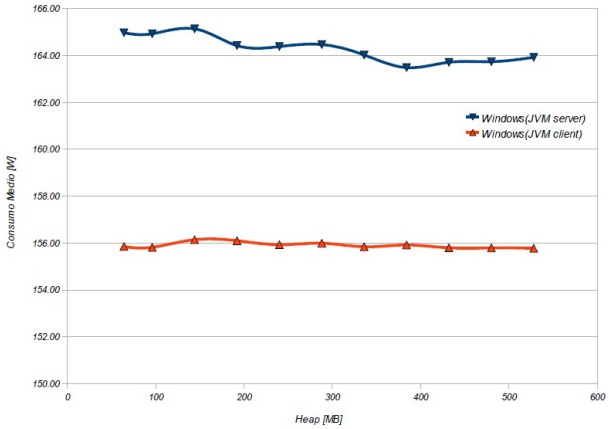
- l'integrale del consumo energetico



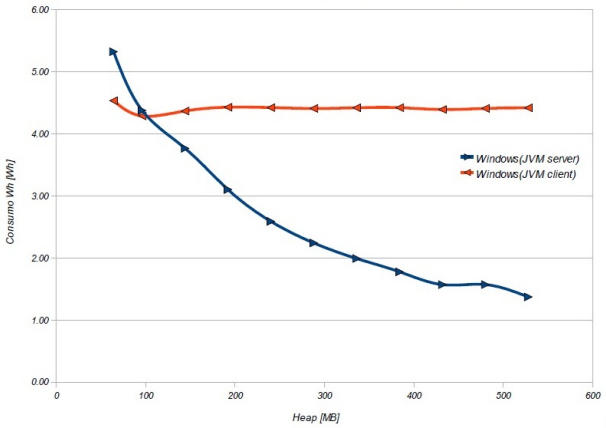
(a) Tempo d'esecuzione al variare della memoria massima



(b) Integrale del consumo al variare della memoria massima

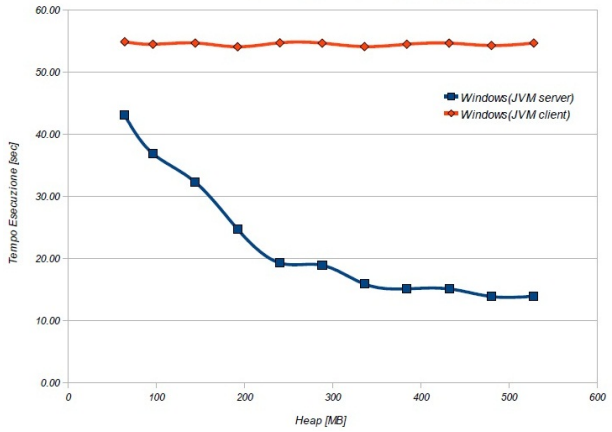


(c) Consumo medio al variare della memoria massima

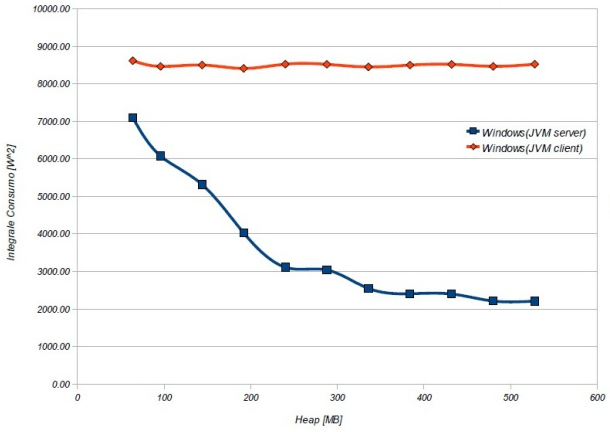


(d) Consumo Wh al variare della memoria massima

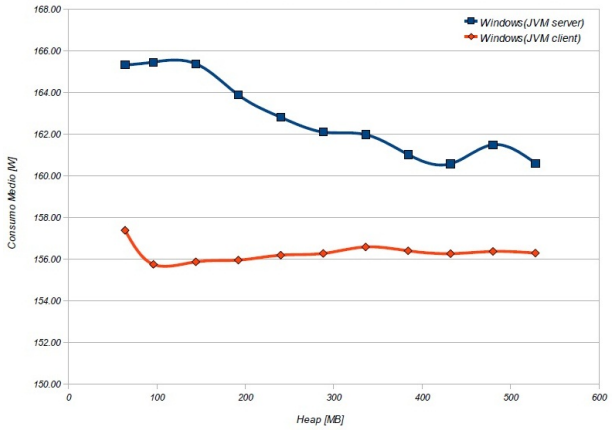
Figura 6.3: *Consumi GCTest: JVM Client e Server*



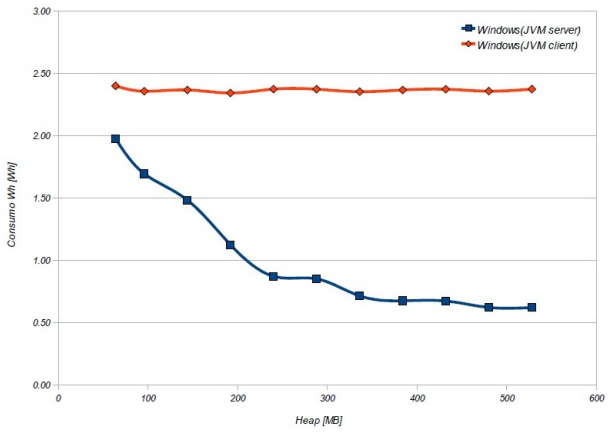
(a) Tempo d'esecuzione al variare della memoria massima



(b) Integrale del consumo al variare della memoria massima



(c) Consumo medio al variare della memoria massima



(d) Consumo Wh al variare della memoria massima

Figura 6.4: *Consumi GCHalfTest: JVM Client e Server*

- il consumo medio
- il consumo Wh

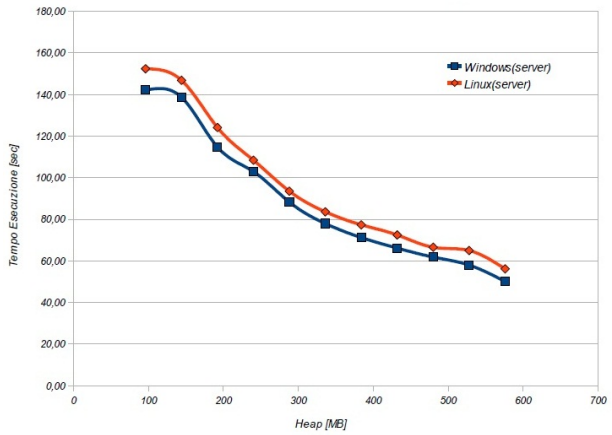
Come osservato precedentemente nell'analisi sul comportamento del GC, la JVM Client risulta più efficiente in termini di tempo d'esecuzione solo nel caso di memoria bassissima e test di deallocazione completa dalla lista. Col crescere della memoria massima la JVM Client, al contrario di quella Server, non ottiene miglioramenti significativi. Nel caso di deallocazione di solo metà lista la JVM Client essendo più svantaggiata non risulta mai essere competitiva.
Dai grafici relativi ai consumi si possono trarre due conclusioni. La prima è che i consumi rispecchiano l'andamento del tempo d'esecuzione e quindi, i discorsi fatti sull'efficienza relativa ai tempi d'esecuzione, possono essere applicati anche a quella sui consumi energetici. La seconda è che tra le due versioni è presente una forte differenza nei consumi medi, perciò si può osservare che la scelta della JVM si ripercuote in modo molto profondo anche sulle caratteristiche del carico di lavoro che è in grado di sopportare e quindi sui consumi energetici.

6.1.2 GCTest Java: Windows 32bit e Linux 64bit

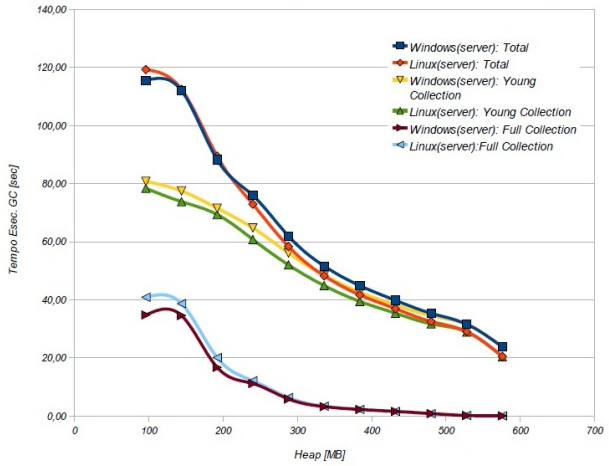
Per questa serie di test i parametri sono stai settati per ottenere un carico di lavoro adattato al tipo di architettura: 32bit per Windows e 64bit per Linux. Il programma calcola le dimensioni finali dei nodi della lista tenendo conto dell'overhead introdotto da puntatori di dimensione maggiore per il secondo tipo di architettura.
In questo modo è possibile posizionare le due JVM sullo stesso piano di lavoro per osservare le differenze a livello di sistema operativo.
Il minimo quantitativo di memoria massima per cui il programma di test riesce a terminare la sua esecuzione si attesta per entrambi i sistemi operativi a 96MB.

Analisi comportamento GC

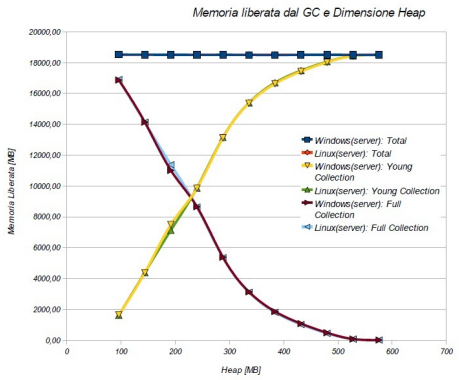
Nelle Figure 6.5 e 6.6 sono rappresentati i grafici dei risultati ottenuti.



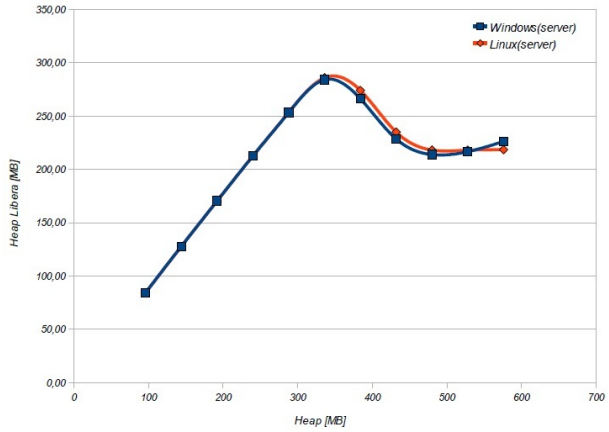
(a) Tempo d'esecuzione al variare della memoria massima



(b) Tempo impiegato dal GC al variare della memoria massima

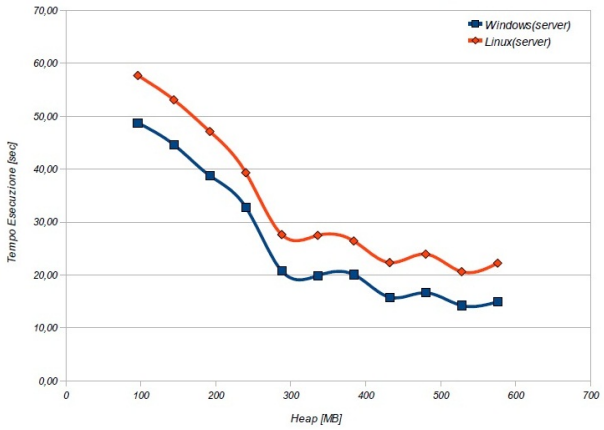


(c) Memoria liberata dal GC al variare della memoria massima

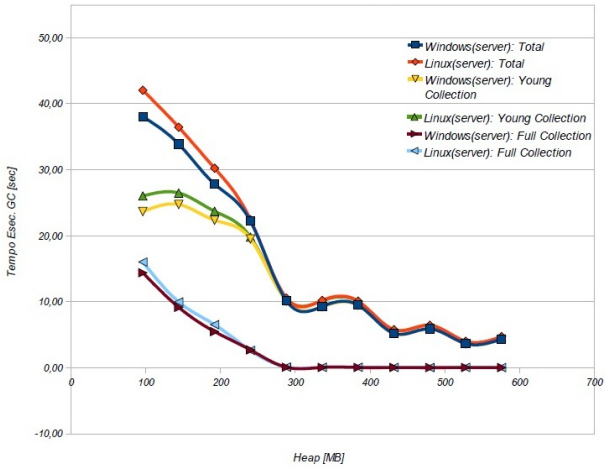


(d) Quantitativo medio di memoria libera al variare della memoria massima

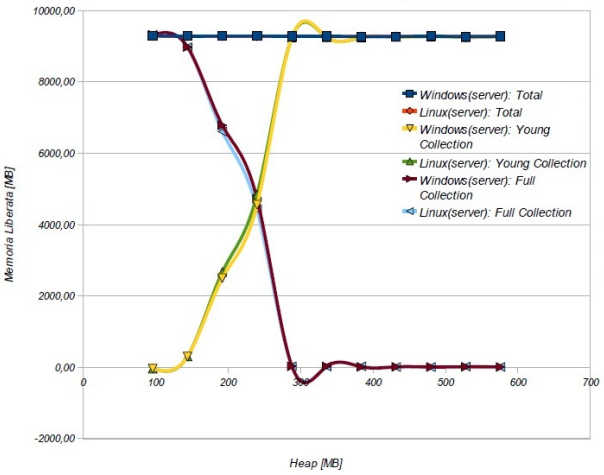
Figura 6.5: *GCTest: Windows e Linux con carico adattato*



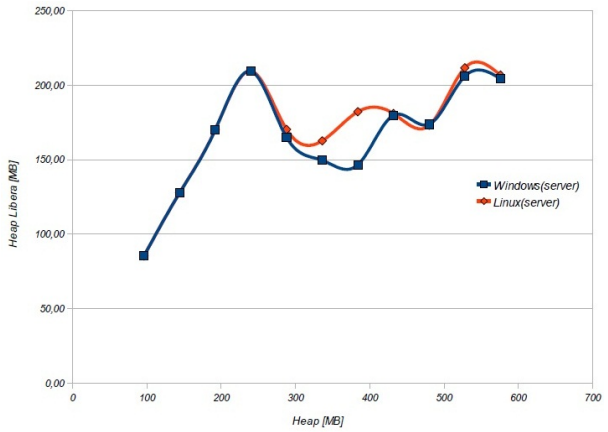
(a) Tempo d'esecuzione al variare della memoria massima



(b) Tempo impiegato dal GC al variare della memoria massima



(c) Memoria liberata dal GC al variare della memoria massima



(d) Quantitativo medio di memoria libera al variare della memoria massima

Figura 6.6: *GCHalfTest: Windows e Linux con carico adattato*

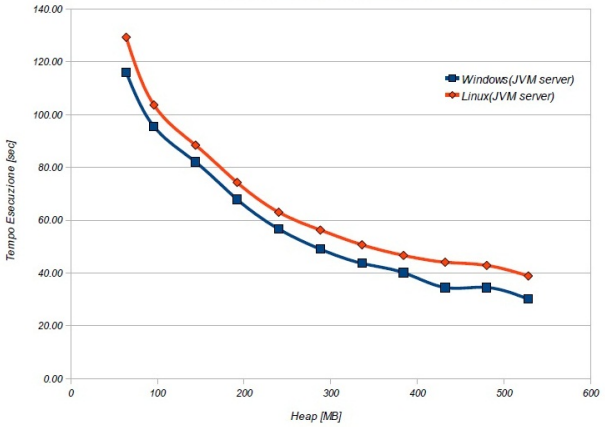
Per il GCTest le differenze in termini di tempi d'esecuzione fra i due sistemi operativi sono minime: mediamente Windows è più veloce di un 8% circa.
Le differenze non sono dovute però al tempo occupato dal GC perchè su tutti i valori di heap i tempi di GC sono praticamente identici.
Lo stesso discorso vale per la memoria liberata e per quella mediamente libera durante l'esecuzione del test.
Nel caso di GCHalfTest le differenze in termini di tempo d'esecuzione sono maggiori: in alcuni casi si arriva a un +20% con Linux.
Per quanto riguarda gli altri parametri del GC non sono presenti differenze se non minime tranne per la memoria mediamente disponibile. Con alcuni valori di heap (336 e 384 MB) Windows occupa fino al 20% circa di memoria in più.

In conclusione la JVM disaccoppia in modo abbastanza efficace lo stack sottostante (Windows o Linux) dalla gestione della memoria. Come si può osservare, dai valori medi di consumo, le differenze che si ottengono tra i due sistemi operativi molto probabilmente sono da imputare alla priorità con cui viene eseguito il programma o alla maggior complessità di un architettura a 64bit.

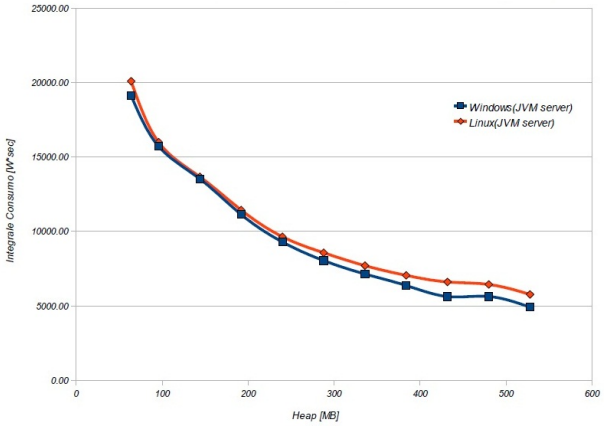
Analisi consumi energetici

Per i test relativi ai consumi sono stati adottati gli stessi quantitativi di memoria massima dei test precedenti.
Nelle Figure 6.7 e 6.8 vengono mostrati i grafici che rappresentano i risultati dei test.

