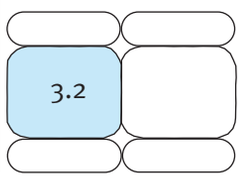
GREEN SOFTWARE
ANCHE LE APPLICAZIONI CONSUMANO ENERGIA

Giovanni Agosta
Eugenio Capra

Ormai da qualche anno si parla diffusamente dell'efficienza energetica dell'IT, ma raramente si affronta il problema del ruolo rivestito dal software nel determinare il consumo energetico dell'IT. È infatti il software a "guidare" il funzionamento dell'hardware e quindi ad essere il primo responsabile del consumo. L'articolo presenta i risultati di alcune ricerche sperimentali che mostrano l'impatto del software sui consumi e propone una panoramica sulle linee di ricerca attualmente in corso per ottimizzarne l'efficienza.



1. INTRODUZIONE AL GREEN SOFTWARE

Il *Green IT*, che nella sua accezione più propria è la disciplina che studia l'efficienza energetica dell'IT, è ormai al centro dell'attenzione a livello sia accademico che industriale da alcuni anni. Nell'ambito del *Green IT* è possibile distinguere diversi campi di azione, tra i quali i più importanti sono:

- Postazioni di lavoro;
- *Data center*;
- *Green hardware*;
- *Green software*;

Il consumo energetico dell'IT distribuito (principalmente PC, *monitor*, *laptop*, stampanti) raccoglie sempre più attenzione in quanto il suo campo di applicazione è molto vasto, poiché la maggior parte delle aziende è ormai informatizzata o comunque dispone di postazioni di lavoro informatizzate. Tuttavia, le leve principali di ottimizzazione dell'efficienza energetica dell'IT distribuito non sono informatiche, ma di tipo gestionale: politiche di acquisto, politiche comportamentali, politiche di gestione e assegnazione delle risorse informatiche in base al reale fabbisogno.

L'efficienza energetica dei *data center* è un tema di ricerca e innovazione su cui diverse aziende e centri di ricerca stanno ormai lavorando da anni, [5, 7, 8, 15]. Un *data center* di medie dimensioni può consumare anche 300 kW, l'equivalente di 100 appartamenti, mentre *un data center* di una grande banca o di un operatore Telecom può arrivare a consumare diversi MW. Le riduzioni di consumo e di costo ottenibili sono quindi rilevanti, anche se riguardano un numero relativamente ristretto di enti e aziende (in Italia si contano circa 3.000 *data center* con più di 5 *rack*). La maggior parte dell'energia assorbita da un *data center* viene dissipata dai componenti infrastrutturali [9, 2] come gli impianti di condizionamento, gli UPS (*Uninterruptible Power Supply*) e i sistemi di distribuzione dell'energia. Un indicatore comunemente utilizzato per misurare l'efficienza energetica di un *data center* è il PUE (*Power Usage Effectiveness*), calcolato come la potenza elettrica entrante nel *data center* divisa per la potenza elettrica che effettivamente arriva ai macchinari IT. Benché al-

cuni *data center* di moderna concezione abbiano PUE molto bassi (per esempio, Google dichiara un PUE di 1,19), la maggior parte dei *data center* esistenti presenta PUE ben più elevati, anche superiori a 2,5. Questo dato fa capire come gli interventi più efficaci siano da effettuare a livello infrastrutturale, ad esempio ripensando il layout del *data center*, ottimizzando il condizionamento, implementando impianti di *free cooling* e sostituendo gli UPS. La seconda leva più efficace è la virtualizzazione, che permette di ridurre i consumi fissi dovuti all'infrastruttura hardware.

Il passo successivo per ridurre il consumo di energia dell'IT è quello di aumentare l'efficienza energetica dell'hardware, più specificatamente dei *server* e dei loro processori. Le grandi case produttrici di hardware svolgono da anni ricerche in questo senso e, in effetti, l'efficienza energetica dell'IT è molto aumentata, in parte spinta dalla diffusione dei dispositivi mobili e dalla necessità di aumentare l'autonomia delle batterie.

Come si può osservare dai dati riportati nella figura 1, considerando tre diverse generazioni di processori, mentre la frequenza di *clock* è aumentata di un fattore 72 la potenza assorbita è aumentata solamente di un fattore 12, con un conseguente aumento dell'efficienza di un fattore 6. Anche i *benchmark* TPC-C confermano un andamento simile. Le architetture *multi-core* hanno sicuramente contribuito a questo aumento dell'efficienza. L'efficienza energetica dei *mainframe*, espressa come Milioni di Istruzioni al Secondo per Watt (MIPS/W) negli ultimi 30 anni è cresciuta di un fattore 28 [1], molto più rispetto all'aumento di efficienza registrato da altri settori, come la produzione di automobili o dell'acciaio.

Benché i dispositivi hardware siano fisicamente responsabili del consumo di energia, che viene dissipata sotto forma di calore, il responsabile primo di tale consumo è il software, che "guida" l'hardware e determina quali e quante operazioni elementari devono essere eseguite. Il "*green software*" è la disciplina che studia le modalità secondo cui il software influisce sui consumi energetici dell'IT e come ottimizzarle.

Per comprendere meglio l'ambito di applicazione del *green software* può essere utile fare un paragone con il mondo automobilistico. Se si vuole consumare poca benzina per andare in auto da Milano a Torino la prima cosa da fare è quella di scegliere una macchina che percorra tanti km con un litro di benzina. Nel mondo informatico questo equivale ad utilizzare hardware efficiente, cioè in grado di effettuare tante transazioni elementari o operazioni *floating point* (flop) con 1 Wh di energia. Ci sono però tante altre soluzioni che si possono adottare per consumare meno benzina. Innanzi tutto si può cercare di viaggiare a pieno carico. Per esempio se si è in 9 persone si può utilizzare un pulmino an-

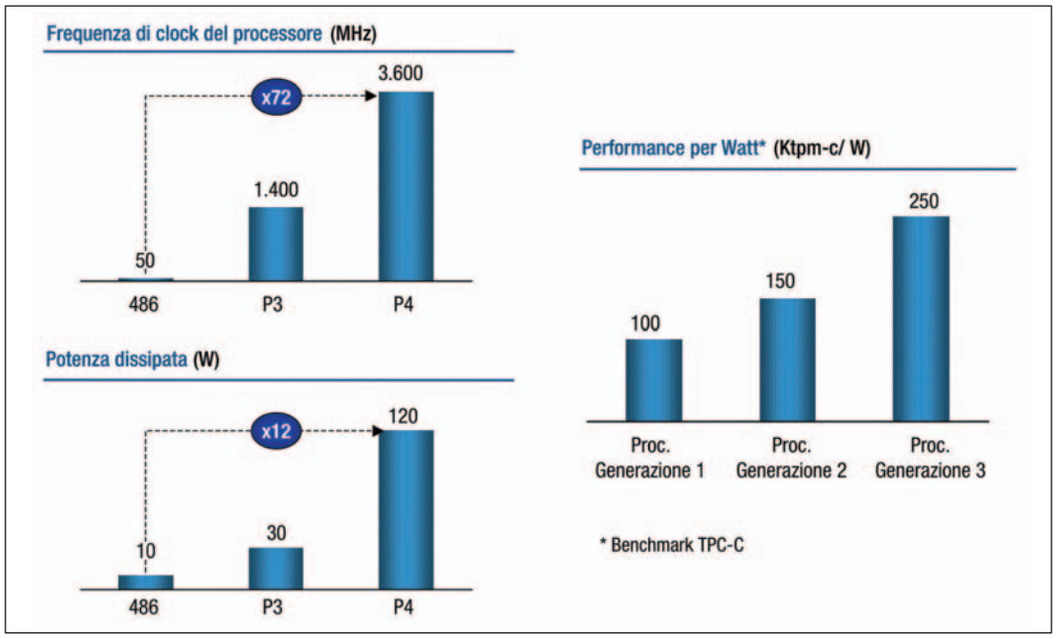


FIGURA 1
Dati sull'efficienza energetica dell'hardware negli ultimi anni (Fonte: Wikipedia, Intel 2002)

