UNIVERSITA' DEGLI STUDI
DI MILANO
DIPARTIMENTO DI SCIENZE
DELL'INFORMAZIONE

BULL ITALIA
CENTRO STUDI
INFORMATICA
E AUTOMAZIONE

Note di Software

Anno 1989
46
Dicembre

Note di software è una pubblicazione trimestrale, edita a cura del Dipartimento di Scienze dell' Informazione dell' Università di Milano e dal Centro Studi Informatica e Automazione della Bull Italia.

Note di software intende costituire uno strumento per la circolazione di informazioni, idee, metodologie ed esperienze nell'area delle tecnologie e della produzione del software.

La collaborazione a **Note di software** è libera. Chi desiderasse pubblicare un contributo è pregato di prendere contatto con il Direttore responsabile o con il Comitato di Redazione. In particolare si sollecitano contributi nella forma di monografie, da inviare in triplice copia, e che non dovrebbero superare le cinquemila parole.
I lavori ricevuti per la pubblicazione verranno revisionati dai membri del Comitato di Revisione Scientifica, che possono avvalersi della collaborazione di specialisti del settore. I lavori pubblicati riflettono comunque le opinioni dei loro autori.

Note di software viene distribuito ad Aziende, Enti e specialisti del settore che ne facciano richiesta alla Redazione.

Autorizzazione del Tribunale di
Milano n. 87 del 23 febbraio 1977

Supporto tecnico - gestionale e distribuzione: Domenico Asinelli
Bull Italia, via Tazzoli 6
20154 Milano, Tel. 02 - 67792960

UNIVERSITA' DEGLI STUDI DI MILANO
DIPARTIMENTO DI SCIENZE DELL'INFORMAZIONE

BULL ITALIA
CENTRO STUDI INFORMATICA E AUTOMAZIONE

QUESTO NUMERO

Uno degli orientamenti di fondo dell'attuale fase di sviluppo dell'informatica è certamente il trend verso sistemi operativi standard. In questo quadro, un posto di indubbia rilevanza è assegnato a UNIX, un fenomeno a livello mondiale, potenziale piattaforma dei "sistemi aperti" degli anni 90.

E' sembrato perciò opportuno dedicare a questo tema un numero monografico di " Note di Software". Nella scelta dei contributi, proprio in relazione alla natura dell'argomento, si è dato spazio a lavori di provenienza sia accademica che industriale, nei quali è possibile vedere l'impegno, sia della Bull che del Dipartimento di Scienze dell'Informazione, speso in questa direzione e manifestato mediante la stesura di numerose tesi di laurea in comune.

In apertura, Gianfranco Coltorti presenta una panoramica di sistemi UNIX intermini di mercato. Tutte le operazioni concordano sul ruolo rapidamente crescente di tali sistemi nello scenario informatico.

L'articolo successivo, di Vittorio Cajani e Angelo Ramolini, illustra i resultati di uno studio approfondito condotto dagli autori sulle architetture multiprocessor a base UNIX. Impiegando un coerente approccio metodologico, vengono confrontate diverse soluzioni alternative ed esposti i criteri che hanno portato alla scelta della architettura adottata per un sistema messo recentemente sul mercato mondiale.

L'ottimazione delle prestazioni costituisce il tema anche del terzo articolo. Nicoletta Airenti, Carlo Leonardi e Paolo Zavattarelli presentano Easytune, uno strumento da essi realizzato per l'analisi in tempo reale delle prestazioni di un sistema multiprocessor UNIX.

Nel quarto articolo, Le Van Hu, Paola Rossaro e Carlo Leonardi presentano un algoritmo per il bilanciamento del carico utente di sistemi multiprocessor UNIX, attraverso il trasferimento di processi tra le varie CPU.

Nell'ultimo articolo, Giuseppe Facchetti illustra le caratteristiche di Mach, Kernel di sistema operativo originato in ambito accademico, e le "RECENSIONI", con una selezione di recenti uscite librarie.