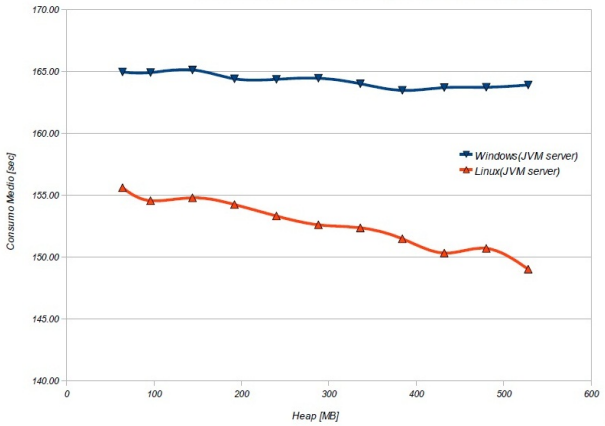
Come si può osservare, anche in questo caso, i test portano ad avere un tempo d'esecuzione molto simile per i due sistemi operativi. I diversi valori ottenuti rispetto al caso precedente sono la conseguenza della diversa metodologia con cui è stato lanciato il programma. Nel primo caso viene lanciato creando una JVM stand alone mentre in questo caso ne viene creata un'altra dal Workload Manager Client; ciò porta ad avere diversi tempi di startup. Dai grafici si nota che ancora



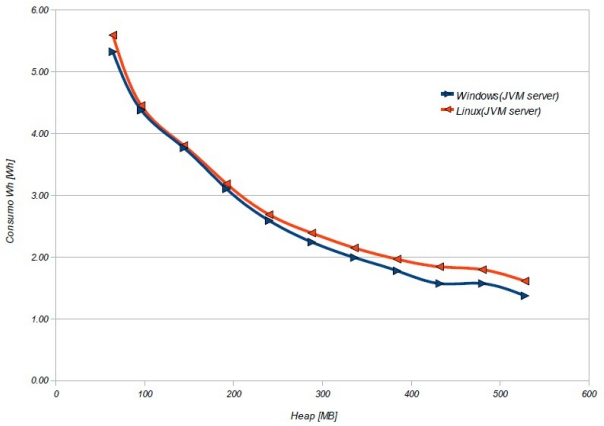
(a) Tempo d'esecuzione al variare della memoria massima



(b) Integrale del consumo al variare della memoria massima

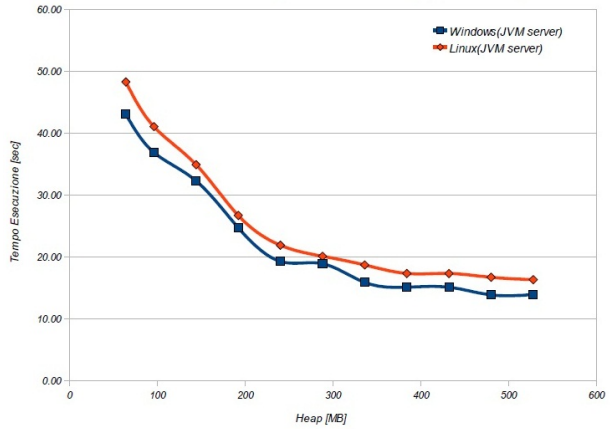


(c) Consumo medio al variare della memoria massima

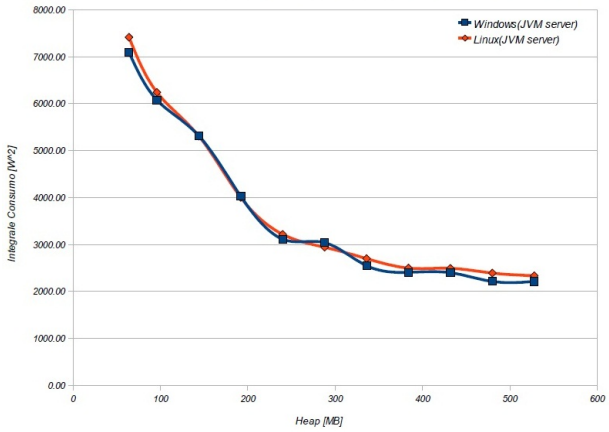


(d) Consumo Wh al variare della memoria massima

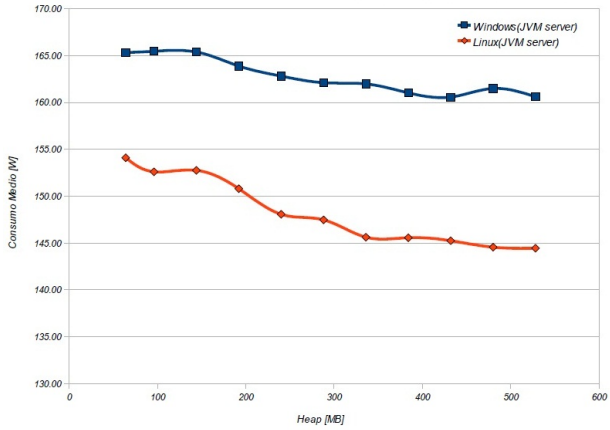
Figura 6.7: *Consumi GCTest: Windows e Linux con carico adattato*



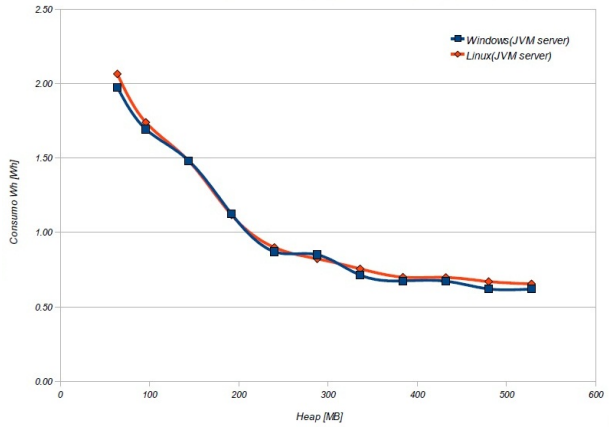
(a) Tempo d'esecuzione al variare della memoria massima



(b) Integrale del consumo al variare della memoria massima



(c) Consumo medio al variare della memoria massima



(d) Consumo Wh al variare della memoria massima

Figura 6.8: *Consumi GCHalfTest: Windows e Linux con carico adattato*

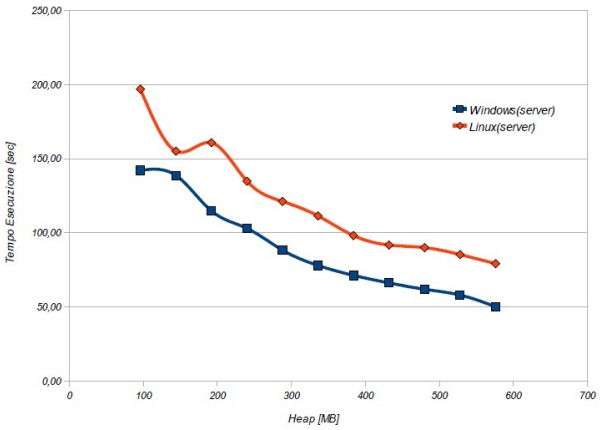
una volta Windows ottiene tempi d'esecuzione leggermente inferiori.
La differenza tra il consumo medio in Windows e Linux avvalora l'ipotesi della diversa priorità con cui viene eseguita la JVM nei due sistemi operativi. Difatti Windows è in grado di eseguire più velocemente il test ottenendo però un valore medio di consumo più alto, mentre in Linux si presenta la situazione inversa.
Le differenze dei consumi medi e dei tempi d'esecuzione portano però ad un integrale di consumo identico. quindi si può affermare che Windows risulta più efficiente, seppur di poco, dato che riesce a terminare più velocemente il test consumando la stessa quantità di energia di Linux.
Il test svolto tenta di mettere sullo stesso piano due sistemi operativi con architetture differenti adattando le dimensioni dei nodi della lista. Si pensa che ciò probabilmente non sia sufficiente ad eliminare l'overhead intrinseco dell'architettura a 64 bit. Quindi le differenze in termini di prestazioni sono da imputare in parte anche al carico maggiore derivante dalla gestione da parte del kernel di uno spazio di indirizzamento più ampio.

6.1.3 GCTest Java: Windows e Linux carico uguale

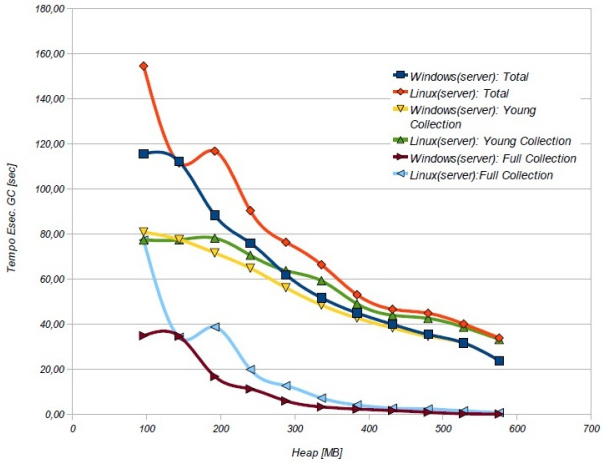
In questo caso di test viene associato a entrambi i sistemi operativi lo stesso carico di lavoro (architettura 32bit) in modo da porre a confronto i due sistemi operativi in termini assoluti.
Nonostante le maggiori dimensioni della lista, Linux, riesce comunque a portare a termine il test con 96MB di heap massima.

Analisi comportamento GC

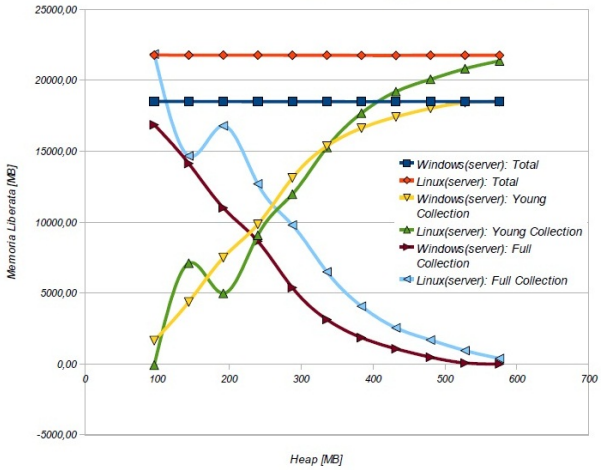
Nelle Figure 6.9 e 6.10 sono descritti i risultati dei test.



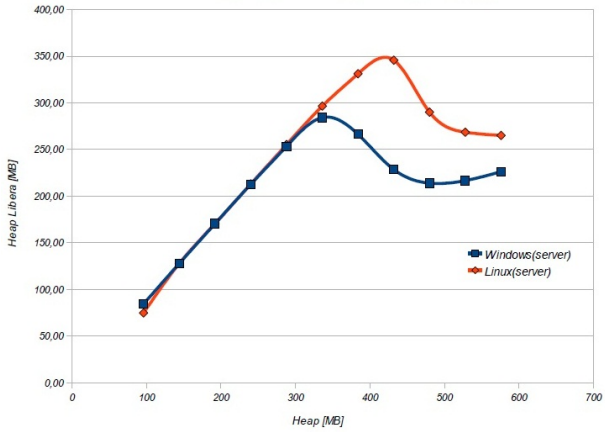
(a) Tempo d'esecuzione al variare della memoria massima



(b) Tempo impiegato dal GC al variare della memoria massima

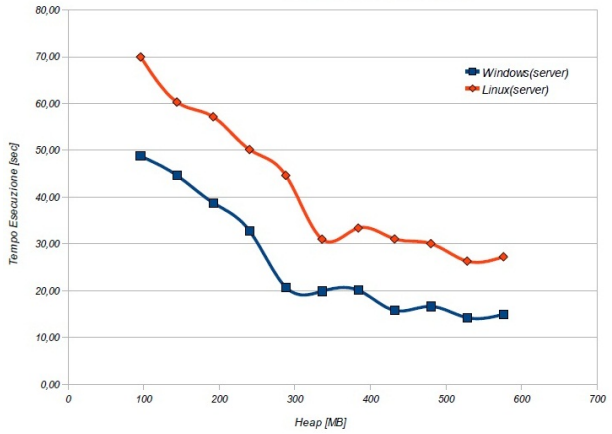


(c) Memoria liberata dal GC al variare della memoria massima

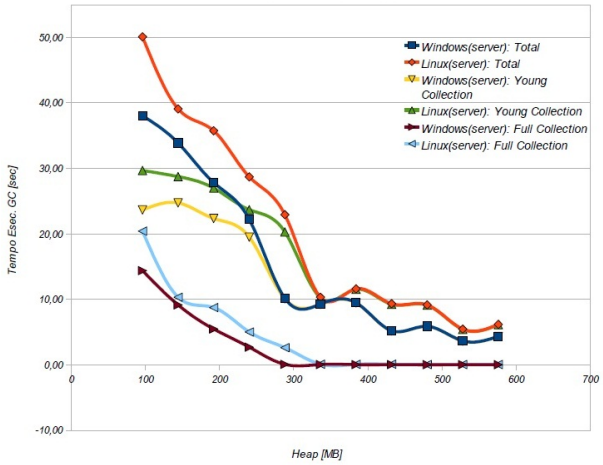


(d) Quantitativo medio di memoria libera al variare della memoria massima

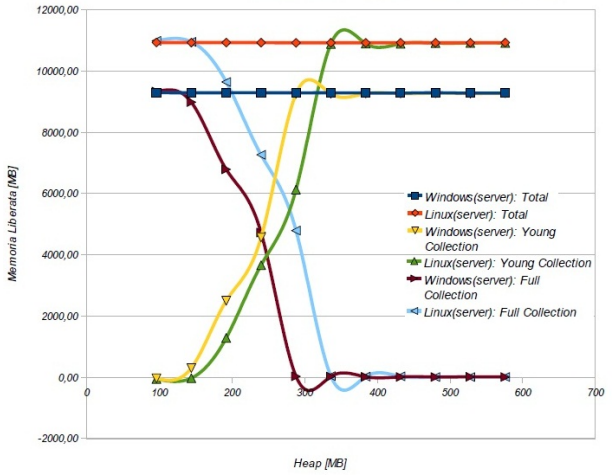
Figura 6.9: *GCTest: Windows e Linux con carico uguale*



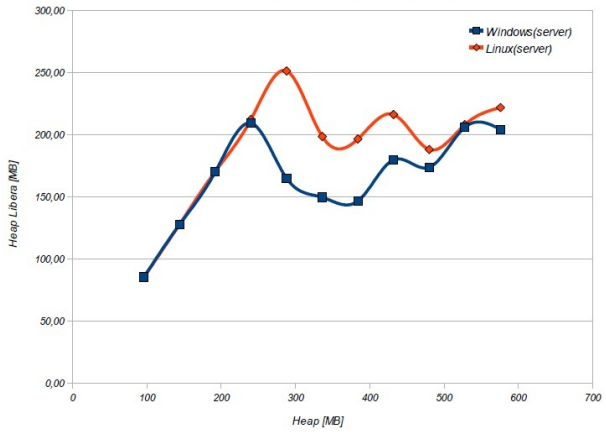
(a) Tempo d'esecuzione al variare della memoria massima



(b) Tempo impiegato dal GC al variare della memoria massima



(c) Memoria liberata dal GC al variare della memoria massima



(d) Quantitativo medio di memoria libera al variare della memoria massima

Figura 6.10: *GCHalfTest: Windows e Linux con carico uguale*

Si nota subito che in questo caso Linux ottiene risultati ben peggiori su quasi tutti gli aspetti: tempo d'esecuzione, tempo impiegato dal GC e memoria gestita. Per quanto riguarda la memoria mediamente libera si notano vantaggi per Windows solo da un certo quantitativo di memoria massima in su: 240MB per il GCHalfTest e 336MB per il GCTest.
L'overhead di memoria che il GC deve gestire durante il test in Linux è calcolabile con la formula:

$$MemoryOvh = (6 * ref64bit - 6 * ref32bit) * NumNodes * AssignLoop$$

Che nel caso dei valori impostati per l'esecuzione del test rappresenta un overhead del 18,75% circa. Nel caso di GCHalfTest il quantitativo aggiuntivo di memoria da liberare è esattamente la metà dato che gli oggetti vivi dall'inizio alla fine dei test sono la metà del totale. Ciò non varia però la percentuale dell'overhead di memoria.
Questo svantaggio per il sistema operativo Linux si ripercuote direttamente sui tempi d'esecuzione, causandone un aumento rispetto ai test precedenti fino al 21% per il GCHalfTest e fino al 29% per il GCTest.

Analisi consumi energetici

In conclusione si può affermare che un sistema operativo a 64 bit comporta un carico maggiore del GC e ciò si ripercuote sulle prestazioni dei programmi. Considerando che il consumo medio durante l'esecuzione di questo test in tutti i casi è risultato praticamente costante, si può affermare con buona approssimazione che i consumi totali dell'applicazione possono crescere di circa il 20% per il GCHalfTest e circa del 30% per il GCTest.
L'adozione di un sistema operativo di questo tipo deve perciò essere valutata attentamente prendendo in considerazione i vantaggi derivanti dalla possibilità di poter utilizzare un maggiore quantitativo di ram e gli svantaggi derivanti dalla

gestione più onerosa della memoria.