ziché 2 automobili. Questo equivale a bilanciare i carichi di lavoro e a virtualizzare i sistemi, pratiche che rientrano nell'ambito del *green data center*. Dopodiché si può viaggiare alla velocità che minimizza il consumo di carburante, che solitamente non coincide con la massima velocità raggiungibile dal veicolo. Infine, si può scegliere la strada che minimizza il numero di km percorsi. Queste due ultime tipologie di azioni corrispondono nel mondo informatico al *green software*. Un'applicazione può essere valutata in base alla sua efficienza energetica, oltre che in base agli altri classici parametri di merito, come il tempo di risposta. Inoltre, un'applicazione scritta "bene" permetterà di soddisfare i requisiti funzionali tramite il minor numero possibile di operazioni elementari e quindi di energia.
Mentre diverse ricerche sono state svolte sull'hardware, sui sistemi *embedded* [4] e sui *data center*, il tema dell'efficienza energetica del software applicativo è relativamente nuovo e inesplorato. Il ciclo di sviluppo del software e le relative metodologie non prendono praticamente mai in considerazione l'efficienza energetica come obiettivo. D'altro canto, è la stessa crescente disponibilità di hardware efficiente e a basso costo che induce i programmatori a non occuparsi dell'efficienza energetica del codice. La letteratura attuale sull'ingegneria del software non fornisce neppure metriche per misurare l'efficienza energetica del software. Neppure le 50 metriche di qualità del software definite dall'*International Standards Organization* (ISO/IEC TR 9126:2003 and TR 25000:2005) includono l'efficienza energetica.
La maggior parte degli IT manager sembra credere che il software di per sé abbia un impatto molto limitato sui consumi, specialmente nell'ambito dei sistemi transazionali classici, come i software bancari o gli *Enterprise Resoruce Planning* (ERP) [6]. È generalmente accettato che i sistemi operativi influenzino le prestazioni e i consumi di energia [10], ma non le applicazioni. Queste convinzioni raramente sono supportate da un approccio scientifico e da misure sperimentali.
In realtà il potenziale di risparmio ottenibile dal *green software* è elevato. Infatti, come accennato sopra, nella maggior parte dei *data center* la percentuale di energia entrante, effettivamente utilizzata per il calcolo, è molto bassa a causa delle inefficienze dei diversi livelli infrastrutturali. Questi dati possono essere interpretati al contrario: per ogni Watt di energia utilizzato per il calcolo sono necessari almeno 5 o più Watt totali. Quindi i risparmi ottenuti a livello software vengono amplificati da tutti i livelli infrastrutturali soprastanti: se vengono eseguite meno operazioni elementari sarà prodotto meno calore, quindi sarà necessaria meno energia per il condizionamento, e così via.
Questo articolo si focalizzerà sul *green software*, dapprima discutendo alcuni risultati sperimentali che mostrano come il consumo energetico del software sia effettivamente significativo, e in seguito proponendo alcune linee di ricerche sul tema, attualmente in corso di esecuzione presso il Dipartimento di Elettronica e Informazione del Politecnico di Milano.

2. IL CONSUMO ENERGETICO DEL SOFTWARE È SIGNIFICATIVO

2.1. Il sistema per la misura sperimentale del consumo

Per poter affrontare in modo scientifico il problema del consumo energetico indotto dal software è opportuno innanzi tutto analizzare qualche dato sperimentale. La figura 2 illustra l'architettura del sistema adottato per misurare empiricamente il consumo di energia. Il sistema comprende un kit hardware per misurare l'energia assorbita dal server basato su pinze amperometriche e uno strumento software per generare carichi di lavoro di *benchmark* per diverse categorie di applicazioni. Per avere misure significative rispetto alle realtà aziendali, si è scelto di confrontare applicazioni di tipo *Management Information Systems* (MIS) con caratteristiche funzionali e parametri confrontabili anziché utilizzare *benchmark* più diffusi in letteratura come gli SPEC o i TPC.
In particolare sono stati confrontati:
- 2 sistemi *Enterprise Resource Planning* (ERP): Adempiere e Openbravo;
- 2 sistemi *Customer Relationship Management* (CRM): SugarCRM e vTiger;
- 4 *Database Management Systems* (DBMS): MySQL, PostgreSQL, Ingres e Oracle.

Tutte le misure sono state effettuate sulla stessa piattaforma hardware e le applicazioni sono state configurate in modo tale da essere il più possibili confrontabili. Per esempio, i DBMS sono stati configurati con la stessa dimensio-

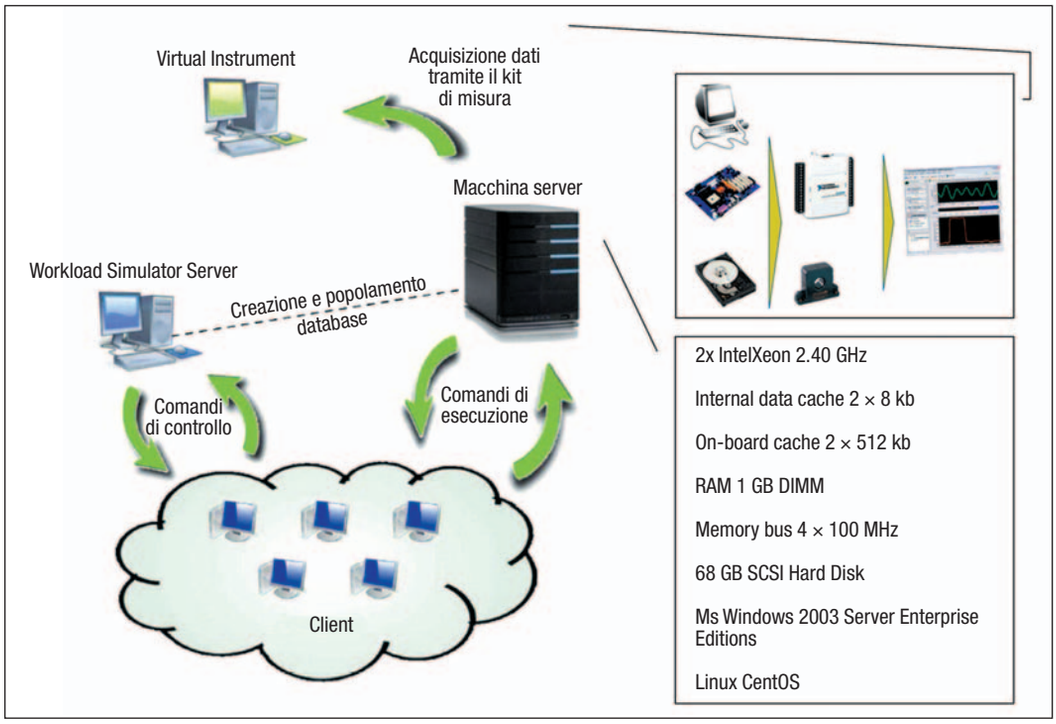


FIGURA 2
Architettura sperimentale per misurare il consumo energetico indotto dal software

ne di pagina, blocco e *cache* e con gli stessi parametri relativi al *buffering*, alla memoria totale di sistema e al numero di connessioni (si veda [3] per ulteriori dettagli).
Le applicazioni vengono eseguite sul server, che è la macchina di cui si misura il consumo di energia. Per ogni applicazione sono stati realizzati diversi esperimenti con un numero di client variabile da 1 a 10. È stato sviluppato un *tool* Java, chiamato *Workload Simulator*, in grado di simulare per ciascuna applicazione un flusso di operazioni e di eseguirlo per un certo numero di volte simulando una certa quantità di utenti simultanei e generando in questo modo un carico di lavoro di *benchmark*. Per gli ERP i carichi di lavoro considerati sono stati l'inserimento di un nuovo *business partner*, di un nuovo prodotto e di un nuovo ordine. Per i CRM sono stati simulati la creazione di un nuovo *account* e di una nuova campagna. Infine, per i DBMS è stata implementata una versione *ad-hoc* del *benchmark* TPC-C, che è comunemente utilizzato per confrontare le *performance* di sistemi transazionali.
Il segnale analogico acquisito dalle pinze amperometriche viene processato da un *Data Acquisition Board* (DAQ) NI USB-6210, che è collegato via USB con un altro PC, la *Virtual Instrument machine*. L'analisi dei dati è effettuata tramite un *tool software* sviluppato tramite LabVIEW (www.ni.com/labview/), che acquisisce e campiona i valori della potenza assorbita ogni 4 microsecondi (quindi con una frequenza di campionamento di 250 MHz) e infine calcola il valore totale dell'energia consumata interpolando i valori della potenza. Per non influenzare la misura sia il *Workload Simulator* che i software per l'acquisizione delle misure risiedevano su macchine diverse dal server.

2.2. Alcuni risultati sperimentali

Le figure 3, 4 e 5 mostrano il valore della potenza consumata rispetto al tempo per le tre categorie di applicazioni considerate eseguite su sistema Windows (grafici **A**) e Linux (grafici **B**). La linea dritta in ciascun grafico rappresenta il valore di potenza assorbito in *idle* dal sistema, mentre nel riquadro in alto a destra vengono indicati i valori dell'energia totale aggiuntiva rispetto all'*idle* assorbita da ciascuna applicazione per completare l'esecuzione del carico di lavoro. L'energia consumata è calcolata integrando i valori della potenza assorbita nel tempo impiegato.
Il primo importante risultato che emerge da questi grafici è che la potenza consumata dal *server* quando esegue l'applicazione è significativamente più alta rispetto a quella assorbi-