LA REDAZIONE

PROSPETTIVE DI MERCATO DEI SISTEMI UNIX

Gianfranco Coltorti
Bull XS
Milano

RIASSUNTO

Nel corso degli ultimi dieci anni UNIX è diventato un sistema operativo di primaria importanza a livello mondiale e, sotto la spinta dell'Amministrazione Statunitense nonchè della maggioranza dei grandi utenti, s'appresta a diventare la piattaforma architetturale dei sistemi aperti degli anni '90. L'analisi del mutamento che ha caratterizzato il passato, e la discussione sulle prospettive future, sono sviluppate con l'ausilio di stime e previsioni di mercato di un consulente di rango mondiale.

ABSTRACT

During the last ten years UNIX has achieved the role of a world-class operating system and in the future, mainly due to the endorsement of the U.S. Administration together with the drive coming from large users, is going to become the open architecture platform for the 1990's. The analysis of UNIX past development and the discussion over its expected evolution has been complemented with historical and forecast data worked out by a leading consultant.

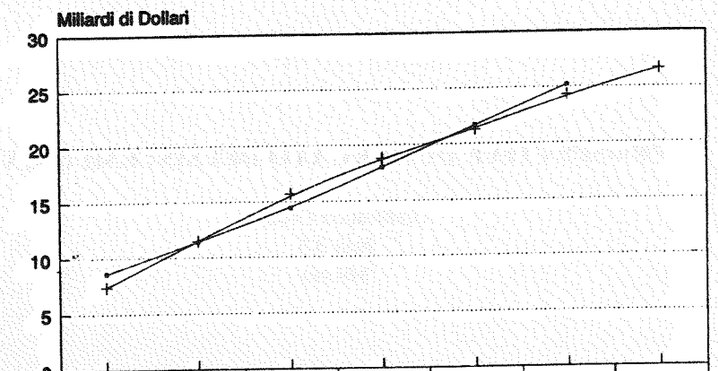
1. INTRODUZIONE

Nel corso degli ultimi dieci anni UNIX è passato dal rango di oscuro sistema operativo conosciuto quasi esclusivamente nel mondo accademico americano, alla posizione di sistema operativo di primaria importanza a livello mondiale, e s'appresta a diventare la piattaforma architetturale dei sistemi aperti degli anni 90.

La portabilità delle applicazioni e l'indipendenza dall'hardware sono le caratteristiche che gli hanno guadagnato in un primo momento l'attenzione del mercato. In seguito, lo sforzo di molte organizzazioni, per aggregare consenso intorno agli standard, l'hanno confermato come realtà.

La discussione intorno al mutamento che ha caratterizzato il passato, e quella sulle prospettive future, non sembra possano prescindere da una qualche misura a consuntivo del mercato e dai dati previsionali più autorevoli finora pubblicati. Sfortunatamente le società di consulenza che pubblicano analisi quantitative globali del mercato, e che le aggiornano regolarmente, sono poche. Fra queste è la InfoCorp di Santa Clara, CA (oggi parte della Gartner Group di Stanford, CT) che ha regolarmente pubblicato stime e previsioni del mercato dei sistemi UNIX dal 1985.
Qui useremo i suoi numeri. La ricchezza della segmentazione e l'articolazione delle categorie sono state determinanti per la loro scelta.
Un'altra società le cui analisi quantitative avrebbero potuto essere qui parimenti adottate è la Dataquest di San Jose, CA le cui prime pubblicazioni in argomento risalgono al 1986. Conforta constatare, che almeno a livello aggregato, i numeri pubblicati dalle due società,

MERCATO MONDIALE dei Sistemi UNIX
CONFRONTO INFOCORP-DATAQUEST
(Valore delle vendite di Hardware)



	1987	1988	1989	1990	1991	1992	1993
DATAQUEST (9.88)	8.6	11.4	14.4	18	21.7	25.2	
INFOCORP (1.89)	7.4	11.5	15.7	18.8	21.3	24.3	26.6

Fig. 1

elaborati in modo da renderli confrontabili, non si discostino macroscopicamente gli uni dagli altri. Il confronto è in figura 1.

I numeri riportati in tutte le figure e tabelle seguenti si riferiscono alle spedizioni sul mercato mondiale dei sistemi con prezzo tipico, al momento della vendita iniziale[1], non inferiore ai 3,000 dollari. Restano quindi esclusi i personal computer piu' piccoli, in quanto estranei, almeno per il periodo di previsione, che va fino al 1993, al mondo UNIX [2]. Il mercato è misurato in termini di *Initial If Sold Value*, cioè di quanto il cliente finale effettivamente paga, in dollari correnti, per hardware e software al momento dell'acquisto primario[3].