6.1.4 GCTest C

Il programma GCTest per il C è stato creato in due versioni differenti: una con e una senza GC.
La versione senza GC utilizza i classici metodi per l'allocazione e la deallocazione della memoria, ovvero il comando *malloc* per l'allocazione e il comando *free* per la deallocazione.
Nel benchmark che utilizza il GC è stato scelto di adottare il *Boehm-Demers-Weiser GC* poiché è quello più famoso e utilizzato.
Per utilizzare tale GC è sufficiente includere all'interno del programma un determinato header file (*gc.h*) e compilarlo assieme alla libreria *libgc.a*. In questo modo è possibile accedere alle funzionalità offerte dal BDW GC. Le primitive per la gestione della memoria offerte sono:

- *GC_MALLOC*
- *GC_CALLOC*
- *GC_REALLOC*
- *GC_FREE*

Tutti questi comandi devono essere utilizzati al posto di quelle corrispondenti utilizzate normalmente in C. Va sottolineato che il comando *GC_FREE* può essere utilizzato al posto del comando *free* ma può anche essere omesso dato che non effettua nessuna operazione. Difatti sarà il GC ad analizzare e nel caso liberare determinate aree di memoria se non più raggiungibili.
Nel capitolo relativo allo *Stato dell'Arte* è stato già spiegato il funzionamento del BDW GC, come può essere utilizzato e il significato del parametro *Free Space*

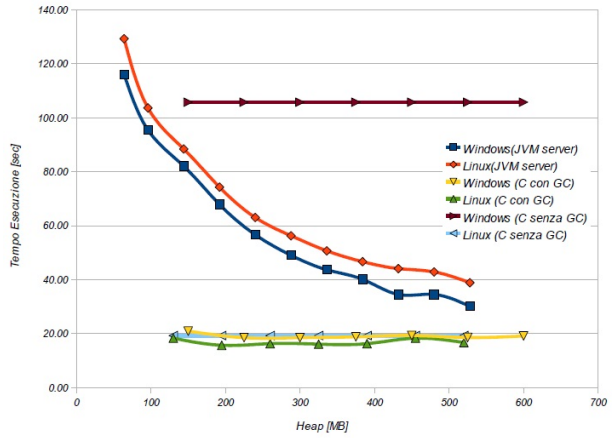
Divisor. Il FSD di fatto indica al GC quando intervenire o quando espandere l'area di memoria utilizzata secondo l'algoritmo mostrato in Figura 2.7.

Java contro C

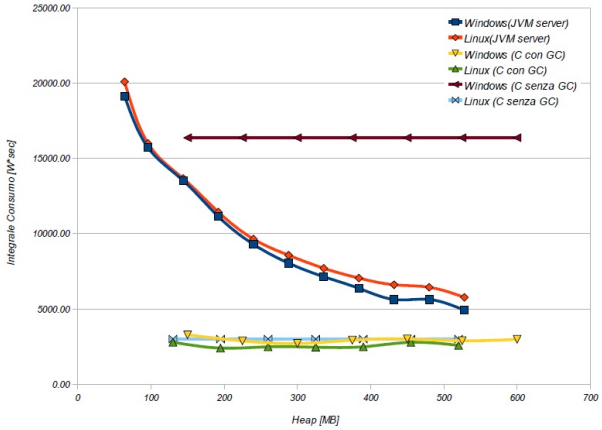
Lo scopo di questi test è confrontare l'efficienza delle tre versioni del GCTest ovvero quella in Java e le due in C. Per svolgere questi test è stata impostata tramite la variabile apposita del BDW GC, chiamata *GC_MAXIMUM_HEAP_SIZE*, la dimensione massima della heap del programma.

Nei grafici rappresentati nelle Figure 6.11 e 6.12 viene mostrato il confronto tra Java e C in entrambi i sistemi operativi. Ovviamente il carico del test è adattato al tipo di architettura di Windows e Linux.

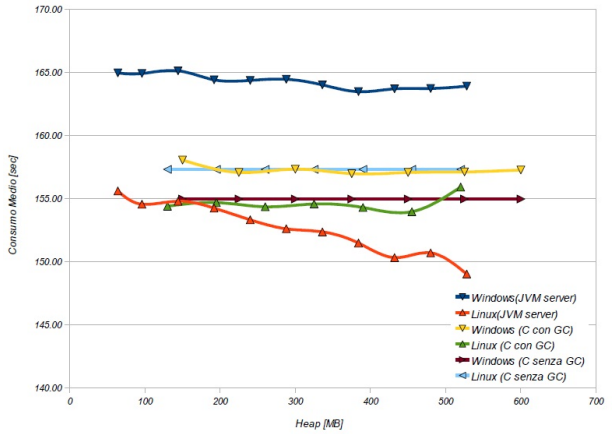
Come si pensava, il GC utilizzato per il programma di test non è influenzato dalla dimensione massima di memoria assegnata poiché reagisce esclusivamente in base al valore assegnato al FSD. Perciò la dimensione della memoria massima occupata dipende in modo quasi deterministico dal rate di creazione delle variabili, dal rate di morte di tali variabili e dal valore specificato per il parametro.
La memoria occupata dalla versione senza GC risulta essere sempre la minima possibile: nel momento in cui la lista è completa si attesta intorno al valore di 45MB ($\approx 347000\ nodi * 128 byte$). Nella versione che utilizza il GC con FSD uguale a 4, che rappresenta il valore di default e quello utilizzato nei test, si attesta invece attorno ai 150MB in Windows e attorno ai 130MB in Linux.
Risulta quindi che la versione con GC comporta un overhead di memoria pari a circa il 230%.
Un FSD maggiore può diminuire l'occupazione di memroia a svantaggio però del tempo d'esecuzione totale



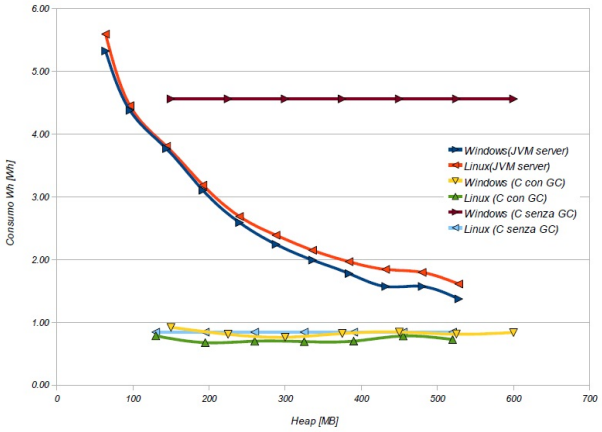
(a) Tempo d'esecuzione al variare della memoria massima



(b) Integrale del consumo al variare della memoria massima

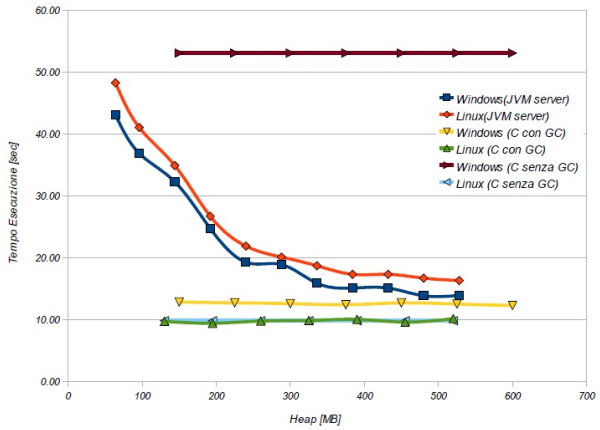


(c) Consumo medio al variare della memoria massima

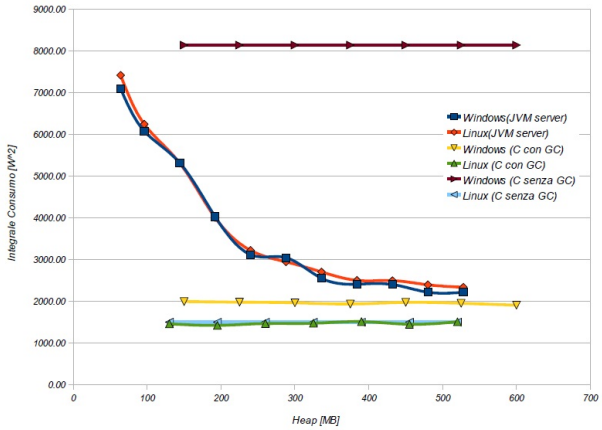


(d) Consumo Wh al variare della memoria massima

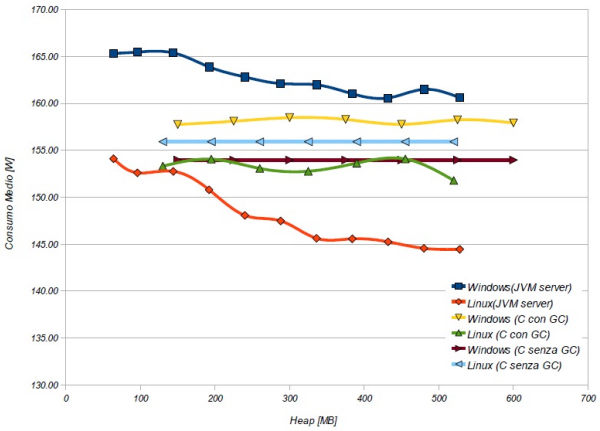
Figura 6.11: *Consumi GCTest in C e Java: Windows e Linux con carico adattato*



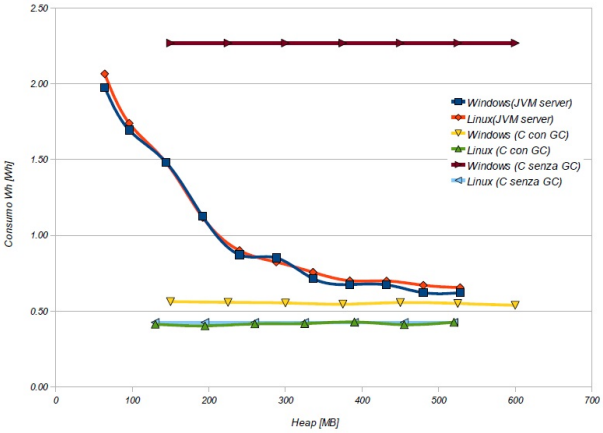
(a) Tempo d'esecuzione al variare della memoria massima



(b) Integrale del consumo al variare della memoria massima



(c) Consumo medio al variare della memoria massima



(d) Consumo Wh al variare della memoria massima

Figura 6.12: *Consumi GCHalfTest in C e Java: Windows e Linux con carico adattato*

Come si può notare immediatamente, nel caso del test senza GC eseguito in Windows, si ottengono risultati pessimi rispetto alla stessa versione su Linux e rispetto alle versioni con GC. Dato che il codice del programma è identico molto probabilmente è presente un problema a livello di compilazione del programma che non è stato approfondito.

Nel GCTest le versioni C ad esclusione del caso di Windows ottengono risultati simili per tutti i parametri analizzati. Un leggero scostamento a favore di Windows è presente nel consumo medio.
Le versioni Java eseguite in Linux ottengono, rispetto la versione C e a parità di memoria, un tempo d'esecuzione maggiore fino al 350% e un consumo totale maggiore fino al 400%. Nel caso di Windows le differenze tra Java e la versione C con GC sono inferiori:fino al 300% per il tempo d'esecuzione e fino al +313% per il consumo totale.
La versione C priva di GC eseguita in Windows si comporta in modo peggiore rispetto al programma Java: il tempo d'esecuzione è maggiore fino al 260% mentre il consumo totale fino al 220%.

Nel GCHalfTest si riscontra una situazione simile per andamento dei tempi d'esecuzione e dei consumi ma con differenze tra C e Java.
In Linux le versioni Java ottengono rispetto a quelle C un tempo d'esecuzione maggiore fino al 260% e un consumo integrale maggiore fino al 284%.
Per Windows la situazione risulta comunque a favore per la versione C con GC dato che il test in Java fa segnare un tempo d'esecuzione maggiore fino al 151% e un consumo integrale maggiore fino al 166%. Anche in questo caso il programma C privo di GC risulta molto peggiore rispetto a quello Java: tempo d'esecuzione maggiore fino al 280% e consumo totale fino al 268%.

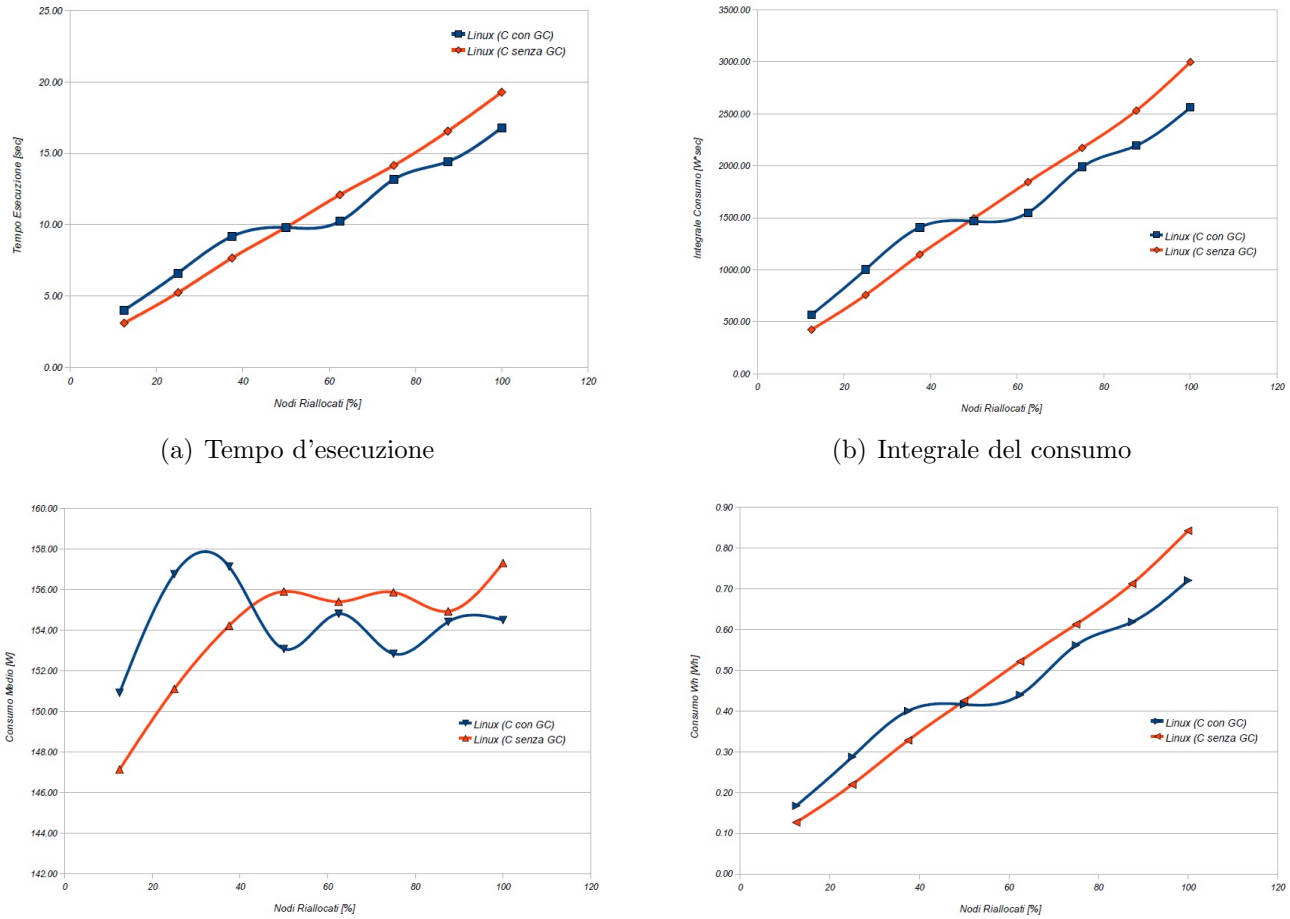
In conclusione, le versioni C ottengono quasi sempre prestazioni nettamente migliori delle versioni Java. Inoltre l'introduzione del GC in C permette miglioramenti poco sensibili in Linux ma molto alti in Windows.

Comportamento del GC in C

Nei paragrafi precedenti è stato analizzato come il GC e le applicazioni Java si comportano diversamente in base al rapporto tra oggetti long e short lived e al NewRatio impostato. Dato che nel BDW GC non è presente quest'ultimo parametro sono stati eseguiti alcuni test per comprendere come variano i consumi al solo variare della percentuale di oggetti short lived.
Come già illustrato precedentemente il numero di nodi reallocati è stato fatto variare dal 12,5% al 100% con intervalli del 12,5%.
Nella Figura 6.13 vengono illustrati i risultati ottenuti dai test. I grafici si riferiscono al sistema operativo Linux; i risultati ottenuti con Windows, in questo caso, non sono stati presi in considerazione perchè difficilmente confrontabili.

Dai risultai si può facilmente comprendere che, in presenza di molti oggetti long lived, il BDW GC risulta svantaggiato perché costretto ad analizzare la memoria anche quando non sono presenti variabili da deallocare. Viceversa, nel caso con un'alta percentuale di oggetti short lived, ottiene risultati sensibilmente migliori poiché con una sola analisi è in grado di deallocare molte più variabili e in modo più efficiente rispetto alla deallocazione classica (*free()*).
Le versioni con e senza GC ottengono gli stessi risultati quando il numero di nodi reallocati è pari al 50%.