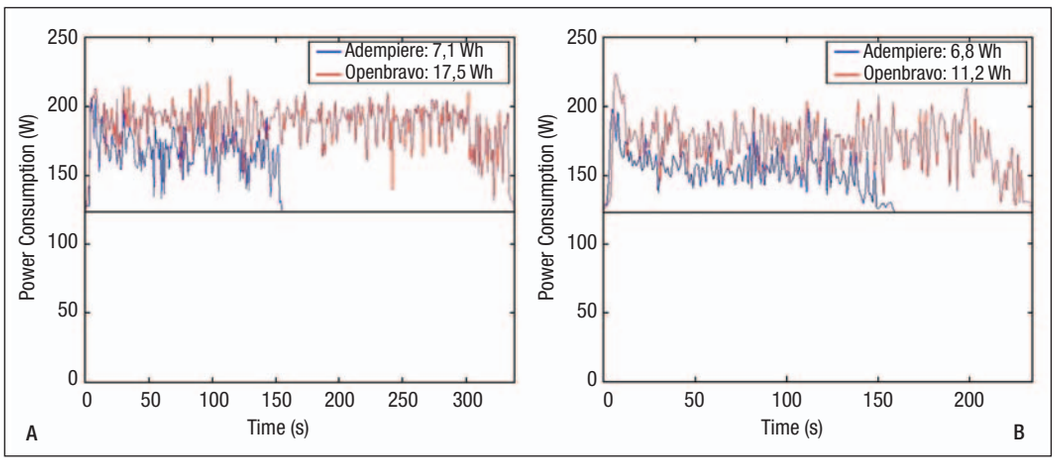


FIGURA 3
Simulazione per sistemi ERP con 3 client:
A - *Windows*
B - *Linux*

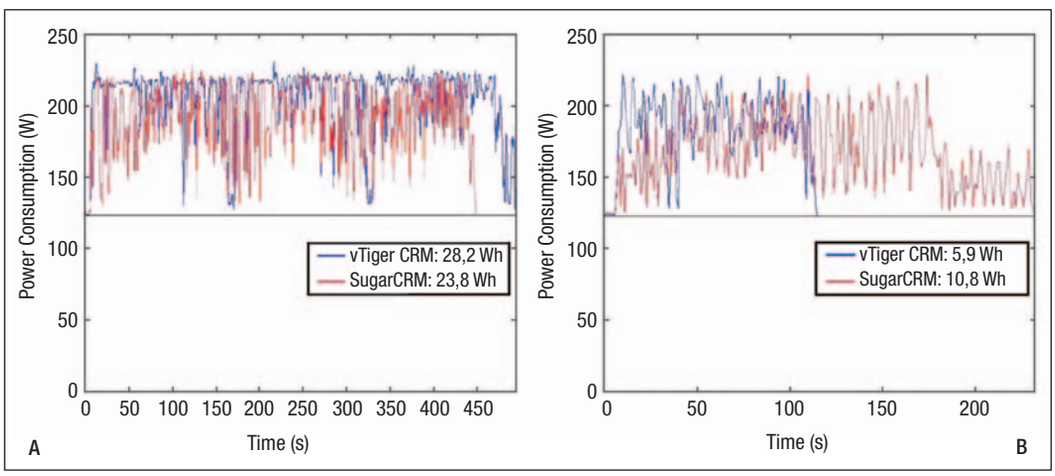


FIGURA 4
Simulazione per sistemi CRM con 3 client:
A - *Windows*
B - *Linux*

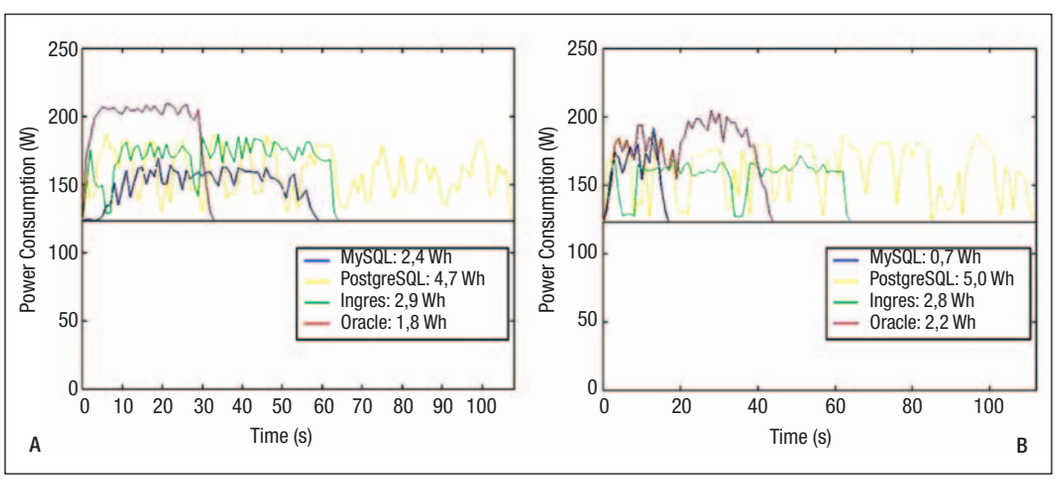


FIGURA 5
Simulazione per sistemi DBMS con 10 client:
A - *Windows*
B - *Linux*

ta in *idle* (fino al 72% in più). Emerge inoltre un altro fatto importante: applicazioni diverse appartenenti alla stessa categoria richiedono al sistema di consumare quantità di energia molto diverse per soddisfare gli stessi requisiti funzionali. Per esempio, OpenBravo per eseguire un particolare carico di lavoro richiede 17,5 Wh, mentre Adempiere per eseguire lo stesso carico richiede solamente 7,1 Wh, con una differenza superiore al 100%. Questi risultati sono consistenti per tutte le categorie di applicazioni e per entrambi i sistemi operativi, con una differenza media del 79%.
Generalmente, l'esecuzione delle applicazioni

in ambiente Linux risulta più efficiente rispetto all'ambiente Windows. Per esempio, come emerge dalla figura 3, il consumo di Adempiere e di Openbravo si riduce rispettivamente del 4% e del 36% quando sono eseguiti su Linux. In media, le applicazioni eseguite in ambiente Linux consumano il 50% in meno di energia.
È tuttavia interessante notare che l'effetto del sistema operativo sul consumo energetico non è del tutto prevedibile. Per esempio, come si può notare dalla figura 4, mentre su Windows, SugarCRM è più efficiente di vTiger, la situazione si inverte sotto Linux, anche se il consumo medio due delle applicazioni in ambiente Linux è comunque inferiore rispetto a quello in ambiente Windows. L'impatto del sistema operativo sul consumo di energia delle applicazioni MIS è quindi molto forte, ma variabile e dipendente dall'applicazione stessa.
Un'osservazione critica dei dati proposti suggerisce inoltre il fatto che l'efficienza energetica debba essere considerata come un nuovo parametro di merito di un'applicazione software, in aggiunta alla *performance*. La relazione tra tempo di esecuzione ed energia consumata non è sempre lineare. Per esempio, Oracle è il DBMS più veloce e che consuma meno energia nel campione considerato. Ingres richiede un tempo doppio per eseguire lo stesso carico di lavoro, ma solo il 60% di energia in più. Analizzando le misure effettuate su tutto il campione, emerge che dimezzare il tempo di esecuzione in media riduce il consumo di energia solamente del 30%. La *performance* temporale non è quindi sufficiente a giustificare la diversità di efficienza energetica delle applicazioni software.

3. LA RICERCA IN AMBITO *GREEN SOFTWARE*

Le osservazioni sperimentali presentate nella sezione precedente dimostrano che il consumo energetico indotto dal software è significativo e lasciano intuire ampi margini di risparmio potenzialmente derivabile dall'ottimizzazione delle applicazioni.
Volendo proporre un programma di ricerca che affronti in modo scientifico questa tematica sono almeno tre i filoni da considerare.
Innanzi tutto, occorre **misurare l'efficienza energetica del software** tramite opportune metriche, esigenza che emerge anche solo per voler presentare in modo più rigoroso i dati proposti nella sezione precedente. Si noti che il concetto di efficienza è diverso da quello di consumo, in quanto rapporta l'energia consumata alla quantità di lavoro che ha permesso di svolgere. Un insieme di metriche per l'efficienza energetica del software può essere utile sia per selezionare il software che viene adottato, complimentando i parametri di merito attualmente utilizzati, sia per valutare il software prodotto *inhouse* e quindi il team di sviluppo. Esse costituiscono lo strumento essenziale per poter confrontare tra loro sistemi diversi e quindi fare *benchmarking*.
Ponendosi invece l'obiettivo di migliorare l'efficienza energetica, gli approcci perseguibili possono essere raggruppati in due grandi categorie. Un primo approccio è quello di **ottimizzare il codice**: scrivere meglio il codice può avere un effetto sulle *performance* energetiche dell'applicazione. Ovviamente, queste tecniche sono applicabili solamente al software sviluppato *inhouse* e possono tradursi sia in metodologie, linee guida e *tool* per lo sviluppo, che in linee guida a livello organizzativo e relative alle competenze e al percorso di formazione degli sviluppatori.
Migliorare l'efficienza energetica ottimizzando il codice potrebbe però non essere sempre fattibile oppure non essere conveniente. Se da una parte potrebbe essere sensato riscrivere e ottimizzare le *routine* più energivore o più comunemente eseguite in un sistema, dall'altra non sarebbe pensabile realizzare l'analisi e l'ottimizzazione di tutto il codice del sistema informativo di una banca e, in ogni caso, lo sforzo necessario per questa operazione difficilmente sarebbe ripagato dai risparmi ottenuti.
Per questo motivo è opportuno sviluppare anche strumenti di **ottimizzazione a livello di sistema**, applicabili cioè in modo trasversale e senza che sia richiesto operare a livello di codice. Molti sono gli strumenti e le azioni che possono essere sviluppati a questo livello e la ricerca è quanto mai aperta. Un esempio di azione, riprendendo le evidenze empiriche discusse nella sezione precedente, potrebbe essere una scelta oculata dello *stack*, considerando l'impatto che l'interazione fra sistema operativo e applicazione ha sul consumo complessivo. Nel seguito di questo articolo saranno pre-

sentati i primi risultati di due ricerche condotte dal Dipartimento di Elettronica e Informazione del Politecnico di Milano riguardanti rispettivamente la gestione green della memoria e del *garbage collector* e l'utilizzo della tabulazione dinamica per aumentare l'efficienza di applicazioni *computation intensive*.
Il paragrafo 4 sintetizza le linee di ricerca proposte individuando per ciascuna di esse i possibili ambiti di applicazione. Nelle sezioni seguenti sarà presentata una panoramica delle ricerche possibili e in corso nei vari filoni.