[1] Nella letteratura americana si parla di *Average Selling Price* che include il prezzo dell'hardware, quello del software di sistema ed eventualmente il prezzo del software applicativo, effettivamente pagati, al netto di sconti, dal cliente finale al momento dell'istallazione. Ai fini della categorizzazione non si considera quindi il valore di quanto acquistato successivamente per ampliamenti: non viene qui considerato il valore medio del sistema nella sua configurazione più tipica, ma il prezzo medio pagato al momento dell'acquisto iniziale.

[2] Il mercato dei sistemi con *Average Selling Price* inferiore ai 3,000 dollari valeva nel 1979 meno del 2% del mercato globale. La percentuale si è assestata intorno all'11% nel 1980, e si prevede che superi il 16% nel 1993. La scelta di limitare l'analisi al mercato dei sistemi con ASP di almeno 3,000 dollari, nonostante la rilevanza del segmento complementare, è quindi dovuta unicamente al tema:il mercato dei sistemi UNIX.

[3] Avendo riguardo ai ricavi delle aziende, sarebbe stato più opportuno usare come misura il *Total Market Value* (TMV) che oltre all'*Initial If Sold Value* (IISV) include il valore degli ampliamenti (hardware e software) venduti successivamente, nonché il valore dei servizi e dei prodotti accessori. Come si intuisce però, questa categoria è legata alla consistenza del parco istallato e risulta quindi poco adatta a misurare fenomeni caratterizzati da un tasso d'espansione diverso da quello medio del mercato. Per avere comunque un'idea del rapporto fra queste categorie di valori si consideri che oggi, a livello del mercato globale, l'Initial If Sold Value è stimabile intorno al 61% del Total Market Value, le vendite di ampliamenti sono circa il 12%, e le vendite di servizi e prodotti accessori il 27%.
L'evoluzione del mercato nei prossimi cinque anni si prevede farà scendere il rapporto IISV/TMV al 56% nel 1993, principalmente per l'aumento fisiologico dei ricavi legati al parco istallato: ricavi da manutenzione in primo luogo ma anche vendite di ampliamenti e di prodotti accessori; nonché, più in generale, per la crescente rilevanza del peso relativo dei servizi.

2. UNO SGUARDO AL PASSATO

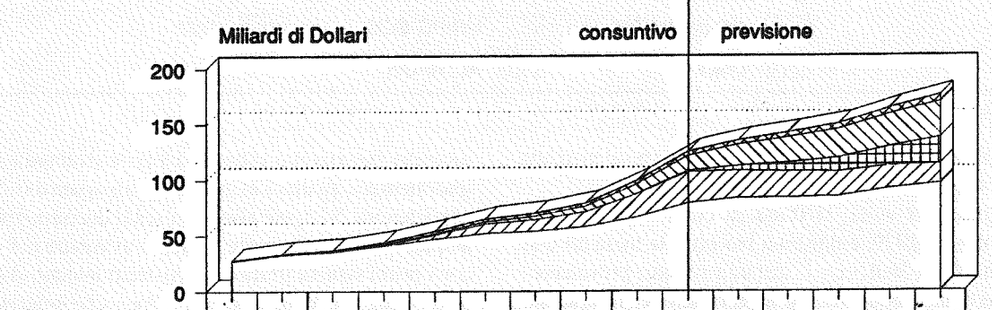
La prima licenza d'uso concessa da AT&T al di fuori del mondo accademico risale al 1976. Il prodotto, la *UNIX Programmer Workbench*, che per disposizioni federali non poteva essere supportata nè promossa da AT&T, aveva però un prezzo troppo elevato per dare l'avvio ad un mercato di massa.