Nel caso si ponesse un valore maggiore all'FSD le due curve si intersecherebbero ad una percentuale maggiore di variabili short lived ottenendo però un valore di memoria occupata inferiore. Questo perché il GC diverrebbe più invasivo. Se invece il valore dell'FSD venisse posto ad un valore minore si verificherebbe una



(a) Tempo d'esecuzione

(b) Integrale del consumo

(c) Consumo medio

(d) Consumo Wh

Figura 6.13: *Consumi GCTest in C al variare del numero di nodi reallocati*

situazione diametralmente opposta alla precedente.
Alla luce di questa considerazione si può affermare che per questa tipologia di carico l'FSD standard, che ha valore pari a 4, rappresenta un buon tradeoff per le diverse percentuali di variabili long lived.

In conclusione, anche nel caso del GC per C, come in quello Java, è molto importante settarlo in base alla tipologia delle variabili trattate e del vincolo sulla memoria massima che si vuole soddisfare. Inoltre, se configurato correttamente, può permettere risparmi sia in termini di tempo d'esecuzione che di consumo.

6.2 Test su sistemi ERP

In questo capitolo vengono illustrati i risultati ottenuti con i test effettuati sugli ERP Adempiere e Openbravo. Ogni test relativo a una determinata configurazione è stato ripetuto cinque volte, per ottenere un intervallo di confidenza per la media, al 99%, di dimensione inferiore al 10% del valore della media aritmetica. Nel capitolo precedente sono stati approfonditi i dettagli relativi a questa scelta.

Ogni volta che venivano completati i test relativi a una configurazione di memoria il database veniva riportato allo stato iniziale. In questo modo è stato possibile eliminare ogni eventuale overhead di computazione dovuto al crescere delle dimensioni della base di dati durante le varie esecuzioni.

Ogni ERP è stato testato utilizzando diversi quantitativi di memoria: dalla minima possibile a quattro volte o più la minima, con incrementi pari alla metà della memoria minima.
La memoria minima è stata individuata testando il programma da un quantitativo di heap massima estremamente basso, fino ad arrivare al quantitativo che

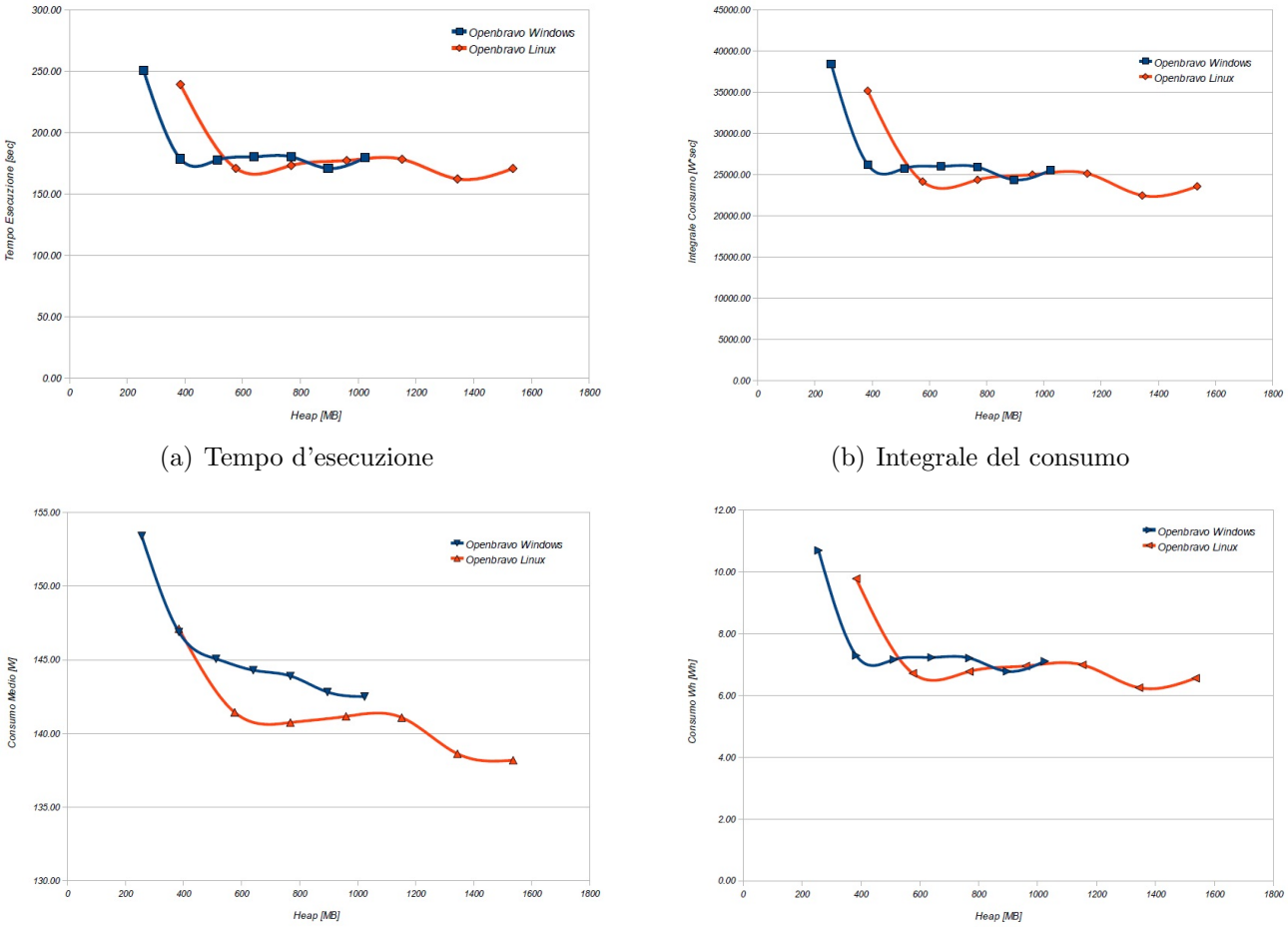
permetteva di avere una buona stabilità dell'applicazione.

Il carico di test è stato impostato, tramite l'interfaccia grafica del Worload Manager Server, nel seguente modo:

- *Business partner groups number = 5*
- *Number of greetings = 4*
- *Number of payment terms = 5*
- *Number of invoice schedules = 2*
- *Business partners number = 5*
- *Contacts number for each BP = 5*
- *Number of products category = 3*
- *Number of tasks category = 2*
- *Units of measure number = 1*
- *Number of attribute sets = 2*
- *Number of price lists (sales) = 1*
- *Number of price lists (orders) = 1*
- *Number of products to sell = 1*
- *Number of products to buy = 1*
- *Number of sale orders = 1*
- *Number of lines for sale = 1*
- *Number of purchase orders = 1*
- *Number of lines for purchase = 1*

6.2.1 Openbravo: Windows e Linux

Nella Figura 6.14 sono mostrati i risultati ottenuti con i due sistemi operativi al variare dei diversi quantitativi di memoria.



(a) Tempo d'esecuzione

(b) Integrale del consumo

(c) Consumo medio

(d) Consumo Wh

Figura 6.14: *Consumi di Openbravo su Windows e Linux al variare della memoria massima*

Come si può notare la memoria minima per i due sistemi operativi risulta essere differente: 256MB per Windows contro i 384 di Linux (+50%). Questa differenza risulta maggiore rispetto a quanto l'opinione comune generalmente considera l'overhead di memoria dei sistemi operativi a 64bit. Difatti viene ritenuto che le applicazioni per tali sistemi operativi generalmente occupano il 30% circa di memoria in più.

Di seguito sono riportati alcuni dati significativi sul comportamento del tempo d'esecuzione al variare della memoria:

- utilizzando una volta e mezza la memoria minima il tempo d'esecuzione si riduce del 28,8% in Windows e del 28,6% in Linux. Oltre questi valori di memoria i vantaggi sono molto meno consistenti.
- a parità di memoria Linux ha un tempo d'esecuzione maggiore rispetto Windows fino al 34% nel caso di bassa memoria disponibile. Per valori più alti Linux può ottenere prestazioni leggermente migliori di Windows.
- in entrambi i sistemi operativi si ottengono le migliori prestazioni con un quantitativo di memoria pari a tre volte e mezzo la memoria minima. Linux fa registrare una diminuzione del 32% mentre Windows del 31% rispetto al caso peggiore (memoria minima).

Per quanto riguarda i consumi si ottengono risultati molto simili a quelli del tempo d'esecuzione:

- a parità di memoria Linux ha un consumo maggiore rispetto Windows fino al 34% nel caso di memoria minima. Per valori di memoria da una volta e mezza la minima in su, Linux ottiene quasi sempre consumi leggermente inferiori rispetto a Windows.
- in entrambi i sistemi operativi si ottengono i migliori consumi con un quantitativo di memoria pari a tre volte e mezzo la memoria minima. Linux fa

registrare una diminuzione del 36,1% mentre Windows del 36,6% rispetto al caso peggiore (memoria minima).

Alla luce dei risultati ottenuti, ricollegandosi allo studio *Quantifying the Performance of GC vs Explicit Memory Management* [15] di cui è già stato discusso nel capitolo sugli studi preliminari, si può ipotizzare che con un valore pari a 3,5 volte la heap minima, l'applicazione ottiene un tempo d'esecuzione pari al valore che otterrebbe senza l'utilizzo del GC, ovvero con gestione esplicita della memoria. Osservando inoltre, che il tempo d'esecuzione è un buon indicatore delle variazioni di consumo totale, si può aggiungere che con tale quantità di heap è possibile raggiungere consumi pari a quelli della stessa applicazione ma con gestione esplicita della memoria.

La forte differenza tra il caso di memoria minima e il caso di una volta e mezzo tale quantitativo di memoria è dovuto al comportamento del GC. Per comprendere meglio le motivazioni, di seguito, sono riportati i dati raccolti analizzando il log del GC durante l'esecuzione di 5 test consecutivi su Linux con i due quantitativi di heap massima.
I risultati con 384MB (memoria minima) sono:

- GC Total Time: 264.26382 sec
- GC Time (young): 2.7715893 sec
- Full GC Time: 261.4922 sec
- Total Released Memory: 42367,512 MB
- GC (young) Released Memory: 16536,860 MB
- Full GC Released Memory: 25830,676 MB
- Medium Available Heap: 355,777

Mentre quelli con 576MB sono:

- GC Total Time: 5.5150642 sec
- GC Time (young): 4.6945252 sec
- Full GC Time: 0.820539 sec
- Total Released Memory: 40735,640 MB
- GC (young) Released Memory: 40728,633 MB
- Full GC Released Memory: 7,7 MB
- Medium Available Heap: 431,742

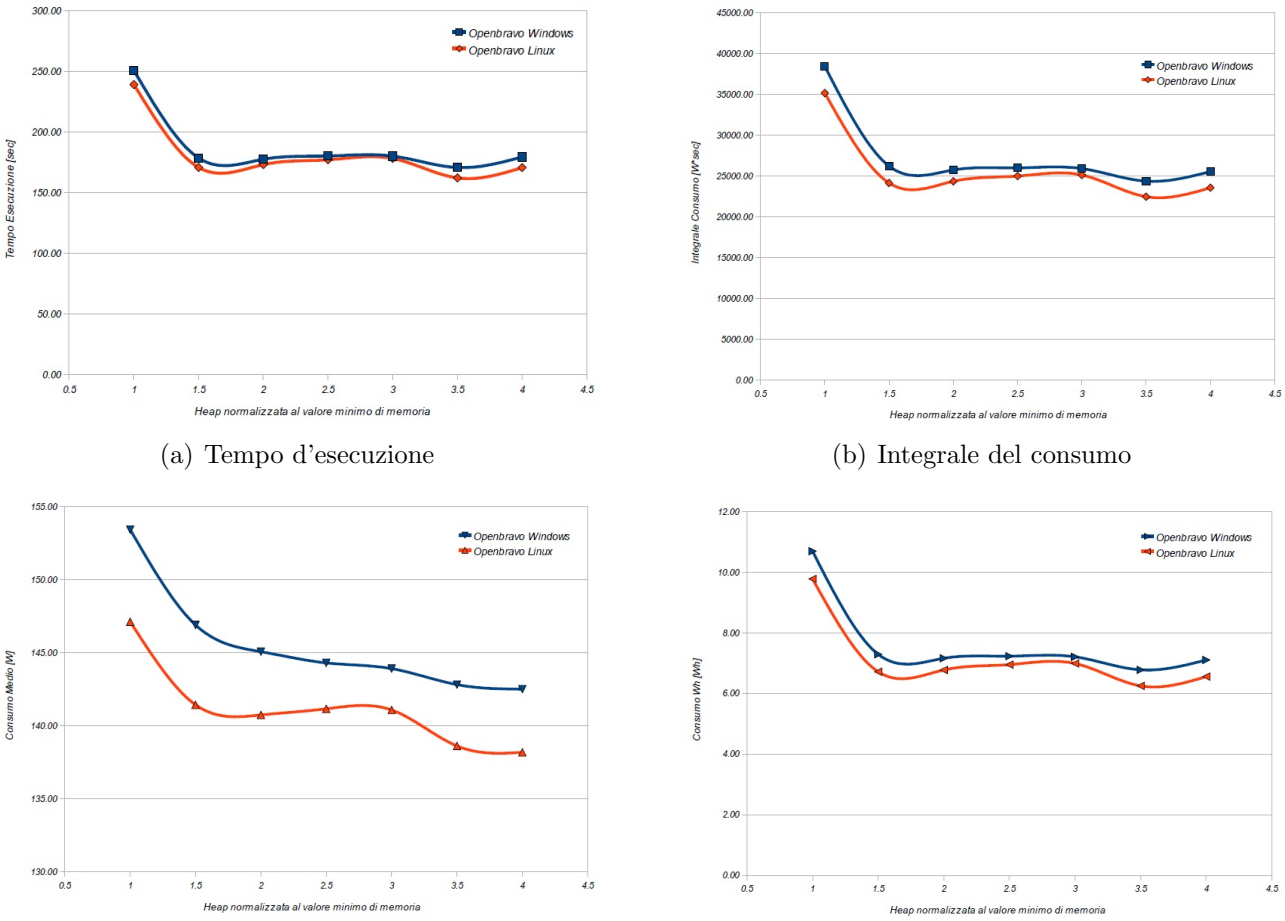
Come si può notare la differenza nella durata delle pause dovute al GC è enorme, 264 secondi contro 5,5 secondi circa. Il motivo principale di ciò è da attribuire all'estrema frequenza con cui vengono richiamate le FullGC dato che l'area young di memoria non è sufficientemente grande da effettuare una scrematura efficace. In questo modo la parte tenuered va a saturarsi molto velocemente. All'aumentare della memoria entrambi questi fenomeni vengono eliminati permettendo di ottenere pause di durata molto breve e migliorando le prestazioni globali.
Le differenze minori, tra il taglio di heap pari a una volta e mezzo il valore minimo e quelli decisamente maggiori, sono da imputare al fatto che il GC non ha più molto margine di miglioramento. La motivazione di questo comportamento deriva da due fattori:

- le FullGC, che comportano le pause più lunghe, sono già state ridotte al minimo con una volta e mezzo la memoria minima.
- i vantaggi derivanti da una frequenza minore delle garbage collection sull'area young, sono quasi completamente compensati dal tempo maggiore per analizzare l'area di memoria divenuta più grande.

Dai grafici risulta evidente che Linux, nonostante l'overhead introdotto dall'architettura a 64 bit, ottiene risultati del tutto comparabili con Windows tranne nel caso di scarsa memoria.

Difatti, se si effettua un confronto tra Linux e Windows normalizzando la quantità di heap al valore minimo relativo a ciascun sistema operativo, come in Figura 6.15, si nota che Linux risulta essere più efficiente di Windows.

Inoltre si nota che il comportamento dei due sistemi operativi è del tutto identico al variare della quantità di memoria.