4. COSA VUOL DIRE EFFICIENZA ENERGETICA DEL SOFTWARE

Esistono già diverse metriche consolidate per misurare l'efficienza energetica dell'IT a vari livelli. Il *Power Usage Effectiveness* (PUE), calcolato come la potenza entrante in un *data center* divisa per la potenza effettivamente assorbita dal carico IT, è comunemente utilizzato per valutare l'efficienza dei livelli infrastrutturali, includendo quindi sistemi di condizionamento e UPS. Per valutare il bilanciamento dei carichi e il livello di virtualizzazione si possono adottare metriche come la percentuale media di utilizzo dei processori, sia fisici che virtuali. Vi sono poi diverse metriche utilizzabili a livello di *server* e di processore per valutare l'efficienza energetica dell'hardware, per esempio W/tpm (Watt per transazioni al minuto), FLOP/Wh (operazioni *floating point* per Wh) oppure MIPS/W (Milioni di istruzioni al secondo per Watt) per gli ambienti *mainframe*.
Tuttavia non vi sono ancora metriche consolidate e sufficientemente generali per misurare l'efficienza energetica del software.
È importante analizzare l'efficienza energetica di un sistema informativo a tutti i suoi livelli, in quanto gli attori coinvolti possono essere molto diversi. Se consideriamo un *data center* infatti, il PUE può essere influenzato da chi gestisce il *data center*, così come il livello medio di utilizzo dei processori. Le prestazioni a livello di hardware dipendono esclusivamente dai fornitori di tecnologia e quindi possono essere influenzate solo in fase di acquisto. Infine, l'efficienza energetica del software dipende dai "clienti" del *data center*, cioè da chi scrive o seleziona le applicazioni software da acquisire.
La difficoltà nella definizione delle metriche di efficienza energetica del software risiede nella quantificazione del "lavoro" svolto da un'applicazione con una quantità unitaria di energia. Riprendendo il paragone presentato inizialmente relativo al mondo automobilistico, se l'efficienza di un'automobile può essere misurata in chilometri percorsi con un litro di benzina, nell'ambito del *green software* non è facile misurare i chilometri, mentre, come si è dimostrato nel paragrafo 2, è relativamente semplice misurare i litri di benzina consumata, ovvero l'energia.
Questo sposta il problema alla definizione di concetti di transazione o di carichi di lavoro standard, il più possibile generali, da far eseguire ad un'applicazione per effettuare le misure. Alcuni istituti come il TPC hanno definito dei carichi di lavoro standard che possono essere utilizzati a questo scopo, anche se molto spesso tali *benchmark* sono finalizzati più a testare un intero sistema che non l'applicazione finale. Per alcune applicazioni la definizione di transazione è abbastanza intuitiva (per esempio, per i DBMS), mentre per altre, come gli ERP, occorre considerare differenti tipologie di transazione e quindi diverse metriche di efficienza. Le diverse transazioni (per esempio, inserimento di un nuovo ordine oppure inserimento di una nuova anagrafica prodotto nei sistemi ERP) corrispondono a differenti moduli e porzioni di codice dell'applicazione, quindi è sensato adottare metriche diverse, anche se è auspicabile giungere ad una metrica unica per ciascuna applicazione, derivata da una media degli indicatori rispetto alla frequenza di utilizzo. Nella definizione di queste metriche occorre anche tenere presente i dati forniti come input e le dimensioni dei database sottostanti (per esempio, a livello di consumo di energia è la stessa cosa modificare la giacenza a magazzino di un prodotto quando ci sono 10 tipologie di prodotti presenti o quando ce ne sono 1.000?).
È evidente come la definizione e il calcolo di queste metriche sia più complicata che per gli altri livelli infrastrutturali.
Una soluzione alternativa potrebbe essere quella di identificare altre metriche basate sul codice, in qualche modo correlate all'efficienza energetica e utilizzarle come *proxy*. Tali metriche risulterebbero molto comode in quanto sarebbero misurabili tramite opportuni strumenti software semplicemente analizzando il

codice dell'applicazione, senza bisogno di eseguirla. Tuttavia le ricerche in questo senso non hanno fino ad ora portato a risultati apprezzabili. Il consumo di energia delle applicazioni analizzate e descritte nel paragrafo 2, e quindi l'efficienza energetica, sono risultate scorrelate da tutte le metriche tradizionali di qualità del *design* del codice (per esempio, coupling, ereditarietà, modularità ecc.).

5. SCRIVERE SOFTWARE PIÙ EFFICIENTE DAL PUNTO DI VISTA ENERGETICO

Come visto nel paragrafo 2, applicazioni che soddisfano gli stessi requisiti funzionali, ma strutturalmente diverse, possono indurre consumi significativamente differenti. Per capire più in dettaglio quali siano le cause di questa diversità di comportamento e quindi identificare possibili strategie di ottimizzazione è necessario analizzare in modo approfondito le applicazioni considerate. Nel prossimo paragrafo sono riportati i risultati relativi ad un'analisi più dettagliata del comportamento dei due ERP Adempiere e OpenBravo.

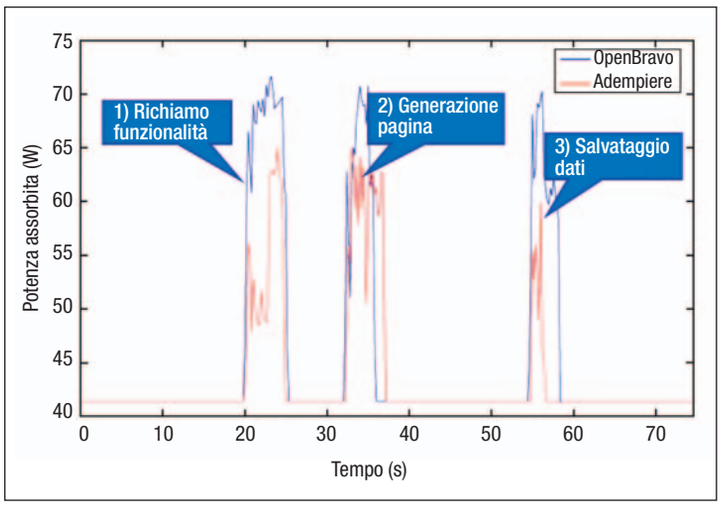


FIGURA 6

Potenza assorbita da Adempiere e OpenBravo per l'inserimento di un nuovo prodotto

Fase	Tempo impiegato da Adempiere rispetto a OpenBravo	Energia consumata da Adempiere rispetto a OpenBravo
1) Richiamo funzionalità	–5%	–50%
2) Generazione pagina	+28%	+4%
3) Salvataggio dati	–61%	–80%
Totale	**–12%**	**–43%**

TABELLA 1

Confronto dei tempi di esecuzione e dell'energia consumata da Adempiere e OpenBravo per l'inserimento di un nuovo prodotto

5.1. L'effetto dei livelli di astrazione

Il confronto tra i due ERP è stato effettuato ovviamente sulla stessa infrastruttura hardware e utilizzando il medesimo DBMS, in particolare PostgreSQL. Le differenze più evidenti nella struttura dei due sistemi analizzati sono due: Adempiere accede alla base di dati tramite una classe che implementa le istruzioni SQL necessarie, mentre OpenBravo gestisce i dati tramite il *framework* HIBERNATE. Inoltre Adempiere, al contrario di OpenBravo utilizza *servlet* con tecnologia Ajax.

L'inserimento di un nuovo prodotto a livello di flusso di esecuzione comporta tre fasi principali: il richiamo della funzionalità, la generazione della pagina che permette all'utente di inserire i dati e infine il salvataggio degli stessi.

Come è già emerso dai risultati discussi in precedenza, Adempiere è generalmente più efficiente di OpenBravo. La figura 6 mostra in dettaglio l'assorbimento di potenza da parte delle due applicazioni durante l'esecuzione del carico di lavoro, e la tabella 1 riporta la differenza percentuale in termini di tempo di esecuzione e di energia consumata rispetto all'*idle*. I picchi di consumo riportati nella figura 6 corrispondono alle tre fasi descritte. Nella prima fase benché il tempo di esecuzione sia quasi lo stesso, il consumo di OpenBravo è decisamente maggiore in quanto vengono caricate le parti di HIBERNATE necessarie, operazione che richiede un alto utilizzo del processore e quindi un alto assorbimento di potenza. HIBERNATE consente una migliore *performance* temporale nella fase successiva (la generazione della pagina richiede di leggere alcune informazioni dalla memoria), tuttavia il consumo di energia rimane più alto rispetto all'esecuzione diretta delle *query* SQL a causa dell'*overhead* dovuto all'esecuzione del *framework*. Complessivamente Adempiere è più veloce del 12% rispetto ad OpenBravo, ma il consumo di energia è inferiore di ben il 43%. L'utilizzo di HIBERNATE fa sì che l'e-

secuzione sia più CPU-*intensive*, con un impatto negativo sull'efficienza.
In generale l'utilizzo di livelli di astrazione e di *application development environment* è molto diffuso nei moderni approcci allo sviluppo del software in quanto semplifica la programmazione. Questi risultati empirici suggeriscono tuttavia che il loro utilizzo abbia un impatto non trascurabile sull'efficienza energetica del software.