E' invece del 1979 il primo passo concreto che ha dato inizio alla diffusione di UNIX. In quell'anno AT&T introdusse politiche di sconto che resero interessante il prezzo della *Version 7*. Questa decisione cadde su un terreno reso fertile da altre due circostanze positive: l'uscita dei primi microprocessori a 16 bit e la disponibilità sul mercato del lavoro dei primi diplomati che avevano avuto modo di conoscere UNIX all'università. Diveniva così possibile intraprendere attività concrete per colmare la lacuna lasciata da AT&T nelle attività, ancora a lei precluse, di marketing e supporto del prodotto, nonchè iniziare la progettazione di sistemi che potessero competere con i minicomputer esistenti, utilizzando microprocessori ormai di potenza adeguata.

In questi primissimi anni comunque, in assenza di un impegno preciso di AT&T, che verrà solamente con la riorganizzazione del 1983, il mercato dei sistemi UNIX cresce quasi esclusivamente in virtù dell'entusiasmo suscitato dalle caratteristiche intrinseche del sistema operativo: i benefici legati alla portabilità in primo luogo, e l'approccio, diciamo così, filosofico di programmare in piccole unità modulari, generalizzate quanto possibile, da usare per costruire moduli più complessi.

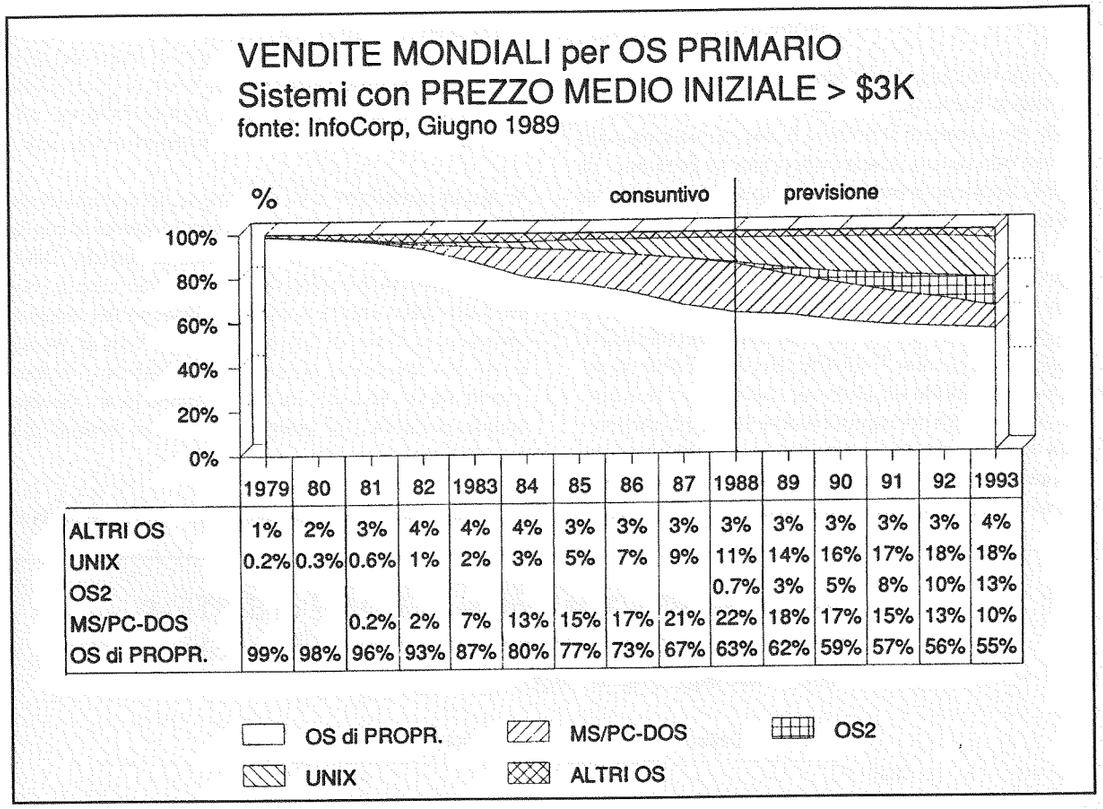
Le figure 2 e 3 danno la misura del valore dei sistemi

VALORE MONDIALE delle VENDITE di SISTEMI
Sistemi con PREZZO MEDIO INIZIALE > $3K
fonte: InfoCorp, Giugno 1989