(a) Tempo d'esecuzione

(b) Integrale del consumo

(c) Consumo medio

(d) Consumo Wh

Figura 6.15: *Consumi di Openbravo su Windows e Linux al variare della memoria massima (valori normalizzati alla memoria minima possibile)*

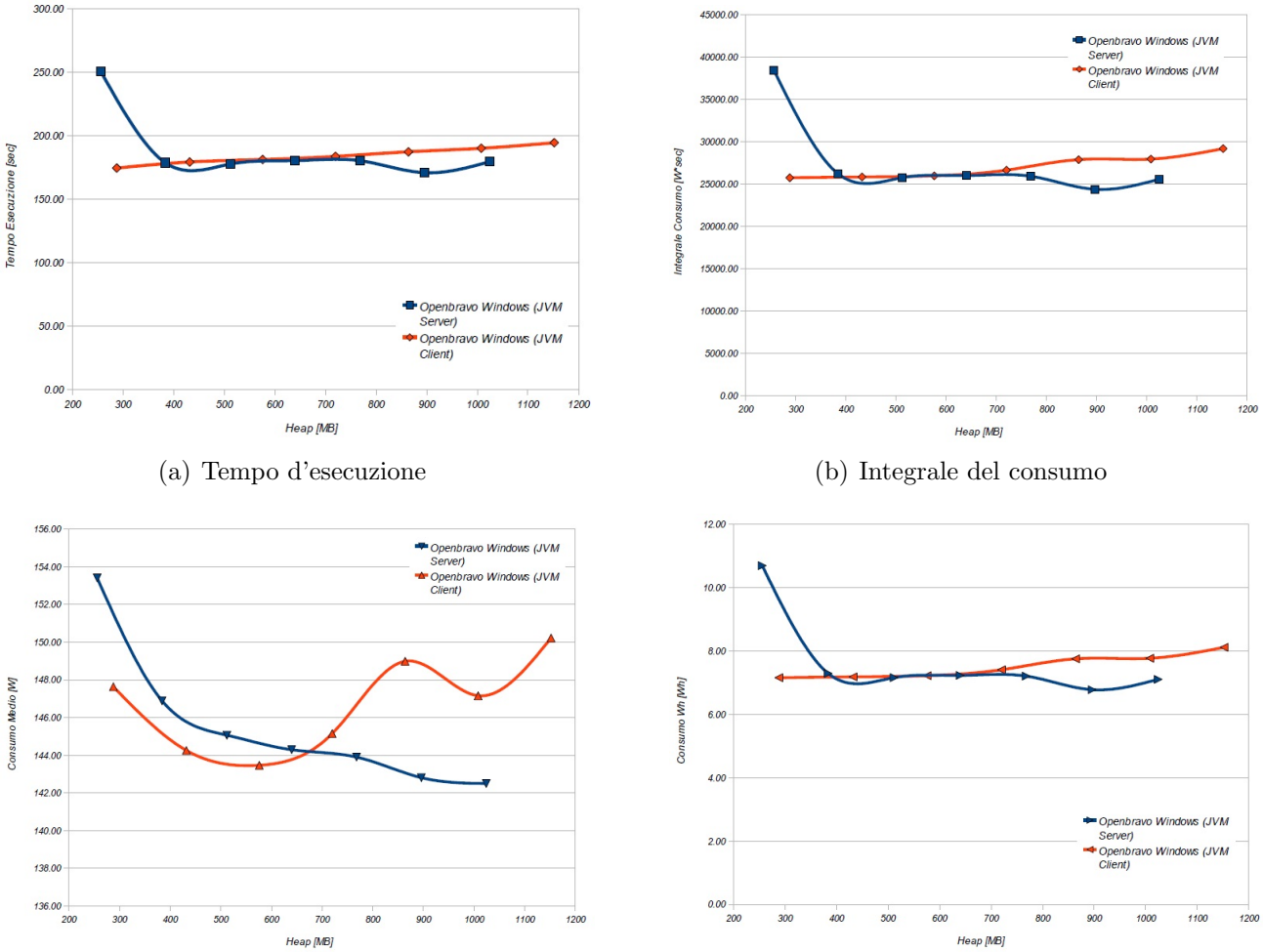
Con ciò si può concludere che Linux risulta essere meno efficiente energeticamente per valori bassi di memoria ma, nel caso di valori appena più alti, risulta essere competitivo con Windows nonostante lo svantaggio derivante dalla tipologia di architettura.
Inoltre dai grafici con heap normalizzata si evidenzia come, non considerando l'overhead di memoria, i due sistemi operativi si comportano in modo del tutto simile. Questo, ancora una volta, evidenzia come la Java Virtual Machine sia in grado di disaccoppiare le prestazioni di un'applicazione dallo stack sottostante.

6.2.2 Openbravo: JVM Client e JVM Server

In questo paragrafo vengono commentati i dati derivanti dai test svolti sull'ERP Openbravo utilizzando due differenti Java Virtual Machine: quella Client e quella Server.
Nella Figura 6.16 sono rappresentati i grafici relativi al tempo d'esecuzione e ai consumi rilevati.

Osservando i risultati ottenuti si può notare che, con un quantitativo molto basso di memoria (il minimo possibile), la JVM versione Client risulta più efficiente, sia in termini di tempi d'esecuzione che di consumi, rispetto alla versione Server. Questo fatto è da attribuire alla maggiore dimensione dell'area tenuered e alla conseguente quantità minore di FullGC. Al contrario nella versione server, come già osservato, esse sono numerosissime.

D'altra parte, però, la versione Client, per rendere stabile l'applicazione, necessita di una quantità di heap massima pari a 288MB contro i 256MB necessari a quella Server (-11%). Molto probabilmente la ragione di questa differenza, opposta a quella riscontrata nei GCTest, è da attribuire al fatto che le pause dovute al GC, con un NewRatio pari a $\frac{8}{9}$, non permettono di soddisfare tutte le richieste inviate all'application server. Il motivo di questo comportamento potrebbe essere



(a) Tempo d'esecuzione

(b) Integrale del consumo

(c) Consumo medio

(d) Consumo Wh

Figura 6.16: *Consumi di Openbravo su JVM Client e Server al variare della memoria massima*

dovuto all'overhead introdotto dalla moltitudine di oggetti da copiare nella parte tenuered di memoria. Oppure al carico maggiore introdotto dall'analisi di un'area di memoria, sempre quella tenuered, molto più grande rispetto a quella della versione Server. Quest'ultima possibilità può essere il motivo per cui all'aumentare della memoria le prestazioni e i consumi peggiorano sempre.
La crescita del consumo medio della versione client, con memorie abbastanza alte, avvalora la tesi secondo cui il GC è soggetto a un carico maggiore all'aumentare della memoria.
Quindi è scorretto pensare che con un aumento della memoria si ottengono sempre miglioramenti, difatti, come in questo caso, con una configurazione del GC non propriamente corretta si possono ottenere svantaggi al crescere della heap.

Associando alla JVM Client la minima quantità possibile di heap si ottiene, rispetto ai risultati della versione Server a parità di memoria, un tempo d'esecuzione minore del 24,1% e dei consumi minori del 26,5%.
Per i quantitativi di memoria superiori a una volta e mezza, invece, è la JVM Server a essere più efficiente sia dal punto di vista energetico che da quello prestazionale. Infatti con tre volte e mezzo la quantità minima ottiene un vantaggio rispetto la JVM Client del 9,1% per il tempo d'esecuzione e del 13% per i consumi.

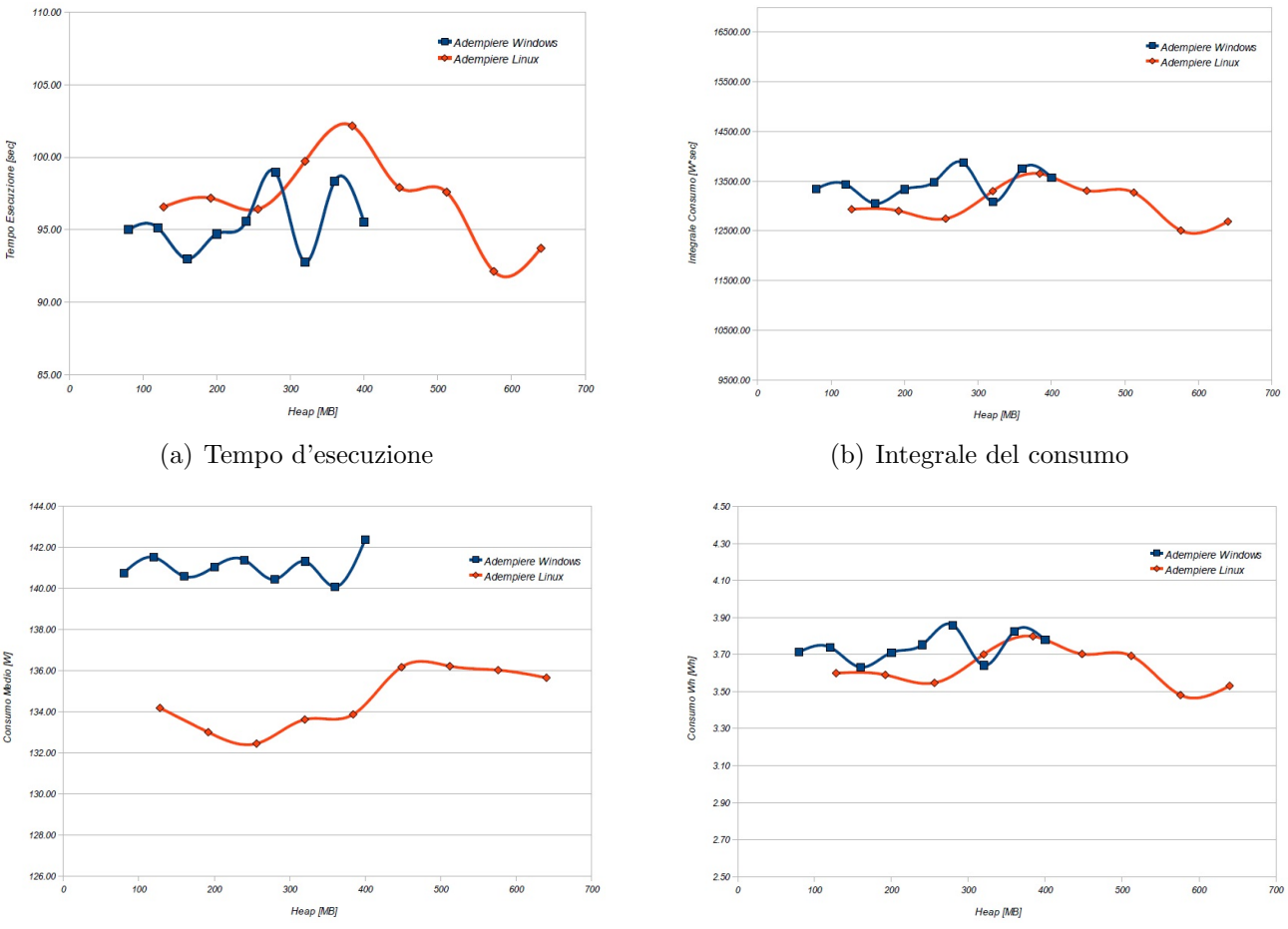
In conclusione si può dire che il carico generato dall'applicazione favorisce la versione server tranne per quantitativi di heap molto bassa. Ciò fa pensare che il quantitativo di oggetti long lived sia sufficientemente alto da avvantaggiare una configurazione con NewRatio basso, come nella JVM Server, ma non abbastanza da rendere la versione Client totalmente inefficiente.

Un dato molto importante riguardante i risultati ottenuti, è che la JVM Client non va scartata a priori per eseguire un'applicazione prettamente *server* come in

questo caso. Perciò è sempre bene effettuare dei test comparativi tra diversi valori di NewRatio prima di scegliere la configurazione da associare alla nostra applicazione.

6.2.3 Adempiere: Windows e Linux

In questo paragrafo vengono analizzati i risultati dei test effettuati su Adempiere utilizzando come sistema operativo prima Windows e poi Linux. Nella Figura 6.17 sono presentati i grafici relativi ai dati raccolti.



(a) Tempo d'esecuzione

(b) Integrale del consumo

(c) Consumo medio

(d) Consumo Wh

Figura 6.17: *Consumi di Adempiere su Windows e Linux al variare della memoria massima*

Per prima cosa si nota che anche in questo caso, come per Openbravo, Linux necessita di una memoria maggiore per rendere stabile l'applicazione. Quest'ultimo ha bisogno di 128MB mentre Windows di soli 80MB (-37,5%). Come già spiegato precedentemente, le differenze in questo caso sono da imputare all'architettura differente dei due sistemi operativi. Anche per questo ERP però, l'overhead di memoria è ben più alto di quello che teoricamente si attribuisce alle architetture a 64bit.

A parità di memoria i due sistemi operativi non ottengono mai valori che si discostino più del ±10% sia nei tempi d'esecuzione che nei consumi. In generale si riscontrano dei vantaggi per Windows nei tempi d'esecuzione ma degli svantaggi nei consumi globali.

Lungo tutta la curva dei tempi d'esecuzione e dei consumi, per entrambi i sistemi operativi, non si notano grandi differenze. I valori relativi ai vari tagli di memoria, rispetto a quelli dei tagli minimi, rappresentano sempre una differenza inferiore al ±10%.

La motivazione di questa insensibilità all'aumentare di memoria è da ricercare nel comportamento del GC col valore minimo di heap. Di seguito sono mostrati i dati relativi al funzionamento del GC in Linux con tale valore di memoria:

- GC Total Time: 2.7550967 sec
- GC Time (young): 2.1938117 sec
- Full GC Time: 0.561285 sec
- Total Released Memory: 6386,187 MB
- GC (young) Released Memory: 6377,958 MB

- Full GC Released Memory: 8,253 MB
- Medium Available Heap: 136,550

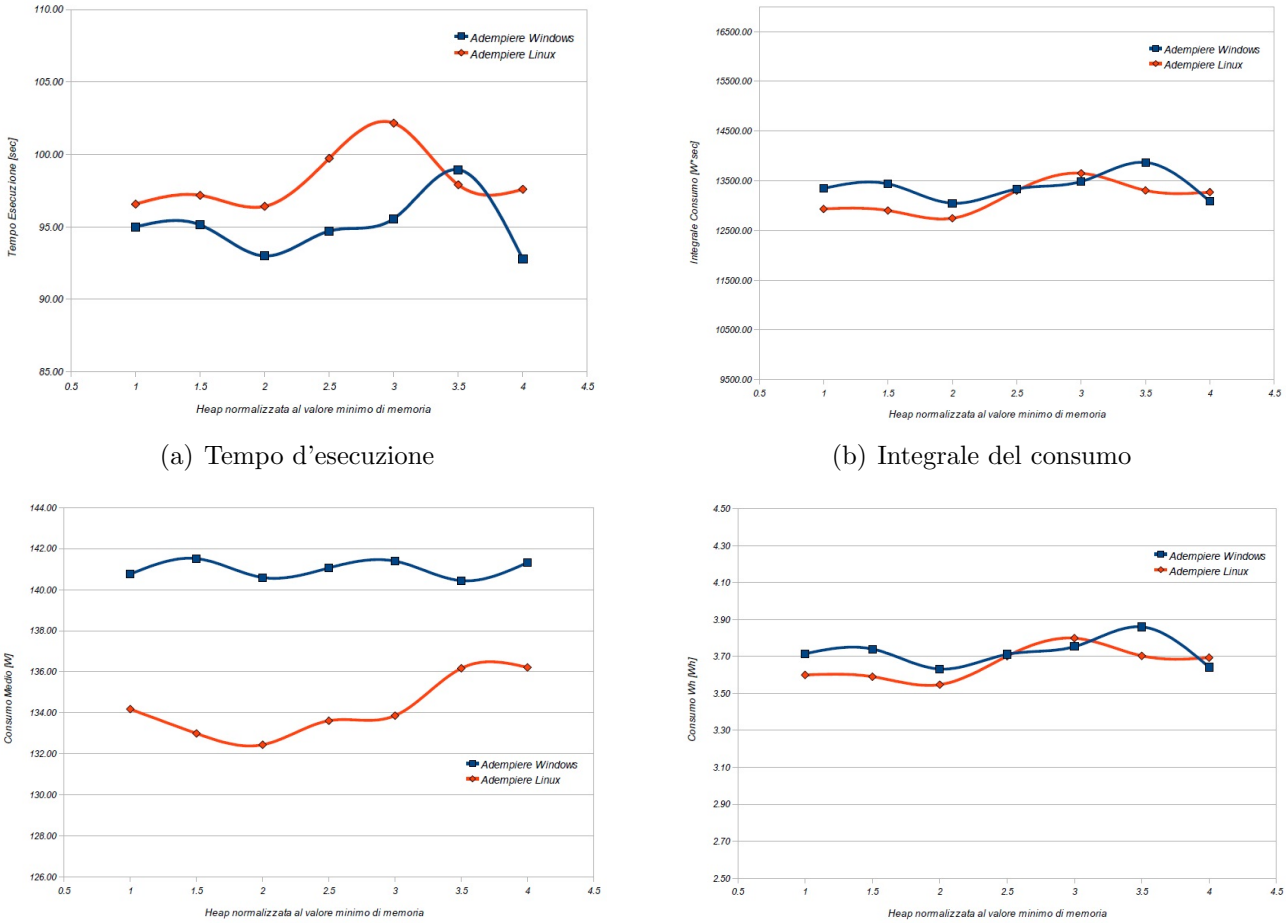
Come si può notare anche con un quantitativo minimo di memoria il GC porta ad avere un tempo totale di pausa molto basso e quindi anche all'aumentare della memoria disponibile non riesce a migliorare le sue prestazioni.
Viene quindi spontaneo chiedersi perché, nonostante il GC stia lavorando ancora in modo efficiente, non sia possibile rendere stabile l'applicazione anche con quantitativi di memoria inferiore. Analizzando il comportamento dell'applicazione con quantitativi massimi di heap inferiori si nota che nella quasi totalità dei casi gli errori si manifestano in due modi:

- pagine visualizzate completamente ma con sezioni mancanti, per esempio con titoli e collegamenti ma senza le caselle di testo dove inserire i dati.
- pagine che visualizzano un errore di timeout non provenienti dall'application server (*Tomcat*) come ci si aspetterebbe, ma generate dall'applicazione stessa.

Queste tipologie di errori porta a ipotizzare che, i problemi con scarsa memoria, non siano da imputare allo spazio insufficiente nella heap ma, all'applicazione che non tollera adeguatamente le pause di garbage collection lunghe. Difatti, come in tutti i test effettuati, nel caso di poca memoria disponibile, le FullGC diventano molto frequenti introducendo molto spesso pause lunghe.

Lo stesso comportamento è osservabile in entrambi i sistemi operativi e analizzando la Figura 6.18 si può notare che, con memoria normalizzata al valore minimo, ottengono risultati simili per andamento.