5.2. Programmatori ninja o bravi operai?

L'utilizzo di livelli di astrazione permette di semplificare la scrittura e la manutenzione del codice, e quindi riduce il costo di sviluppo. È sicuramente più facile sviluppare o modificare un'applicazione che accede ai dati tramite un *framework* come HIBERNATE, che maschera la complessità della struttura dati e della gestione del *data base*, rispetto ad avere a che fare ad una classe Java che incorpora il codice SQL per le query necessarie. I livelli di astrazione, le librerie e i componenti standard e i *framework* di sviluppo permettono di impiegare per lo sviluppo delle applicazioni programmatori non specializzati, ossia "bravi operai" che possono essere formati per l'uso di quei particolari strumenti e componenti preconfezionati in poco tempo e con un investimento ridotto. Queste applicazioni soddisfano i requisiti funzionali, ma la generalità che deve caratterizzare gli strumenti utilizzati non può che inficiarne le prestazioni. Se le stesse applicazioni fossero sviluppate da programmatori "ninja", capaci di scrivere codice ottimizzato specifico per il contesto di utilizzo, l'efficienza risultante sarebbe probabilmente maggiore.
Un'ulteriore conferma del fatto che scrivere bene il codice influisce pesantemente sull'efficienza, emerge anche da un'analisi empirica effettuata su diverse implementazioni della funzione XIRR utilizzate da alcune banche italiane. La funzione XIRR calcola il tasso di rendimento di un investimento e richiede l'identificazione degli zeri di un polinomio. Questa operazione può essere svolta utilizzando diversi algoritmi, con un impatto devastante sul consumo di energia, come si evince dalla figura 7.
Ottimizzare l'implementazione della funzione utilizzando l'algoritmo più efficiente riduce il consumo di energia di 3 ordini di grandezza, ma richiede programmatori con una profonda esperienza del dominio e degli algoritmi di programmazione.
Ovviamente l'impiego di programmatori "ninja" fa aumentare considerevolmente i costi di sviluppo, e potrebbe alzare anche i costi di manutenzione in quanto il codice potrebbe essere più difficile da interpretare e modificare. Occorre quindi valutare il TCO esteso di un'applicazione e bilanciare i vari fattori per determinare la strategia di sviluppo ottima. Ad oggi la ricerca non ha ancora prodotto indicazioni chiare in questo senso, ma è molto probabile che un approccio 80-20 si possa rivelare vincente: ottimizzare tramite programmatori specializzati il 20% del codice responsabile dell'80% dei consumi, puntando invece sulla facilità di sviluppo della restante parte.

5.3. Ricerche future

L'aumento dell'efficienza energetica tramite l'ottimizzazione del codice rimane un campo aperto per la ricerca. Come discusso in precedenza è probabile che le *skill* e la formazione dei programmatori abbiano un impatto sull'efficienza del codice prodotto, anche se non è ancora chiaro in che misura questo avvenga e quali siano le *skill* con un impatto più diretto sull'efficienza energetica.
Occorre inoltre identificare i costrutti, i *design pattern* e le metodologie di programmazione che permettono di ridurre i consumi indotti dalle applicazioni. Il metodo più immediato per far progredire la ricerca in questo senso è la realizzazione di esperimenti di comparazione tra porzioni di codice che effettuano le stesse operazioni, utilizzando costrutti e strutture

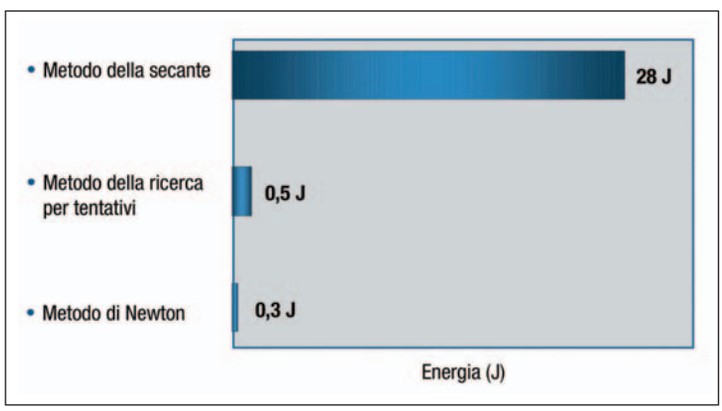


FIGURA 7
Consumo di energia della funzione XIRR implementata secondo diversi algoritmi

diverse. L'analisi dettagliata del flusso di esecuzione permette di identificare in che misura le diverse parti sono responsabili dei consumi e quindi di definire *best practice* di sviluppo. Tali *best practice* potrebbero confluire in strumenti di ausilio allo sviluppo, in grado di guidare il programmatore anche meno esperto ad una programmazione "*green oriented*".
Queste analisi possono essere complementate dallo studio dei profili e dei percorsi formativi degli sviluppatori, per poter arrivare alla definizione complessiva di strategie per lo sviluppo "*green*".

6. OTTIMIZZARE IL CONSUMO A LIVELLO DI SISTEMA

I risultati riportati nel paragrafo 5 evidenziano come, per ottenere una base di codice che garantisca l'efficienza energetica, si possa fare leva sul livello di astrazione adottato nella programmazione delle applicazioni – riducendo il peso dello stack applicativo si liberano risorse e si ottiene un controllo a granularità più fine di quelle effettivamente utilizzate. Questo tipo di azione richiede, rispetto a basi di codice esistente sviluppato senza tener conto dei problemi di consumo, sostanziali revisioni. Ciò implica un considerevole costo di sviluppo, dato che la revisione deve essere affidata a programmatori abili e dotati di esperienza specifica, ma anche un potenziale incremento nei costi di manutenzione, dato che la leva impiegata sposta inevitabilmente il rapporto fra software "*off the shelf*" e software sviluppato internamente o quantomeno *ad hoc* a vantaggio di quest'ultimo.
Di conseguenza, queste tecniche sono utilizzabili principalmente nella scrittura di nuovo codice e in particolare per applicazioni critiche, in cui il costo aggiuntivo di sviluppo e manutenzione sia più che bilanciato dai benefici ottenuti in termini di risparmio energetico, in modo da garantire nel complesso una diminuzione del TCO.
Al contrario, per grandi basi di codice preesistenti, in cui il costo di manutenzione può risultare il fattore critico, è necessario trovare leve diverse. Sono desiderabili in questo caso metodologie di ottimizzazione almeno semiautomatica, applicabili a livello di sistema sull'intera base di codice da personale che abbia competenze di tipo sistemistico, e che non richiedano quindi modifiche manuali al codice, ma piuttosto la regolazione di parametri di ottimizzazione.
Le sezioni seguenti sono dedicate ad illustrare due tipi di intervento a livelli diversi: la regolazione della gestione della memoria e la memoizzazione semiautomatica al livello dell'applicazione.

6.1. L'impatto del *garbage collector* e della gestione della memoria

Uno dei contributi principali alla complessità di sviluppo dei sistemi software e nello stesso tempo alle prestazioni dei sistemi sviluppati è dato dalla gestione della memoria. Nella maggior parte degli ambienti di sviluppo e dei linguaggi di livello medio-alto (per esempio Java SE o Python), tale gestione è automatizzata attraverso un livello software intermedio che, a tempo di esecuzione, fornisce un'ulteriore astrazione sopra la memoria virtuale offerta dal sistema operativo. Al contrario, nello sviluppo a basso livello (per esempio nel linguaggio C o in certe specializzazioni di Java per sistemi real-time), la gestione della memoria è sovente demandata al programmatore[1].
Sono evidenti i vantaggi della gestione automatica in termini di facilità di sviluppo: la gestione automatica della memoria attraverso un *garbage collector* riduce i problemi dovuti ad oggetti non deallocati correttamente, in quanto il sistema si prende carico della deallocazione di oggetti non più raggiungibili[2].

[1] Non è sempre così, in quanto la gestione automatica della memoria può essere aggiunta a qualsiasi linguaggio di programmazione che non ne disponga in modo nativo attraverso librerie e una opportuna disciplina di programmazione. Nel caso del linguaggio C, la libreria di *garbage collection* più nota è il Boehm GC.

[2] Sebbene il *garbage collector* sia in grado di gestire gli errori più comuni dovuti a puntatori contenenti indirizzi di aree deallocate (*dangling pointer*) o aree allocate ma non più raggiungibili in quanto non vi sono più puntatori che ne mantengano gli indirizzi (*memory leak*), è comunque infondata l'idea che esso sia in grado di risolvere *tutti* gli errori relativi alla memoria. In realtà, oggetti non più utilizzati ma raggiungibili attraverso riferimenti da altri oggetti ancora utili possono portare a situazioni non dissimili dai più tradizionali *memory leak*, e persino più difficili da identificare, in quanto non immediatamente distinguibili dalla situazione corretta [11].