	1979	80	81	82	1983	84	85	86	87	1988	89	90	91	92	1993
ALTRI OS	0.4	0.5	1.1	1.6	2.4	2.6	2.6	2.7	3.2	3.7	3.8	4.3	4.8	5.3	6.1
UNIX	0.05	0.1	0.2	0.6	1.1	2	3.3	5.2	8.8	13.7	18.9	22.7	25.7	29.3	32.1
OS2										0.9	4.3	7.7	12.3	17.1	23.5
MS/PC-DOS			0.1	0.7	3.6	8.4	10.3	13.2	21	28.1	24.7	23.7	22.4	20.3	17.9
OS di PROPR.	27.2	32.8	35.3	41	46.9	52	53.8	57.8	66.2	78.7	83.4	83.7	84.7	91.8	96

Fig. 2



	1979	80	81	82	1983	84	85	86	87	1988	89	90	91	92	1993
ALTRI OS	1%	2%	3%	4%	4%	4%	3%	3%	3%	3%	3%	3%	3%	3%	4%
UNIX	0.2%	0.3%	0.6%	1%	2%	3%	5%	7%	9%	11%	14%	16%	17%	18%	18%
OS2										0.7%	3%	5%	8%	10%	13%
MS/PC-DOS			0.2%	2%	7%	13%	15%	17%	21%	22%	18%	17%	15%	13%	10%
OS di PROPR.	99%	98%	96%	93%	87%	80%	77%	73%	67%	63%	62%	59%	57%	56%	55%

Fig. 3

spediti in quegli anni e il loro peso relativo sul mercato globale.
In termini di unità, si tratta di circa 1,600 sistemi prodotti nel 1979, e destinati per oltre la metà all'uso interno della Bell, con il restante spedito prevalentemente a università o all'Amministrazione Americana. Si stima che in quell'anno solamente cinquanta sistemi andarono al mercato commerciale vero e proprio. Nel 1980 le spedizioni raggiunsero i 3,700 sistemi per passare a circa 9,000 nel 1981 e quindi a 29,000 del 1982. Questi ultimi si distribuirono in maniera già fondamentalmente diversa rispetto al 1979 in quanto più della metà furono assorbiti dal mercato commerciale.

Altos, Fortune, Tandy, Zilog sono i primi nomi che vengono in mente ad esemplificazione di quella schiera di pionieri che annunciarono e spedirono sistemi UNIX già nel 1982. Per le dimensioni modeste dei soggetti, la vendita ed il supporto, e quindi lo sviluppo stesso del mercato, poggiarono allora quasi esclusivamente sugli intermediari, gli unici in grado di sviluppare e vendere applicazioni. Si ripetè allora, su scala decisamente più ampia, qualcosa di simile a quanto avvenuto nel 1972 con i primi sistemi gestionali, costruiti intorno al *Business BASIC* da Wang e da Basic Four, che intaccarono per la prima volta il monopolio dell'offerta globale delle grandi compagnie. Come allora, il mercato beneficiò dell'acquisizione di nuove fasce d'utenza, grazie alla ricaduta delle conoscenze maturate nelle università.

In questo periodo la confusione era notevole soprattutto per la mancanza di standard: ogni versione del sistema operativo era incompatibile con la precedente, e comunque diversa da quella usata internamente da AT&T; la facilità di apportare modifiche induceva molti licenziatari ad elaborare versioni proprie di UNIX. Fra le più significative tra queste sono da ricordare le BSD 4.1 e