Si può concludere, perciò, che un'applicazione che in teoria non necessita di un quantitativo di memoria alto, in pratica può averne bisogno soprattutto se pro-



(a) Tempo d'esecuzione

(b) Integrale del consumo

(c) Consumo medio

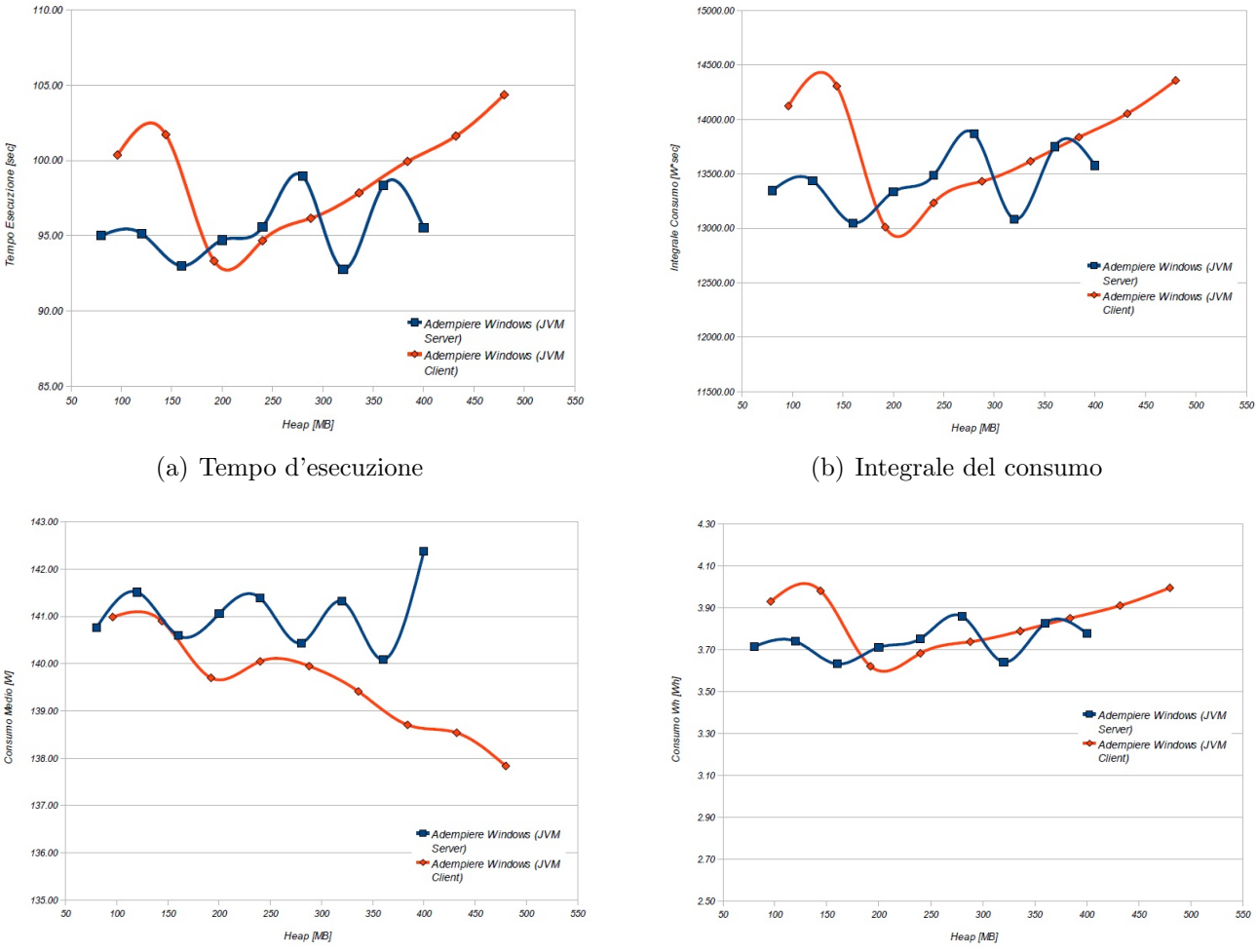
(d) Consumo Wh

Figura 6.18: *Consumi di Adempiere su Windows e Linux al variare della memoria massima (valori normalizzati alla memoria minima possibile)*

grammata in un modo non sufficientemente robusto. Inoltre anche in questo caso la JVM riesce a disaccoppiare efficacemente l'applicazione dallo stack sottostante.

6.2.4 Adempiere: JVM Client e JVM Server

I dati rilevati dai test di confronto tra JVM Client e Server sono illustrati in Figura 6.19.



(a) Tempo d'esecuzione

(b) Integrale del consumo

(c) Consumo medio

(d) Consumo Wh

Figura 6.19: *Consumi di Adempiere su JVM Client e Server al variare della memoria massima*

Come si può notare la memoria minima necessaria all'applicazione, se abbinata alla JVM Client, cresce da 80MB a 96MB (+20%). Probabilmente il motivo di ciò

è che a parità di heap massima, con una configurazione della memoria simile e con una distribuzione degli oggetti non propriamente di tipo client, vengono richiamate più FullGC rispetto alla versione Server. Di conseguenza l'applicazione, non tollerandole, diventa instabile.

Perciò dovendo utilizzare un taglio di memoria dove il GC è già sufficientemente efficiente esso non ha molto margine di miglioramento.

In conclusione anche per la JVM Client le differenze tra i vari tagli di memoria sono poco significativi (massimo 10%). Inoltre, osservando le differenze abbastanza contenute tra i risultati ottenuti dalle due JVM, si può osservare che l'applicazione genera una percentuale di oggetti long lived tale da non avvantaggiare significativamente una o l'altra versione. Lo stesso comportamento è stato possibile notarlo nel caso di Openbravo.

6.3 Valutazione ipotesi di ricerca

Alla luce dei test si può concludere che non tutte le ipotesi di ricerca sono valide. Nella Tabella 6.1 vengono mostrati i risultati ottenuti.

Ipotesi	**Descrizione**	**Validità**
Hyp 1	Aumento heap diminuisce consumo	Confermata in parte
Hyp 2	Configurazione GC influenza consumi	Confermata
Hyp 3	GC consuma meno di gestione esplicita se presenti molti oggetti short-lived	Confermata
Hyp 4	Sistema operativo influenza i consumi	Confermata in parte
Hyp 5	Arch. software a 64 bit consuma più di arch. 32 bit	Confermata

Tabella 6.1: *Ipotesi di ricerca e loro validità*

La prima ipotesi è confermata solo in parte dal momento che non in tutti i casi, aumentando la memoria massima utilizzata dall'applicazione, è possibile ridurre i consumi.
Nei benchmark *GCTest* in versione Java è presente una riduzione delle richieste di energia solo nel caso si utilizzi una JVM versione Server. Difatti, nel caso si opti per la versione Client, i vantaggi derivanti da un numero inferiore di garbage collection sono resi inefficaci dall'overhead causato dalla configurazione di memoria poco efficiente. La particolare configurazione della memoria interna di questa versione difatti non è adatta al carico generato dal benchmark, che presenta una quantità di oggetti long-lived molto alta.
Nel caso della versione C dello stesso benchmark i risultati praticamente non variano con l'aumentare della heap massima. Questo perché il Boehm GC deve essere configurato con un *Free Space Divisor* differente per poter sfruttare tutta la memoria disponibile. Mantenendo lo stesso valore di questo parametro il programma andrà sempre ad utilizzare la stessa quantità di memoria nonostante il suo valore massimo cresca.
Nel caso degli ERP si sono notati dei vantaggi significativi soprattutto nel caso di Openbravo. Nel caso di Adempiere, invece, non è stato possibile porre sotto stress il GC a causa dei problemi di stabilità già descritti nel paragrafo precedente. Va sottolineato però che i vantaggi più consistenti si hanno tra il minimo quantitativo possibile di memoria massima e una volta e mezzo tale quantitativo. Valori più alti di memoria massima portano a vantaggi marginali in termini di consumo.

La seconda ipotesi è confermata dai risultati ottenuti in tutti i test effettuati. Sia nel caso dell'applicazione Java di benchmark che nel caso degli ERP, i consumi registrati con le due versioni della JVM, che possiedono diverse configurazioni del GC, differiscono in modo significativo. In alcuni casi e con alcuni tagli di memoria è avvantaggiata la versione Client mentre in altri quella Server. Si è notato dai

test che la scelta del settaggio energeticamente più efficiente del GC dipende, oltre dall'applicazione, anche dal quantitativo di memoria massima imposto.

La terza ipotesi è pienamente confermata dai benchmark in versione C eseguiti con e senza l'utilizzo del Boehm GC. Si osserva infatti che nel caso si reallochi più del 50% di variabili ad ogni ciclo, generando molti oggetti short-lived, la versione con GC risulta essere energeticamente meno dispendiosa di quella che non lo utilizza. Inoltre si è notato che è valido il discorso inverso con molti oggetti long-lived.

Per verificare la quarta ipotesi sono state eseguite le versioni Java e C del GCTest con carico adattato all'architettura del sistema operativo. In questo modo è stato possibile mettere sullo stesso piano di lavoro Windows e Linux. I risultati dei test confermano tutti che non vi sono differenze significative tra i due sistemi operativi nel caso di Java, ma sono presenti consumi molto diversi nel caso di C. Ciò implica che la scelta del sistema operativo è molto importante se si utilizzano programmi C mentre se si utilizzano programmi Java la JVM permette di disaccoppiare in modo molto efficace lo stack sottostante.

La quinta ipotesi è pienamente verificata nella quasi totalità dei test effettuati. Solo con alcuni tagli di dimensione massima della memoria il sistema operativo Linux a 64 bit riesce ad essere leggermente più efficiente.

Dai risultati riscontrati dei test per la verifica delle ipotesi si possono trarre due conclusioni principali:

- l'adozione e la conseguente scelta della configurazione energeticamente più efficiente del GC, devono tener conto sia della distribuzione degli oggetti fra short e long lived che del taglio di heap massima impostato

- la scelta dello stack sottostante all'applicazione influenza i consumi soprattutto nel caso di architetture software differenti

Capitolo 7

Applicazione a supporto della configurazione della memoria

Come abbiamo visto nei capitoli precedenti, la scelta della configurazione di memoria riveste un ruolo molto importante nella ricerca di una maggior efficienza energetica del software.
In contesti in cui su una singola macchina sono in esecuzione più applicativi, si deve riuscire a trovare la distribuzione della memoria per rendere il sistema il più efficiente possibile dal punto di vista dei consumi.
A questo scopo è stata progettata una metodologia che è supportata da un software sviluppato appositamente.

Prima di poter proseguire col calcolo effettivo della configurazione migliore devono essere svolti alcuni passi preliminari:

1. si verifica la miglior configurazione della memoria interna, per esempio nel caso di Java testando l'applicazione con differenti valori per il NewRatio

2. con opportuni test si ricerca la quantità minima di memoria con la quale l'applicazione risulta stabile

3. si testano i consumi dell'applicazione con diversi tagli di memoria eventualmente settando NewRatio differenti se c'è necessità di farlo. È consigliabile

testare fino ad un taglio pari, se possibile, a 4 volte la memoria minima e cercare di effettuare test su un numero ragionevole di tagli di memoria (es. incrementi di un quarto o un mezzo della memoria minima)

Per il rilevamento dei consumi dell'applicazione si può calcolare il consumo del sistema con la sola applicazione considerata in esecuzione. Il server è bene che sia monitorato per un tempo significativamente lungo per poter ottenere un valore più accurato della media del consumo giornaliero.

Una volta effettuati i test si potranno ottenere i seguenti dati per ogni applicazione (x):

- M_y : quantitativo di memoria y (non tutte le applicazioni sono testate con tutti i quantitativi di memoria possibile).
- $C(x, y)$: consumo dell'applicazione x con un quantitativo di memoria y (in Wh)

Successivamente è necessario definire altri parametri per il calcolo della configurazione migliore:

- P_x : ad ogni applicazione andrà associata una priorità di esecuzione (1-99) che la avvantaggerà nella scelta dei tagli di memoria più efficaci
- M_{max} : memoria massima del sistema

La somma delle P_x ovviamente dovrà dare come risultato 100 per ottenere risultati sensati. Nella definizione del valore di priorità è bene che si tenga in considerazione gli effetti che una scelta poco oculata può avere sul consumo finale del server. Avvantaggiare molto un programma che ottiene miglioramenti molto scarsi con l'aumento di memoria porterà a un inefficienza energetica non trascurabile.

Per comunicare i dati dei parametri e dei consumi al tool, è necessario compilare un file di testo seguendo una formattazione prestabilita. Un esempio di questo file è riportato di seguito:

```
Application Number:
3
Max Memory:
400
Application Priority:
25
40
35
Application:
50
4
100
3
150
2
Application:
25
4
50
3
75
2
Application:
75
4
```

```
150
3
225
2
```

Per ogni applicazione vengono specificati il taglio di memoria e i dati di consumo abbinati in quest'ordine: memoria, consumo, memoria, consumo, ecc. .
Nell'esempio la prima applicazione ha una priorità pari a 25, con 50 MB di memoria consuma 4 Wh, con 100 MB di memoria consuma 3 Wh e con 150 MB di memoria consuma 2 Wh. Così via anche per le altre applicazioni.

Utilizzando queste informazioni il programma sarà in grado tramite ricorsione di calcolare la configurazione ottimale del sistema. La ricerca della soluzione migliore avviene in modo semi-esaustivo, nel senso che, se viene incontrata nel percorso di discesa della ricorsione una configurazione che non può soddisfare il vincolo di memoria massima del sistema, verrà interrotta l'analisi per quel ramo.

Il programma ad ogni iterazione completa valuterà differenti distribuzioni della memoria (vettore M). Ad ogni configurazione assegnerà un valore calcolato tramite la formula:

$$\sum_{x=1}^{n} \frac{C(x, M(x))}{P_x}$$

Dove $M(x)$ indica il valore della memoria per l'applicazione x nella configurazione attualmente considerata.

La configurazione che porta ad un risultato minore, è quella che permette al sistema di avere la miglior efficienza energetica possibile, compatibilmente con le scelte effettuate per le priorità delle applicazioni.

Capitolo 8

Conclusioni e Studi Futuri

Questo lavoro si è posto come obiettivo quello di esplorare l'influenza del *Garbage Collector* sui consumi e sui costi di sviluppo.

Inizialmente è stata svolta un'analisi per comprendere quanto la gestione automatizzata della memoria consente di risparmiare nello sviluppo e nella manutenzione del software. Durante questa fase, è stato analizzato il codice di applicazioni Java di vario tipo, per identificare la quantità di istruzioni per l'allocazione di nuovi oggetti (*new()*). Si è giunti alla conclusione che utilizzando un linguaggio privo di *Garbage Collector*, i costi di sviluppo possono aumentare anche del 7%. Inoltre aumenta la presenza di bug nel codice facendo crescere a sua volta i costi di manutenzione.

D'altro canto il GC può avere effetti negativi; per identificarne l'influenza sui consumi energetici, sono stati presi in considerazione degli applicativi Java sviluppati ad hoc o già esistenti. Questi sono stati testati con diverse configurazioni del garbage collector per poi verificarne gli impatti sui consumi energetici. Gli applicativi considerati sono un benchmark, chiamato *GCTest*, e due ERP. Inoltre è stata creata una versione in linguaggio C dello stesso benchmark.
Per analizzare le differenze di comportamento del GC e l'impatto sui consumi in sistemi operativi differenti, sono stati analizzati i risultati ottenuti con Windows

Server 2003 32bit e Fedora 13 64bit.
Tutti i test sono stati effettuati utilizzando due *Java Virtual Machine* chiamate *Client* e *Server* che possiedono una differenza nella configurazione del Garbage Collector. Questa differenza, più in dettaglio, riguarda la distribuzione della heap utilizzata dalla JVM tra oggetti a vita breve e oggetti a vita lunga. Solitamente in presenza di oggetti di vita lunga, tipici delle applicazioni server, una configurazione più vicina a quella della JVM versione Server dovrebbe essere avvantaggiata.

Dai benchmark di allocazione della lista (*GCTest*), si è notato come le differenti JVM utilizzate, portano ad avere consumi molto diversi tra loro in base alla distribuzione della tipologia di oggetti e alla quantità di memoria massima.
La JVM versione Client rispetto a quella Server consuma fino al 223% in più nel caso di molte variabili a vita breve, mentre fino al 285% in più nel caso le variabili generate siano distribuite equamente tra a vita breve e a vita lunga.
Inoltre dalla memoria minima testata a quella massima è possibile ottenere un risparmio in consumo fino al 74%.
Si è notato quindi come una configurazione errata del garbage collector possa portare a consumi molto superiori.
Successivamente gli stessi benchmark sono stati eseguiti, utilizzando la JVM Server sui due sistemi operativi. A parità di carico e memoria, Fedora comporta un consumo superiore approssimativamente del 30% rispetto a Windows. Questo overhead è dovuto alle maggiori dimensioni degli oggetti nel sistema a 64bit che si attestano a un +19% circa.
Nei test eseguiti, variando le dimensioni dei nodi allocati per permettere un confronto paritario tra i due sistemi operativi, si è notato che il comportamento è del tutto simile al variare della memoria massima. Si può concludere perciò che la JVM disaccoppia in modo eccellente lo stack sottostante all'applicazione.
Nei confronti tra la versione C e la versione Java del *GCTest* la seconda ha fatto

registrare consumi quadrupli rispetto alla prima.
In seguito, la stessa versione C è stata modificata per utilizzare anch'essa un garbage collector, il Bohem Garbage Collector. I test hanno dimostrato come, al variare della tipologia delle variabili, la versione priva di garbage collector risulta essere meno efficiente energeticamente nel caso di pochi oggetti a vita lunga, ma più efficiente nel caso opposto.

Oltre al *GCTest* sono stati testati anche due ERP, Openbravo e Adempiere, i quali hanno dimostrato comportamenti molto diversi tra loro al variare della memoria e della configurazione del garbage collector.
In Openbravo la versione Client della JVM rispetto a quella Server ha fatto registrare, a memoria minima possibile, consumi minori del 25%, mentre a quattro volte la memoria minima consumi maggiori del 13%. Con questo test di evidenzia come non sempre le applicazioni prettamente server siano da associare alla JVM Server.
Le differenze tra Windows e Linux sono dovute principalmente alla diversa architettura che adottano. Linux ha un overhead di memoria del 50% che si ripercuote sui consumi rendendoli fino al 34% più alti di quelli di Windows.
Normalizzando la memoria al valore minimo in modo da eliminare l'overhead introdotto dall'architettura a 64 bit si nota come entrambi i sistemi operativi seguono un andamento del tutto simile; ciò avvalora la tesi secondo la quale la JVM è capace di disaccoppiare l'applicazione dallo stack sottostante.
Nel caso di Adempiere le variazioni dei consumi con tagli di memoria differenti sono estremamente basse. Analizzando il comportamento dell'applicazione è stato individuato che la causa di ciò è derivata da un problema all'interno di essa.
Invece, le basse differenze tra le versioni Client e Server della JVM, sono da attribuire più che altro alla tipologia di oggetti prodotti dall'applicazione. La percentuale di oggetti short lived è tale da porre sugli stessi consumi le due versioni.