Tuttavia, le relazioni fra l'attività del *garbage collector*, le prestazioni e i consumi di energia legati alle applicazioni che ne fanno uso non sono immediatamente evidenti [14].
L'effetto del *garbage collector* sul profilo di consumo di memoria di un'applicazione, in linea di massima, ne altera limitatamente i picchi massimi – vi sarà infatti un costo, in genere contenuto, dovuto alle strutture dati mantenute per la gestione della memoria – ma impatta in modo più marcato sui valori medi, in quanto l'eliminazione dalla memoria delle strutture dati non più in uso non può essere troppo frequente.
In particolare, bisogna ricordare che la frequenza dell'operazione di "*collection*" degli oggetti non più raggiungibili (e quindi non più necessari) ha un effetto immediato sulle prestazioni e sul consumo di energia – si tratta infatti di una operazione onerosa, in quanto richiede che *tutte* le strutture dati vengano traversate, per confermare la loro raggiungibilità a partire dall'insieme delle variabili non allocate nello *heap* (il *root-set*). Quindi, in molti casi sarà preferibile mantenere in memoria strutture dati non più necessarie, ma in cambio ridurre la frequenza della *collection*.
Un'ulteriore complicazione è data dall'interferenza fra il *garbage collector* e il sottosistema di allocazione della memoria. Il *garbage collector*, infatti, si appoggia sulle normali primitive offerte dal sistema operativo per allocare la memoria (malloc/realloc), ma richiede in blocco aree piuttosto grandi di memoria, usandole poi internamente per l'allocazione di più oggetti richiesti dall'applicazione. L'operazione di allocazione interna al *garbage collector* può facilmente risultare più efficiente rispetto all'equivalente operazione offerta dal sistema operativo, e quindi il *garbage collector*, se si esclude il costo dell'operazione di *collection*, può risultare più efficiente dal punto di vista delle prestazioni rispetto all'allocatore di memoria offerto dal sistema operativo, in quanto riduce le operazioni svolte da quest'ultimo e introduce al loro posto operazioni meno onerose.
L'impiego del *garbage collector* - e anche il tipo di algoritmo di *garbage collection* utilizzato - ha un impatto notevole sulle prestazioni complessive del sistema. Nella figura 8 riportiamo i risultati in termini di consumo energetico di un *benchmark* sintetico, basato sull'allocazione di liste di pari dimensioni, realizzato sia in Java che in C. I risultati sono riportati al variare della dimensione massima dello *heap*, per tenere conto dell'impatto diverso dell'operazione di *collection*. La versione Java evidenzia come l'astrazione introdotta abbia di fatto reso indipendente il consumo dal sistema operativo: la memoria viene infatti preallocata alla JVM, la cui gestione è responsabile del profilo di consumo. Si può anche notare che il consumo scende nettamente al crescere della disponibilità di memoria, in quanto il *garbage collector* esegue l'operazione di *collection* molto più raramente.
Il *garbage collector* impiegato nella versione C del *benchmark* (Boehm GC) non sfrutta invece questa opportunità e quindi la memoria massima disponibile non incide sui tempi di esecuzione né sui consumi.
Un aspetto importante è dato dal tenere in considerazione la natura degli oggetti allocati in memoria: la maggior parte delle applicazioni impiega una parte della propria memoria *heap* per oggetti quasi permanenti e la parte restante per oggetti a vita breve, che tipicamente vengono allocati e deallocati con elevata frequenza.
Il rapporto fra la memoria occupata da oggetti a vita breve e oggetti a vita lunga determina

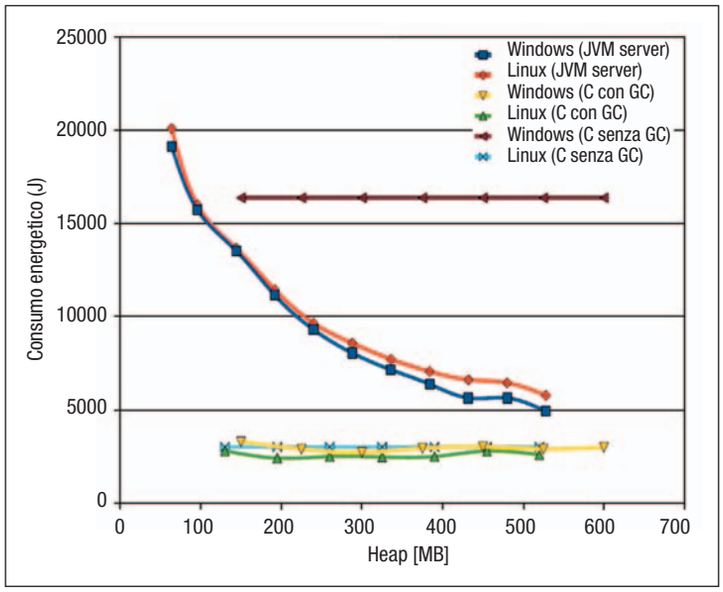


FIGURA 8
Consumi energetici al variare della dimensione massima dello heap

l'efficacia degli algoritmi di *garbage collection* più semplici, come quello impiegato dal Boehm GC. Nella figura 9 riportiamo i risultati relativi ad una variante del *benchmark* sintetico impiegato in precedenza, in cui si fa variare la percentuale di oggetti a vita breve.
Si può notare che il *garbage collector* funziona bene quando più della metà degli oggetti hanno vita breve, ma è superato dall'allocazione manuale nel caso contrario. Questo effetto non si verifica con algoritmi di *garbage collection* più avanzati, come quelli generazionali, che mantengono traccia degli oggetti allocati da più tempo (e quindi probabilmente di tipo quasi permanente), ed evitano di eseguire su di essi l'operazione di *collection*. È questo, per esempio, il caso dell'algoritmo di *garbage collection* impiegato nella macchina virtuale Java.
In prospettiva, è importante che l'allocazione e la deallocazione automatica della memoria sia controllata, sia in termini di algoritmi di *collection* sia in termini di determinazione della memoria preallocata (o dei blocchi di memoria che vengono allocati alla macchina virtuale quando la memoria preallocata è esaurita).
Questo tipo di ottimizzazione può essere automatizzata e integrata nella macchina virtuale, o anche svolta in modo manuale o semiautomatico dal gestore del sistema – in questo caso, con il vantaggio di poter ottimizzare in modo coordinato la disponibilità di memoria di più applicazioni.

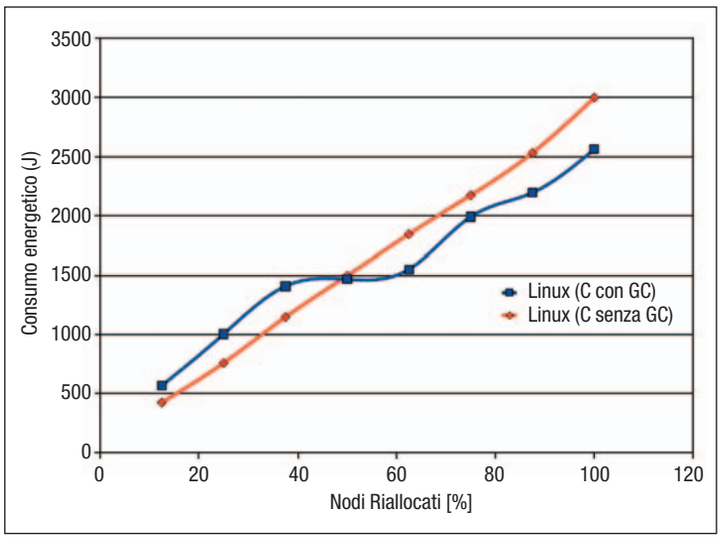


FIGURA 9
Impatto della frequenza di oggetti a vita breve nel Boehm GC

6.2. La memoizzazione per ottimizzare consumi e performance

Negli attuali elaboratori, le memorie hanno, in generale, consumi meno fortemente dipendenti dal carico di lavoro rispetto alle unità di elaborazione. Di conseguenza, ci si attende che, in quei casi in cui sia possibile un *trade-off* fra spazio e tempo di elaborazione, sia utile - anche dal punto di vista dei consumi energetici - impiegare quantità maggiori di memoria se questo porta a dei risparmi sui tempi di esecuzione. In sostanza, si tratta di memorizzare i risultati di computazioni frequenti e sostituire le successive istanze degli stessi calcoli con letture dei valori memorizzati.
Approcci simili, noti sotto il nome di *memoizzazione*, sono stati impiegati in passato, sia nell'ambito dei linguaggi funzionali – per esempio per applicazioni scientifiche per la costruzione di *parser* e per la programmazione dinamica [12, 13].
Questo tipo di approccio è applicabile quando il numero di valori assunti dai parametri delle funzioni non è troppo ampio. In questi casi, la tabulazione delle funzioni stesse è possibile nei limiti della memoria disponibile.
In generale, le tecniche di memoizzazione sono applicabili solo su funzioni pure, che non abbiano cioè effetti collaterali, come modifiche a variabili non automatiche che non vengano deallocate prima del completamento della funzione, oppure interazioni con l'utente o dispositivi esterni. La ragione di questa limitazione risiede nel fatto che queste interazioni sarebbero perse se l'esecuzione della funzione fosse sostituita dalla semplice lettura del risultato da una tabella.
Tuttavia, quando questi requisiti sono soddisfatti, la memoizzazione può dare ottimi risultati. Come esempio, prendiamo il caso della funzione di calcolo dello XIRR già vista nel sottoparagrafo 5.2. Questa funzione è piuttosto onerosa dal punto di vista dell'occupazione del processore, anche se abbiamo visto che un'attenta selezione dell'algoritmo impiegato permette di aumentare drasticamente le prestazioni. Si tratta di una funzione pura, dato che non modifica i parametri in ingresso, ed esegue solo il calcolo del valore XIRR cercato. Può essere quindi memoizzata: le prestazioni, in termini di TCO, dello XIRR con memoizzazione superano quelle dell'algoritmo migliore. In questo caso, infatti, il tempo di esecuzione dell'algoritmo miglio-