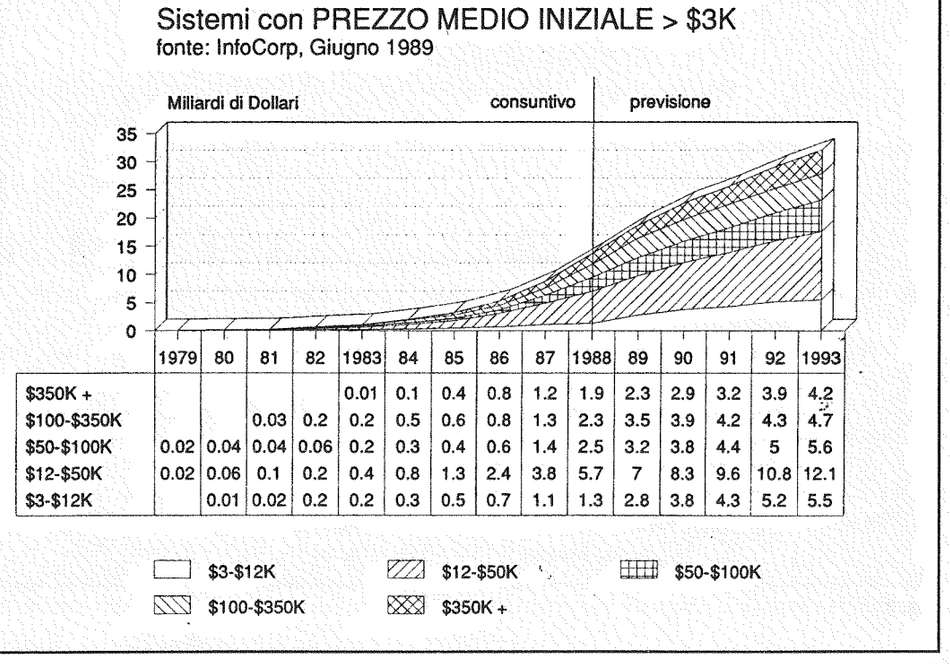
4.2 dell'Università di California a Berkeley che nascono in questo periodo, e la versione Xenix della Microsoft annunciata nell'agosto del 1980.

Il 1983 è un anno di svolta. Dopo la riorganizzazione, in gennaio, AT&T annuncia il *System V* impegnandosi a standardizzare il prodotto e a mantenere, da quel momento in poi, la compatibilità fra vecchie e nuove release. Inizia anche una campagna promozionale intensiva e in maggio annuncia l'impegno ad implementare il System V sui microprocessori più diffusi di Intel, Motorola, National Semiconductor e Zilog. In giugno offre i primi corsi dando concreto inizio all'attività di supporto.

Come conseguenza, molti nuovi produttori annunciano o preannunciano prodotti basati su UNIX. Convergent Technologies, N.C.R., Pyramid, Sun Microsystems, per non citarne che alcuni, che in seguito si distingueranno per l'impegno, sono fra i circa ottanta, individuati già ad aprile 1983, che operano o che intendono operare sul nuovo mercato. Alla fine del 1988 InfoCorp ha contato a livello mondiale 113 produttori maggiori che avevano venduto sistemi UNIX durante quell'anno. Fra questi è doveroso ricordare la Bull H.N. Italia che ha iniziato a sviluppare a Pregnana Milanese sistemi UNIX nel 1984 spedendo le prime unità nel 1985.

Dal 1983 al 1988 le spedizioni di sistemi UNIX crescono in valore ad un tasso medio annuo composto del 67%, contro il 18.3% dell'intera industria EDP. In termini di unità si passa dai circa 45,000 sistemi spediti nel 1983 agli oltre 440,000 che si stima siano stati spediti lo scorso anno. Alla fine del periodo l'Initial If Sold Value spedito è valutato intorno ai 13.7 miliardi di dollari (figura 2).

Il mercato è sostenuto dalla percezione ormai chiara dei vantaggi legati alla portabilità e dai riconoscimenti, come quello autorevole di X/Open e dell'Amministrazione Statunitense, che UNIX è il più rispondente ai requisiti. La disponibilità di strumenti di sviluppo che



	1979	80	81	82	1983	84	85	86	87	1988	89	90	91	92	1993
$350K +					0.01	0.1	0.4	0.8	1.2	1.9	2.3	2.9	3.2	3.9	4.2
$100-$350K			0.03	0.2	0.2	0.5	0.6	0.8	1.3	2.3	3.5	3.9	4.2	4.3	4.7
$50-$100K	0.02	0.04	0.04	0.06	0.2	0.3	0.4	0.6	1.4	2.5	3.2	3.8	4.4	5	5.6
$12-$50K	0.02	0.06	0.1	0.2	0.4	0.8	1.3	2.4	3.8	5.7	7	8.3	9.6	10.8	12.1
$3-$12K		0.01	0.02	0.2	0.2	0.3	0.5	0.7	1.1	1.3	2.8	3.8	4.3	5.2	5.5