In conclusione il lavoro di tesi ha dimostrato sei concetti fondamentali:

- la configurazione del garbage collector influisce sui consumi fino al 285%
- per differenti applicazioni la configurazione energeticamente più efficiente è differente
- la JVM disaccoppia in modo eccellente lo stack sottostante
- il linguaggio con cui è scritta un'applicazione influisce sui consumi
- i sistemi operativi a 64bit, rispetto a quelli a 32bit, introducono un consumo fino al 34% superiore
- un aumento della memoria associata ad un'applicazione non sempre migliora prestazioni e consumi

Per avere ulteriori conferme sulle conclusioni tratte dai test sarebbe bene che venissero testati altri applicativi su diversi sistemi operativi. Un confronto diretto tra sistemi operativi uguali ma a diversa architettura può fornire risultati più precisi sull'overhead introdotto dai 64 bit.
Ancora più interessante sarebbe, modificando la Java Virtual Machine, effettuare un confronto tra gestione della memoria con garbage collector e gestione esplicita, ovvero senza garbage collector. Possibilmente effettuando tale comparazione con applicazioni che generino distribuzioni diverse tra oggetti young e old.

L'applicazione creata, per la ricerca delle configurazioni per i GC che permettano la maggior efficienza energetica globale, costituisce un tool già utile di per se, ma pone anche pone le basi per un possibile sviluppo di un gestore delle JVM. Questo per permettere di modificare dinamicamente le configurazioni dei vari GC reagendo a variazioni significative del carico; per esempio permettendo a un'applicazione sotto stress di utilizzare più memoria, a scapito delle applicazioni

a riposo, in modo da ridurre i consumi globali.

Un'ulteriore evoluzione del gestore potrebbe aggiungere il riconoscimento automatico della situazione di lavoro dei garbage collector; sarebbe così possibile settare in modo del tutto automatizzato le applicazioni sulle configurazioni energeticamente più efficienti.

Bibliografia

[1] A. W. Appel. Garbage collection can be faster then stack allocation. *Information Processing Letters*, 1987.

[2] Alberto Alberio Barbara Brianza. Un'analisi esplorativa sull'ottimizzazione del consumo energetico del software. 2009.

[3] Filippo Bonoli. Una metodologia per ridurre il consumo energetico del software, ed il suo impatto sul total cost of ownership. 2010.

[4] G. Agosta C. Francalanci, E. Capra. Developing energy-efficient software: Enersoft. 2009.

[5] K. P. Vo D. G. Korn. In search of a better malloc. In *USENIX Conference Proceedenigs*, 1985.

[6] D. L. Detlefs. Concurrent garbage collection for c++. *MIT Press*, 1991.

[7] K. S. McKinley E. D. Berger, B.G. Zorn. Composing high-performance memory allocator. In *ACM SIGLPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2001.

[8] K. S. McKinley E. D. Berger, B.G. Zorn. Reconsidering custom memory allocation. In *Conference on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA)*, 2002.

[9] G. Galli. Un approccio alla valutazione dei consumi energetici del software basato sull'analisi statica del codice. 2008.

[10] Stefano Gallazzi Giulia Formenti. A methodology to evaluate empirically software energy consumption and its impact on total cost of ownership. 2008.

[11] Brian Goetz. Java theory and practice: Garbage collection in the hotspot jvm. *developerWorks - IBM*, 2003.

[12] M. Weiser H. J. Boehm. Garbage collection in an uncooperative environment. *Software and Practice Experience*, 1988.

[13] G. Insolvibile. Garbage collection in c. *Linux Journal*, 2003.

[14] L. Steele J. Guy. Multiprocessing compactifying garbage collection. *Communications of the ACM*, 1975.

[15] E. D. Berger M. Hertz. Quantifying the performance of garbage collection vs. explicit memory management. 2005.

[16] P.R. Wilson M. S. Johnstone. The memory fragmentation: solved? In *International Symposium on Memory Management*, 1998.

[17] David A. Solomon Mark E. Russinovich. Microsoft windows internals. 2004.

[18] Sun Microsystem. Memory management in the java hotspot™ virtual machine. 2006.

[19] Geoffrey Phipps. Comparing observed bug and productivity rates for java and c++. 1999.

[20] Simone Piccardi. Amministrare gnu/linux. 2006.

[21] Rafael Lins Richard Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley and Sonsy, 1999.

[22] K. S. McKinley S. M. Blackburn, P. Cheng. Myths and realities: The performance impact of garbage collection. 2004.

[23] C. Li H. Palm T. Brecht, E. Arjomandi. Controlling garbage collection and heap growth to reduce the execution time of java application. 2001.

[24] B.G. Zorn. The measured cost of conservative garbage collection. *Software and Practice Experience*, 1993.

Appendice A

Configurazione di Adempiere e Openbravo

Java Virtual Machine

Il primo passo da effettuare, prima di installare i vari software necessari agli ERP, è installare la Java Virtual Machine che verrà utilizzata per eseguire il web server Tomcat. La versione utilizzata durante i test è la *JDK 1.6u21* della Sun.
In windows è sufficiente scaricare il file eseguibile di installazione corretto (versione 32bit) e lanciarlo. Dopo di che devono essere aggiunte le variabili d'ambiente necessarie alle installazioni successive, raggiungendo l'elenco cliccando col tasto destro su *Risorse del Computer*, scegliendo *Proprietà*, poi il segnalibro *Avanzate* e infine il tasto *Variabili d'ambiente*. Le variabili da aggiungere sono:

```
JAVA_HOME=c:\percorso_corretto\jdk1.6.0_21
JRE_HOME=c:\percorso_corretto\jdk1.6.0_21\jre    (opzionale)
PATH=valori_presenti;c:\percorso_corretto\jdk1.6.0_21\bin;
                     c:\percorso_corretto\jdk1.6.0_21\jre\bin
```

Per linux è consigliabile scaricare la versione, non a pacchetto rpm ma, quella contenente l'archivio per poi estrarla in una cartella a scelta (es. /opt/). Le variabili d'ambiente da aggiungere sono le medesime e lo si può fare per esempio modificando il file *.bashrc* dell'utente includendo le seguenti righe di codice:

```
export JAVA_HOME=/opt/jdk1.6.0_21
export JRE_HOME=/opt/jdk1.6.0_21/jre
export PATH=/opt/jdk1.6.0_21/bin:/opt/jdk1.6.0_21/jre/bin:\$PATH
```

PostgreSQL in Windows

La versione utilizzata è la 8.3.11-1; per installarla basta lanciare il file eseguibile e seguire le istruzioni seguenti:

- impostare come Username e Password *postgres* per entrambi
- selezionare l'installazione di PL/Java
- selezionare di eseguire PostgreSQL come servizio
- selezionare al posto di C il proprio linguaggio e al posto di SQL-ANSI scegliere UTF8
- consentire a PostgreSQL di accettare connessioni remote
- procedere nell'installazione dello stack e installare i driver JDBC

Infine aggiungere alla variabile d'ambiente *PATH* il percorso della cartella *bin* contenuta nel percorso di installazione scelto.

PostgreSQL in Linux

Per linux è stata utilizzata la versione disponibile nei repository; per installarla accedere alla shell come root e lanciare il comando seguente:

```
yum install postgresql postgresql-devel postgresql-server
        postgresql-contrib postgresql-jdbc uuid-devel
        uuid-pgsql pgadmin3
```

Una volta terminata l'installazione è necessario inizializzare il data base eseguendo, sempre da root, il comando:

```
service postgresql initdb
```

Dopo di che è necessario modificare il contenuto del file *pg_hba.conf* inserendo *trust* di fianco agli indirizzi specificati. Ovviamente non è una configurazione sicura ma dato che il nostro sistema non sarà mai collegato ad internet è un'opzione ragionevole per semplificarne la configurazione.
Una volta terminata l'installazione sarà possibile lanciare, riavviare e fermare il servizio PostgreSQL coi comandi:

```
service postgresql start
service postgresql restart
service postgresql stop
```

Adempiere 3.6.0 in Windows

Estrarre il contenuto del file compresso in una directory priva di spazi (es. c:/adempiere).
Aggiungere alla variabile d'ambiente PATH il percorso della cartella *bin* di PostgreSQL e creare la variabile ADEMPIERE_HOME settandola alla directory in cui è stato estratto Adempiere.
Creare un nuovo utente (*login roles*) con username e password *adempiere* attribuendogli tutti i diritti da super user.
Creare un database sempre di nome *Adempiere* settando come owner il ruolo appena creato.
Questi ultimi due passi possono essere fatti in modo molto semplice tramite l'interfaccia grafica fornita da *PGAdminIII*.

Una volta preparato il database si può iniziare con l'installazione eseguendo da prompt dei comandi il file *RUN_setup.bat* contenuto nella cartella di Adempiere. Nella finestra che si aprirà sostituire:

- il tipo di DB da OracleXE a PostgreSQL
- il nome del DB da xe a adempiere
- la system password con quella inserita nell'installazione di postgres (*postgres*)
- il db server da your_computer_name a localhost
- Application Server Web Port con 8080 e SSL con 8443

In seguito cliccare su *Test* e poi, se è tutto corretto, su *Save* per iniziare il deployment dell'applicazione.
Una volta terminato il deployment sarà necessario ricreare il database se è stato cancellato e lanciare il comando *RUN_ImportAdempiere.bat* contenuto nella directory *utils* di adempiere.
Per i passi successivi è necessario lanciare l'application server tramite il comando *RUN_Server2.bat* dalla directory *utils* di adempiere.
Per verificare il corretto funzionamento del server accedere col browser a *http://ip_server:8080/admin/* o a *http://ip_server:8080/adempiere/*. Prima di utilizzare l'applicazione per i test devono essere eseguiti altri step per configurare l'ERP:

- eseguire da prompt il comando *RUN_adempiere.bat*
- settare cliccando sulla riga server i dati di connessione al server
- inserire come username *SuperUser* e come password *System*
- nella schermata successiva selezionare *System Administrator* nella prima casella e *System* in quella sotto
- dal menu ad albero scegliere *System Admin* poi *Client Rules* e infine *Initial Client Setup*
- chiamare il nuovo client *greenIT* e impostare il resto come in Figura A.1

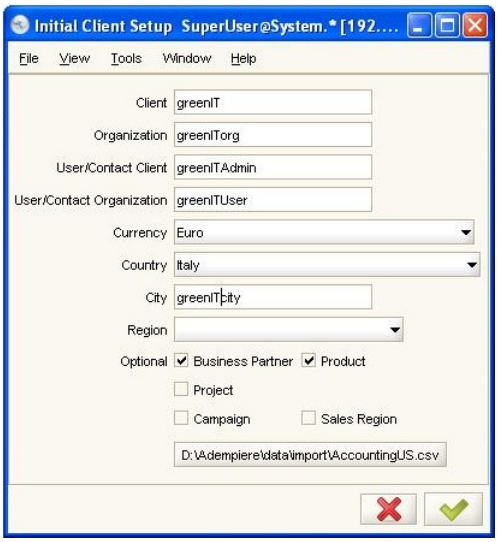


Figura A.1: *Pagina per la configurazione del client di Adempiere*

- selezionare il cvs AccountingIT.csv e avviare la creazione del client

D'ora in poi sarà possibile loggarsi nell'ERP tramite l'interfaccia via browser utilizzando come username e password *greenITAdmin*.

Adempiere 3.6.0 in Linux

La procedura è la medesima di windows con due sole differenze:

- non è necessario modificare la variabile d'ambiente PATH ma basta aggiungere la ADEMPIERE_HOME
- i file da lanciare tramite console sono quelli con estensione *.sh* e non *.bat*.

Ant 1.8.1

Ant viene utilizzato da Openbravo per compilare ed effettuare il deployment l'applicazione. La procedura di installazione è la medesima per Windows e Linux. Dopo aver estratto il contenuto del file compresso in una directory senza spazi (es. c:/ant, /opt/ant) settare le variabili d'ambiente:

- ANT_HOME con il percorso alla directory dove è stato estratto Ant
- ANT_OPTS = -Xmx1024M -XX:MaxPermSize=128M
- aggiungere a PATH la cartella *bin* contenuta nella directory di Ant

Apache-Tomcat 6.0.26 su Windows

Al contrario di Adempiere, Openbravo non ha incluso un application server (*Tomcat*) e per questo motivo si deve appoggiare ad un'installazione esterna.
Per installarlo basta lanciare l'eseguibile e seguire la procedura standard inserendo però come porta la 8091 se si desidera poterlo mantenere attivo insieme al server di Adempiere.
Per lanciare e stoppare l'application server utilizzare l'interfaccia del programma *Tomcat Monitor*.

Apache-Tomcat 6.0.26 su Linux

Per installare Tomcat su Linux è necessario estrarre il contenuto del file compresso e modificare il file *tomcat-users.xml*, contenuto nella cartella *conf*, aggiungendo i ruoli *admin* e *manager* e attribuendoli all'utente con username e password *admin*.
Creare le seguenti variabili d'ambiente aggiungendo i seguenti comandi nel file *.bashrc* o eseguendoli direttamente nella shell:

```
export TOMCAT_HOME=/opt/tomcat
export CATALINA_HOME=/opt/tomcat
export CATALINA_BASE=/opt/tomcat
export CATALINA_OPTS="-server -Xms128M -Xmx512M -XX:MaxPermSize=256M"
```

Per lanciare e stoppare Tomcat utilizzare i comandi *startup* e *shutdown* contenuti nella directory *bin*.

Openbravo 2.50mp19 su Windows

Per l'installazione di Openbravo è necessario aggiungere alla variabile d'ambiente PATH il percorso alla cartella *bin* di Tomcat e creare le seguenti:

- CATALINA_HOME settandola col percorso della cartella principale di Tomcat
- CATALINA_BASE settandola come la precedente
- CATALINA_OPTS impostandola uguale a *-Xms128M -Xmx512M -XX:MaxPermSize=256M*

Dopo di che estrarre dal file compresso di Openbravo il suo contenuto e tramite prompt dei comandi posizionarsi nella cartella principale.
Eseguire il comando *ant setup* e in seguito spostarsi nella sottocartella *config* e eseguire *setup-properties-windows.exe*.
Si aprirà un'interfaccia grafica per impostare l'installazione, per completare la procedura seguire le istruzioni seguenti:

- inserire i percorsi corretti quando richiesti
- scegliere PostgreSQL come DBMS
- inserire come utente e password la parola *postgres*
- inserire come nome del database *openbravo* e come utente e password *tad*

Una volta terminata la configurazione digitare il comando *ant install.source* dalla cartella principale di Openbravo.
Terminata l'installazione digitare il comando *vacuumdb -f -z -h localhost -d openbravo -U tad* per pulire il database.
Accedere col browser a *http://Openbravo_server_ip:8091/openbravo/* per verificare il corretto funzionamento del server.