re per lo XIRR è superiore rispetto al tempo di accesso alla tabella in cui sono memoizzati i risultati dello XIRR. In termini generali, la convenienza della memoizzazione per una data funzione pura *f* dipende dalla *frequenza di successo* α, ovvero il rapporto fra il numero di invocazioni di *f* per cui è possibile trovare il risultato in memoria e il numero totale di invocazioni di *f*. In modo più preciso, il tempo di calcolo di *f* con memoizzazione è espresso, in funzione di α, del tempo di esecuzione dell'algoritmo originale T_{comp}, e dei tempi di accesso alla memoria in caso di successo (T_{hit}) e fallimento (T_{miss}):

$$T_{memo} = \alpha T_{hit} + (1 - \alpha)(T_{miss} + T_{comp})$$

Si raggiunge il punto di *break even* quando $T_{memo} = T_{comp}$.
Se la differenza nei tempi di accesso alla memoria in caso di successo e fallimento è circa uguale, ne risulta che, per ottenere un guadagno, deve essere vera la seguente relazione:

$$\alpha > T_{miss}/T_{comp}$$

La conclusione è che, da una parte, è necessario scegliere funzioni computazionalmente onerose e che, allo stesso tempo, vengano invocate frequentemente con gli stessi parametri, e dall'altra implementare tabelle di memoizzazione con tempi di accesso rapidi.
Di conseguenza, l'impiego della tecnica di memoizzazione richiede l'impiego di programmatori e progettisti esperti – non solo perché sono necessarie la modifica del codice e l'implementazione di strutture di memoizzazione molto efficienti, ma soprattutto perché l'individuazione delle funzioni che devono essere memoizzate non è banale, ed è cruciale per ottenere un effetto significativo sui consumi e sulle prestazioni. Inoltre, modifiche al codice come quelle richieste per la memoizzazione richiedono l'accesso al codice sorgente, che, nel caso di codice sviluppato da terze parti, potrebbe non essere disponibile.
Per ovviare a questi problemi, si può cercare di automatizzare almeno in parte il processo di ottimizzazione del codice attraverso la memoizzazione. A questo scopo, è stato sviluppato presso il Politecnico di Milano un prototipo di strumento per la memoizzazione semiautomatica. Lo strumento opera su codice bytecode Java, quindi senza richiedere l'accesso al codice sorgente, ed è costituito dai componenti illustrati nella figura 10:

- analisi statica del codice: permette di deter-

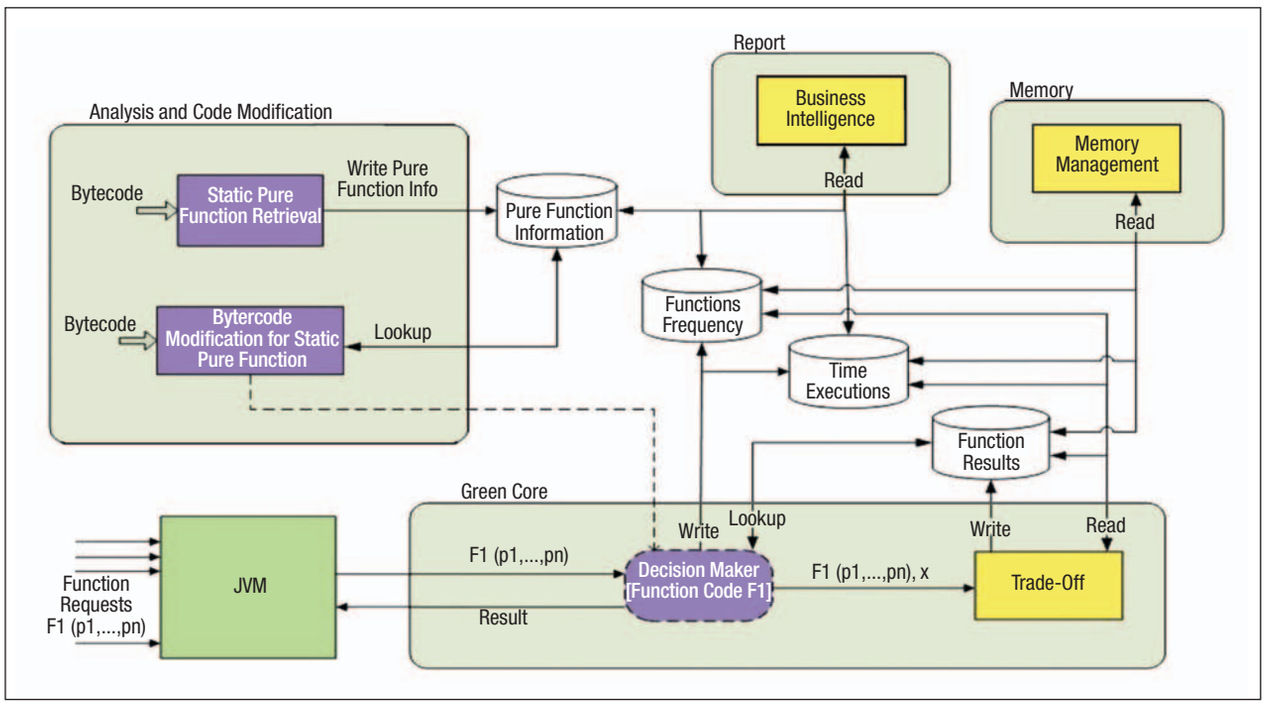


FIGURA 10
Schema dell'architettura per la memoizzazione

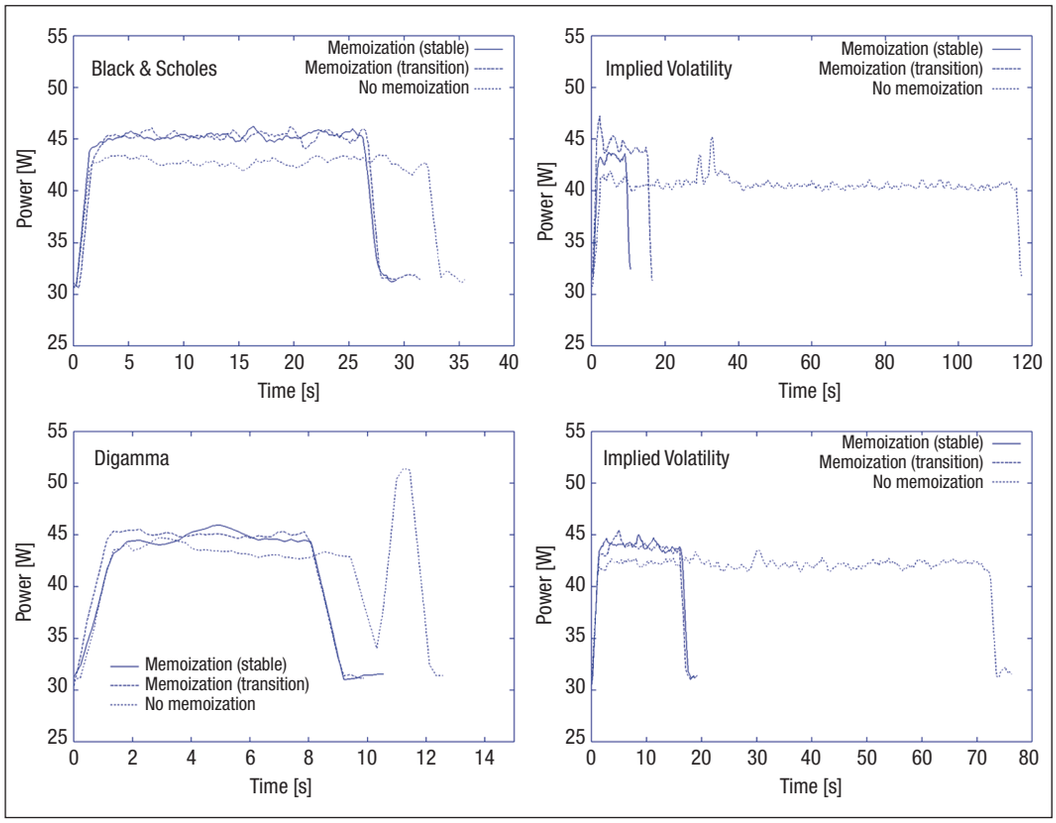


FIGURA 11

Confronto della potenza assorbita dalle funzioni analizzate con e senza l'approccio di memoizzazione

minare quali metodi Java possono essere considerati *funzioni pure* ai fini della memoizzazione;

❑ selezione dei metodi da memoizzare: le funzioni pure che risultano troppo piccole per poter fornire un contributo significativo (T_{miss} ~= T_{comp}) sono automaticamente scartate, ma viene richiesto un intervento esplicito all'utente per selezionare fra i metodi restanti quelli ritenuti più importanti;

❑ instrumentazione del codice: inserisce all'interno del bytecode dei metodi selezionati il codice necessario per realizzare la memoizzazione;

❑ moduli di runtime: durante l'esecuzione, le tabelle di memoizzazione vengono popolate, inserendo man mano i valori calcolati (se un valore richiesto non è presente in memoria, viene calcolato impiegando l'algoritmo originale e inserito nella tabella opportuna);

Il prototipo è stato testato su un insieme di funzioni finanziarie e matematiche per valutare l'impatto della memoizzazione sia sulle prestazioni che sui consumi. Riportiamo nella figura 11 profili di consumo per quattro casi di studio:

❑ *Implied Volatility*: calcolo della volatilità di una *stock option* dato il prezzo di mercato ed un modello di prezzo delle opzioni [8];

❑ *Black & Scholes*: calcolo del prezzo delle opzioni *put* e call secondo il metodo *Black-Scholes-Merton* [3];

❑ *Binomial*: calcolo dei coefficienti binomiali;

❑ *Digamma*: calcolo della derivata seconda logaritmica della funzione gamma.