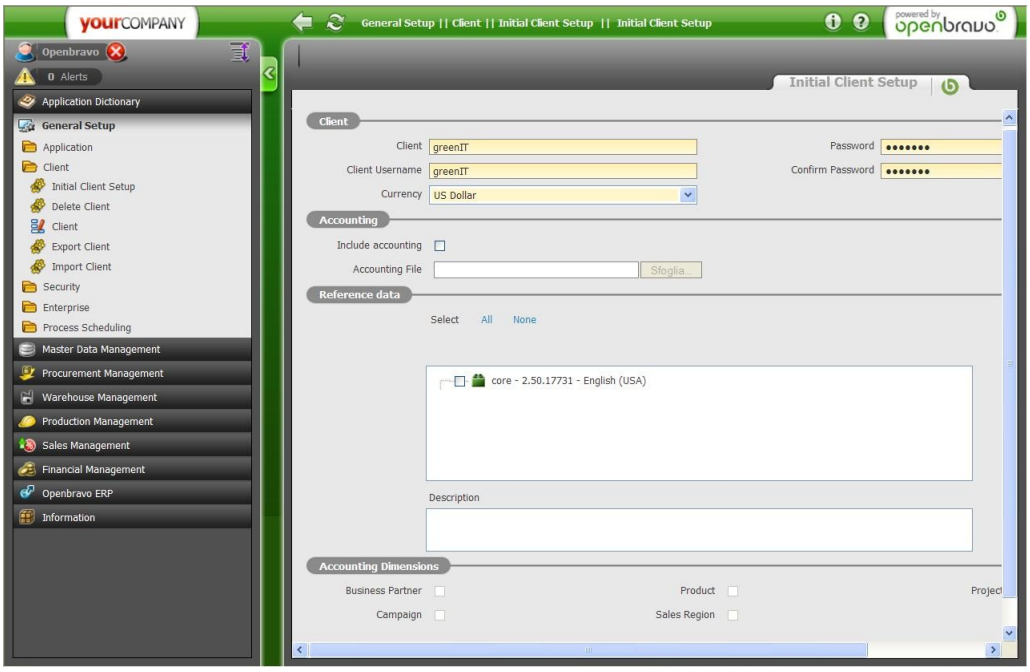


Figura A.2: *Pagina per la configurazione del client per Openbravo*

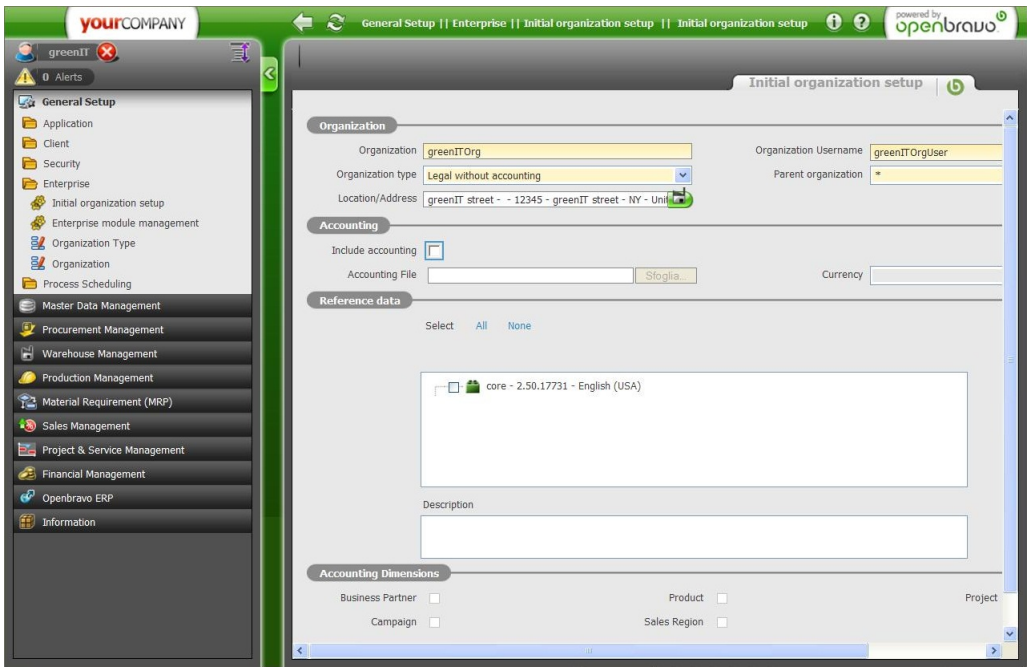


Figura A.3: *Pagina per la configurazione dell'organizzazione per Openbravo*

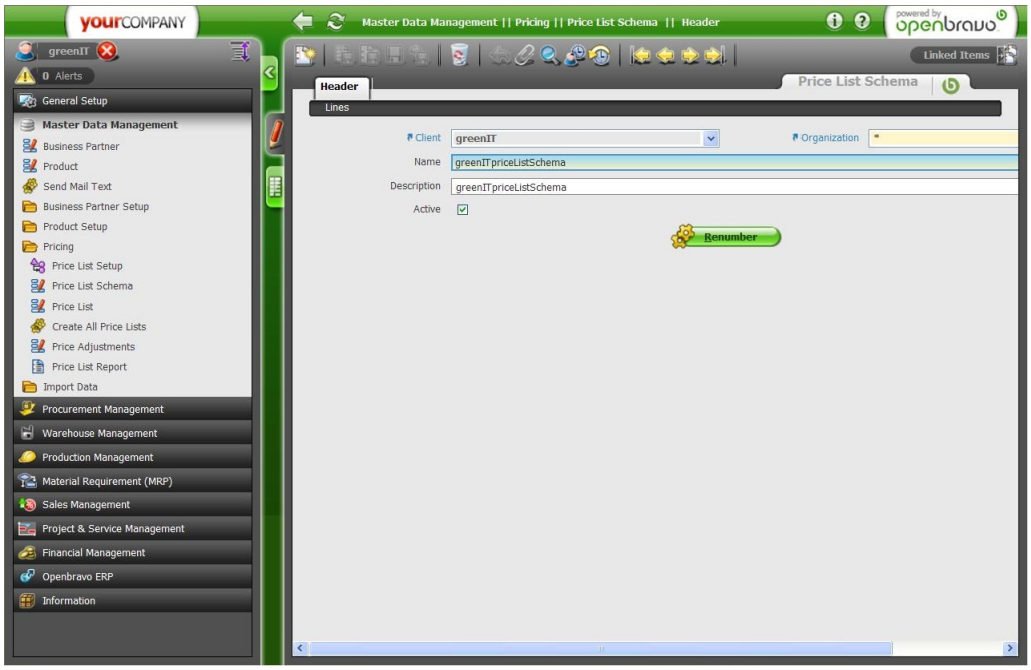


Figura A.4: *Pagina per la configurazione del price list schema per Openbravo*

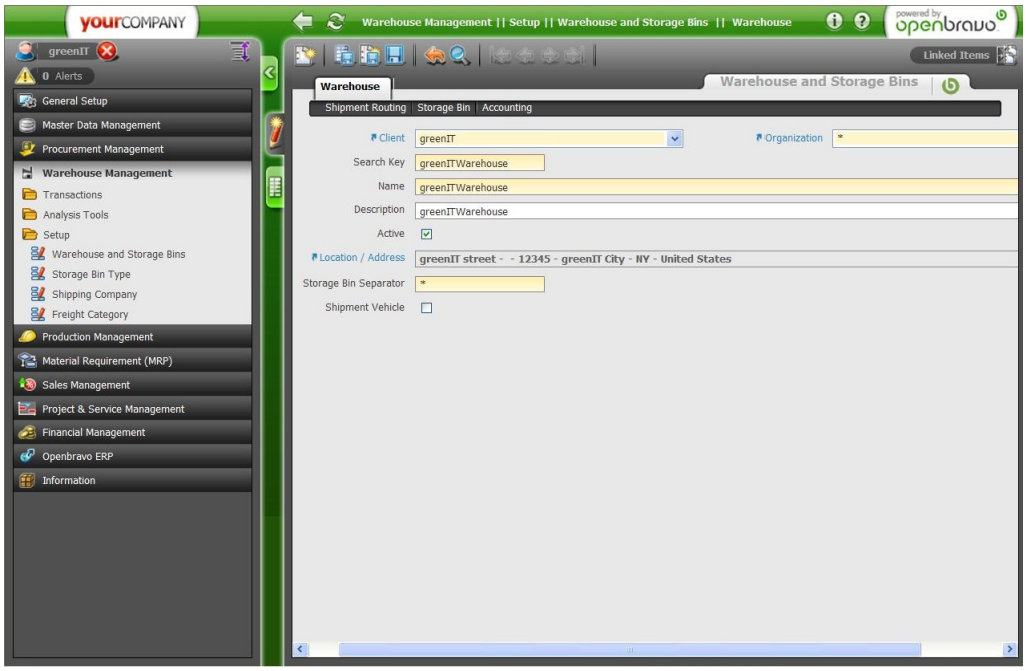


Figura A.5: *Pagina per la configurazione del magazzino per Openbravo*

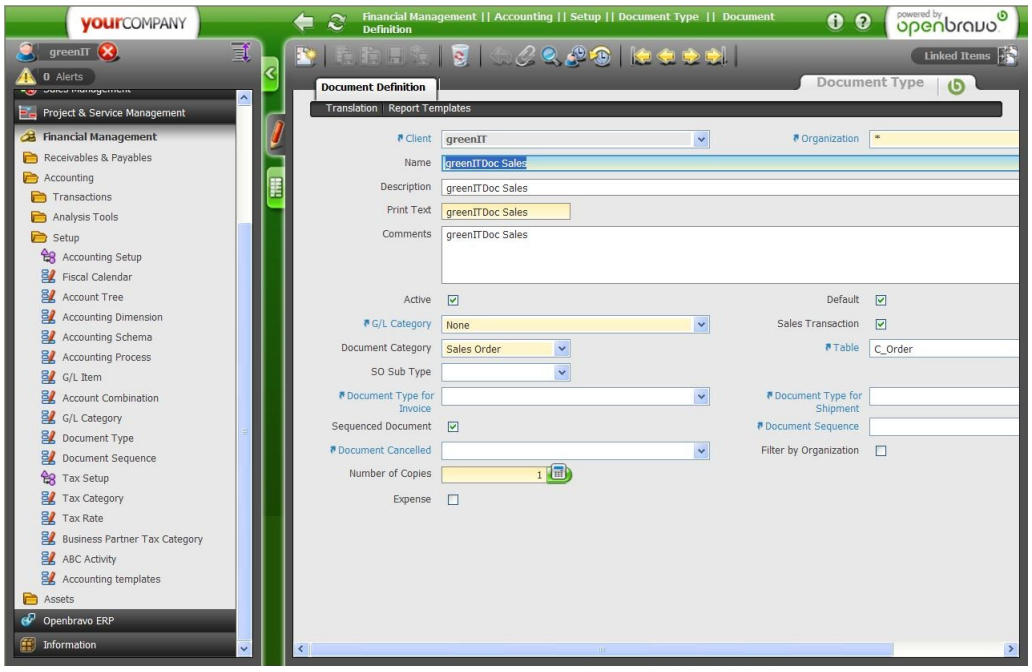


Figura A.6: *Pagina per la configurazione del purchase document type per Openbravo*

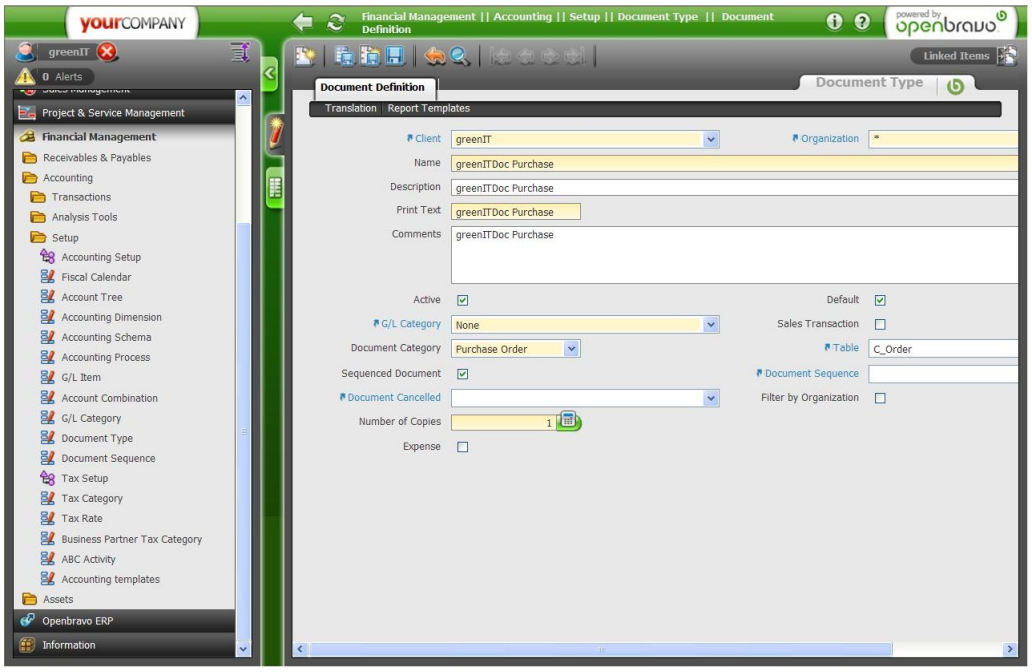


Figura A.7: *Pagina per la configurazione del sales document type per Openbravo*

Per poter utilizzare Openbravo nei test dovrà essere configurato, per farlo seguire le indicazioni di seguito aiutandosi con le Figure A.2, A.3, A.4, A.5, A.6 e A.7:

- loggarsi come *Openbravo* con password *openbravo*
- creare il client inserendo nelle prime quattro caselle di testo sempre *greenIT*
- creare l'organization come legal without accounting
- creare un price list schema
- creare un warehouse
- creare un document type per purchase e settarlo come default
- creare un document type per sales e settarlo come default

Openbravo 2.50mp19 su Linux

Il procedimento è identico a windows tranne che il file da eseguire non è *setup-properties-windows.exe* ma *setup-properties-linux.bin*.

Appendice B

Collegamenti delle schede di misurazione

Entrambe le schede di misurazione devono essere alimentate perciò vanno collegate all'alimentatore. Il polo positivo dell'alimentatore va collegato in parallelo sia al pin di alimentazione contrassegnato col simbolo + della prima scheda che a quello della seconda. Il polo di massa invece ai pins contrassegnati col simbolo O. La scheda di misurazione del flusso di corrente totale deve essere inserita in serie tra la presa elettrica e la spina dell'alimentatore del computer. Per fare ciò basta collegare la spina della scheda alla rete elettrica e la spina del computer alla ciabatta a due prese connessa alla scheda.
La seconda scheda deve essere collegata in serie ai vari cavi di alimentazione dei componenti connettendo le uscite dell'alimentatore di hard disk, scheda madre e processore a quest'ultima. Dopo di che andranno collegati con altri cavi dello stesso tipo la scheda di misurazione con l'hard disk, la scheda madre e la CPU.
Nella Tabella B.1 vengono mostrati i collegamenti da effettuare tra le schede di misurazione, quella di acquisizione e l'alimentatore.

Sch. Acquisizione	Sch. Mis. Totale	Sch. Mis. Componenti	Alimentatore
15 (AI0)		8 (CPU 12V)	
16 (AI8)	Out		
17 (AI1)		7 (HD 12V)	
19 (AI2)		6 (HD 5V)	
21 (AI3)		1 (ATX 12V)	
24 (AI4)		2 (ATX 5V)	
26 (AI5)		3 (ATX 3.3V)	
28 (AIGND)			GND
29 (AI6)		5 (ATX -12V)	
31 (AI7)		4 (HD 5V stand-by)	
		+	5V
		O	GND
	+		5V
	O		GND

Tabella B.1: *Tabella dei collegamenti tra le schede di misurazione, acquisizione e l'alimentatore*

Appendice C

Implementazione: packages e classi

In questa appendice si vuole mostrare quali sono le classi create durante lo sviluppo del software utilizzato nella tesi.

Il primo package che sarà illustrato è quello denominato *ERP*, al suo interno sono presenti due sole classi: *Adempiere.java* e *Openbravo.java*. Entrambe hanno lo scopo di creare gli script da eseguire tramite *iMacros* in base ai parametri specificati nel costruttore. Tramite una procedura lanciano lo script creato sfruttando le funzionalità fornite dalla *jawin.dll*. La procedura, una volta terminato il test, ritorna un valore che indica se esso ha avuto successo o meno. Ovviamente ogni classe genera un test per il relativo ERP.

Il package *GCTest* contiene tutte le classi relative all'applicativo GCTest e GCHalfTest:

- *ListNode.java*: rappresenta la classe che definisce il tipo di ogni nodo della lista.
- *GCTest.java*: rappresenta la classe eseguibile contenente il codice del test di allocazione e deallocazione dell'intera lista. Utilizza la classe *ListNode* per creare i nodi della lista.

- *GCHalfTest.java*: rappresenta la classe eseguibile contenente il codice del test di allocazione e deallocazione di metà lista. Utilizza la classe *ListNode* per creare i nodi della lista.

In questo package potrebbe essere inclusa anche l'applicazione GCTest scritta in linguaggio C.

Nel package *ClientServer* sono incluse sette classi i cui nomi e funzionalità sono elencate di seguito:

- *Client.java*: rappresenta il vero e proprio corpo del Workload Manager client. Utilizza le classi contenute nei due package precedenti per lanciare i relativi test. Utilizza anche la classe *ClientThread* per ricreare un suo clone dopo ogni esecuzione in modo da rimanere sempre attivo anche in caso di errori.

- *ClientThread.java*: estende la classe *Thread* e ha il compito di creare una nuova classe *Client* ogni volta che il metodo *run()* viene richiamato.

- *ClientGCAnalysis.java*: è una versione della classe *Client* che permette di analizzare durante i test sugli ERP l'output generato dal GC relativo alla JVM di Tomcat.

- *GCThread.java*: estende la classe *Thread* e viene utilizzata da *ClientGCAnalysis* per calcolare i risultati aggregati dell'output proveniente dal GC durante i test sugli ERP.

- *ServerCore.java*: rappresenta la logica del Workload Manager server e utilizza i thread di tipo *ServerThread* per gestire le comunicazioni con i diversi client. Ogni thread è associato a un client. Inoltre utilizza l'interfaccia grafica implementata nella classe *ServerGUI*.

- *ServerThread.java*: estende la classe thread ed è utilizzata da *ServerCore* per la comunicazione coi Workload Manager client.

- *ServerGUI.java*: classe che implementa l'interfaccia grafica utilizzata dalla classe *ServerCore*.

L'ultimo package, *OthersTools*, è quello relativo ai vari tools che sono stati sviluppati durante la tesi per aiutare nella comprensione del funzionamento del GC, per velocizzare le procedure di analisi dei dati energetici e per comprendere le migliori configurazioni di memoria. Le classi contenute in questo package sono tutte eseguibili e sono:

- *ParseGCInfoFile.java*: analizza tutti i file di log contenenti l'output generato dal garbage collector durante l'esecuzione dei vari GCTest in java. Restituisce in un unico file importabile in excel i valori relativi a durata del test, tempo totale utilizzato dal GC, tempo utilizzato dal GC per l'analisi dell'area di memoria young, tempo utilizzato dal GC per le FullGC, memoria liberata totale, memoria liberata con garbage collection sull'area young e memoria liberata con le FullGC.

- *ParseGCInfoFileSingle.java*: analizza il file di log specificato in ingresso contenente l'output generato dal garbage collector per restituire gli stessi valori della classe precedente.

- *UnifyEnergyFiles.java*: specificando in ingresso il percorso di una cartella restituisce in un unico file excel il contenuto di tutti i file, contenuti nelle sottocartelle presenti, relativi ai dati energetici dei test.

- *MaxEfficency.java*: impostando tramite un file specifico le relazioni tra memoria, consumo e priorità per ogni applicazione presente su un server, calcola in modo ricorsivo la migliore configurazione per ogni applicazione per ottenere il miglior rapporto efficienza-consumi.