Per ciascuno dei metodi in esame, si riportano i profili relativi all'esecuzione del codice non modificato e all'esecuzione del codice modificato, partendo da tabelle di memoizzazione vuote, e infine all'esecuzione del codice modificato, partendo però da tabelle di memoizzazione già inizializzate. I parametri delle invocazioni sono generati casualmente, secondo un modello gaussiano.

Si nota che, una volta esaurito il transitorio necessario per il popolamento delle tabelle, il codice modificato automaticamente ottiene consumi complessivi migliori, grazie ad una

		Tg [s]	Risp. [%]	Δ Int [J]	Risp. [%]	Δ Media [W]	Risp. [%]
Implied Vol.	*Orig.*	117,50		824,46		7,01	
	Trans.	17,66	85%	182,05	78%	10,29	–47%
	Stabile	11,89	90%	81,92	90%	6,85	2%
Black Scholes	*Orig.*	35,74		284,08		7,94	
	Trans.	28,22	21%	267,78	6%	9,47	–19%
	Stabile	26,54	8%	266,53	2%	10,03	-30%
Binomial	*Orig.*	76,50		613,68		8,02	
	Trans.	18,31	76%	150,43	75%	8,19	–2%
	Stabile	18,86	75%	147,95	76%	7,82	2%
Digamma	*Orig.*	12,79		107,92		8,40	
	Trans.	10,01	22%	85,88	20%	8,52	–1%
	Stabile	10,23	19%	76,78	29%	7,42	12%

TABELLA 2

Sintesi dei risultati (Tg = tempo di esecuzione; Δ Int = differenza dei consumi, calcolati come integrale della potenza assorbita sul tempo di esecuzione; Δ Media = differenza della potenza media)

netta riduzione dei tempi di esecuzione. Nei casi in cui l'algoritmo originale è più complesso, come *Implied Volatility*, si ottiene anche una riduzione dei consumi istantanei medi.
La fase di transitorio, valutata con l'esecuzione del codice modificato nel caso con tabelle di memoizzazione inizialmente vuote, mostra invece un peggioramento dei consumi istantanei medi, dovuto alla maggiore attività, e di conseguenza anche una diminuzione dei benefici complessivi.
Nella tabella 2 riportiamo una sintesi dei risultati, sia in termini di tempi di esecuzione sia in termini di consumi energetici.

6.3. Ricerche future

La memoizzazione può essere molto efficace, come abbiamo visto, ma è necessario poterla applicare a funzioni critiche per i consumi. Spesso, l'analisi automatica del codice non è in grado di determinare se una funzione critica può essere memoizzata oppure no. In questo caso, è necessario riportare al programmatore o al gestore del sistema nel modo più chiaro le ragioni per cui il sistema non è in grado di prendere la decisione, in modo che il dubbio possa essere risolto nel tempo minimo. Inoltre, è necessario integrare i diversi approcci al risparmio energetico, per garantire che non vi siano interferenze negative.

7. CONCLUSIONI

Il consumo energetico è uno dei fattori di costo primari per i sistemi informatici. Sebbene i processori non siano la principale fonte di consumi all'interno di un sistema, il software che viene eseguito su di essi determina l'attività dell'intero sistema.
Come in molti casi simili, anche nel consumo energetico dovuto all'esecuzione di applicazioni software vale la legge di Pareto, per cui la maggior parte degli effetti dipende da poche cause, che devono essere identificate per poter mettere in atto strategie di ottimizzazione efficaci.
Abbiamo presentato due diversi tipi di ottimizzazione, che operano a livelli diversi nello *stack* applicativo, ma è importante rimarcare che si tratta solo di alcuni esempi – vi sono molti altri aspetti della progettazione e della manutenzione del software che devono essere esplorati, e possono, nelle opportune condizioni operative, fornire significativi risparmi energetici.
In futuro, sarà determinante per minimizzare

i consumi la capacità di coordinare queste ottimizzazioni sull'intero *stack* applicativo.

Ringraziamenti

Gli autori ringraziano la Prof. Chiara Francalanci e l'Ing. Marco Bessi per il prezioso contributo. Si ringrazia inoltre Accenture Italia per aver supportato le fasi iniziali della ricerca.

Bibliografia

[1] ACEEE: *A Smarter Shade of Green*. ACEEE Report for the Technology CEO Council, 2008.

[2] Barroso L.A., Hölzle U.: T*he Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Ed. Morgan & Claypool, Madison, 2009.

[3] Capra E., Formenti G., Francalanci C., Gallazzi S.: *The impact of MIS software on IT energy consumption*. European Conference of Information Systems, 2010.

[4] Fornaciari W., Gubian P., Sciuto D., Silvano C.: Power estimation of embedded systems: A hardware/software codesign approach. *IEEE Trans. on VLSI Systems*, Vol.6, n. 2, 1998, p. 266-275.

[5] Lee C., Brown E.G.: *Topic overview: Green it*. Technical report, Forrester Research, Novembre 2007.

[6] Katz D.M.: *CIOs called clueless about extra costs*. CFO.com, 27 settembre 2010.

[7] Kumar R.: *Important power, cooling and green it concerns*. Technical report, Gartner, Gennaio 2007.

[8] Saxena A., Chung D.: *Optimizing the datacenter for cost and Efficiency*. IDC White Paper, 2009.

[9] Stanford E.: *Environmental trends and opportunity for computer system power delivery*. 20-th Int'l Symposiumon Power Semiconductor Devices and IC's, 2008.

[10] Vahdat A., Lebeck A., Ellis C.S.: *Every joule is precious: the case for revisiting operating system design for energy efficiency*. ACM SIGOPS European Workshop, 2000, p. 31-36.

[11] Xu G.Q., Rountev A.: *Precise memory leak detection for java software using container profiling*. ACM/IEEE 30-th International Conference on Software Engineering, 2008. ICSE '08, May 2008, p.151-160.

[12] Acar Umut A., Blelloch Guy E., Harper R.: S*elective memoization*. In: Proceedings of the 30-th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '03, New York, NY, USA, 2003. ACM, p. 14-25.

[13] Norvig P.: Techniques for automatic memoization with applications to context-free parsing. *Comput. Linguist.*, Vol. 17, March 1991, p. 91-98.

[14] Berger E.D., Hertz M.: *Quantifying the performance of garbage collection vs. explicit memory management*. 2005.

[15] Sissa G.: Green Software. *Mondo Digitale*, settembre 2009.

Giovanni Agosta è ricercatore confermato di Sistemi di Elaborazione dell'Informazione presso il Politecnico di Milano, dove ha conseguito la laurea in Ingegneria Informatica nel 2000 e il Dottorato in Ingegneria dell'Informazione nel 2004 e dove è docente di Algoritmi e Principi dell'Informatica e Piattaforme Software per la Rete. La sua attività di ricerca verte principalmente sulle interazioni tra architetture di elaborazione e compilatori, spaziando dalla compilazione dinamica alle metodologie di progetto e implementazione di applicazioni per sistemi dedicati, ai modelli di programmazione per architetture parallele e alla sintesi logica. È autore di più di trenta articoli su riviste e conferenze internazionali.
E-mail: agosta@elet.polimi.it

Eugenio Capra è professore a contratto di Sistemi Informativi al Politecnico di Milano, presso cui ha conseguito il Dottorato di Ricerca in Ingegneria dell'Informazione, nel 2008, e la laurea in Ingegneria Elettronica nel 2003. È stato Visiting Researcher presso la Carnegie Mellon West University (NASA Ames Research Park, CA) da settembre 2006 a marzo 2007. Ha lavorato come business analyst per McKinsey & Co. dal 2004 fino al 2005, svolge attività di consulenza nell'ambito di gestione e innovazione dei processi IT. Le sue attività di ricerca principali riguardano il Green ICT, i modelli manageriali in ambiente open source e l'impatto dell'IT sui processi di business. Su questi temi ha scritto diversi articoli a livello sia nazionale che internazionale.
E-mail: eugenio.capra@polimi.it