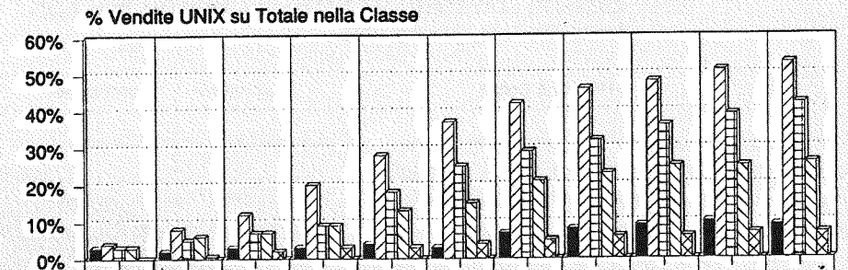
Fig. 4

permettono di scrivere software di qualità guadagnano a UNIX l'attenzione dei grandi utenti - si pensi alla General Motors e all'industria automobilistica americana in genere - portando ulteriori stimoli per i produttori a proseguire sulla strada del suo arricchimento funzionale.

Anche l'offerta di software inizia ad essere stimolata dalle dimensioni del mercato: software orizzontale soprattutto. Verso la fine del periodo su UNIX erano già disponibili i migliori *Relational Data Base Management Systems*, un'ampia scelta di software per *Office Automation*, e un insieme di *Communications tools* fra i più ricchi dell'industria. L'abbondanza e la qualità di questo software è anche oggi uno dei punti di forza dell'offerta UNIX.

Nonostante il persistere d'incertezze gravi che provocheranno il mutamento avviato nel 1988, la penetrazione globale, in valore, s'avvicina all'11% alla fine del periodo (figura 3). Il mercato assorbe per lo più sistemi multi-utente (figura 6)[4] sebbene la percentuale di stazioni individuali UNIX sia in continua crescita. I sistemi più piccoli vengono venduti prevalentemente per applicazioni gestionali (figura 7) mentre i sistemi più costosi vanno alle applicazioni tecnico-scientifiche e all'insegnamento. Gartner Group l'anno scorso stimava che il rapporto prestazioni/prezzo dei sistemi UNIX era migliore di quello dei sistemi di proprietà per circa il 60-70 per cento e che soprattutto i costi di fruizione (*cost of ownership*) erano decisamente più bassi. Ciò spiega la penetrazione di UNIX sul mercato delle applicazioni gestionali, mentre sembra indubbio che la crescente preferenza accordatagli dal mondo della ricerca e da quello accademico sia da accreditare ai meriti intrinseci del prodotto. Le stime di InfoCorp sembrano comunque indicare un progressivo livellamento fra questi due tipi fondamentali d'impiego.

PENETRAZIONE Sistemi UNIX
per Classe di Prezzo
fonte: InfoCorp, Giugno 1989



	1983	1984	1985	1986	1987	1988	1989	1990	1991	1992	1993
$3-$12K	3%	2%	3%	3%	4%	3%	7%	8%	9%	10%	9%
$12-$50K	4%	8%	12%	20%	28%	37%	42%	46%	48%	51%	53%
$50-$100K	3%	5%	7%	9%	18%	25%	29%	32%	36%	39%	42%
$100-$350K	3%	6%	7%	9%	13%	15%	21%	23%	25%	25%	26%
$350K +		0.5%	2%	3%	3%	4%	5%	6%	6%	7%	7%

$3-$12K $12-$50K $50-$100K
$100-$350K $350K +

Fig. 5

[4] Le figure 6, 8 e 9 dettagliano l'analisi solamente per i sistemi con prezzo tipico sotto i 100,000 dollari in quanto al di sopra di questa soglia, nella quasi totalità, i sistemi sono multi-utente.

L'analisi per classi di prezzo evidenzia che le vendite hanno interessato prevalentemente i sistemi con un prezzo tipico iniziale inferiore ai 100,000 dollari. In questo ambito la sottoclasse dai 12,000 ai 50,000 dollari si è distinta sia per il valore aggregato (figura 4) che per la penetrazione (figura 5). Un ultimo rilievo: la penetrazione di UNIX fra i sistemi multiutente è risultata inversamente proporzionale al prezzo dei sistemi stessi (figura 8), mentre nell'ambito delle stazioni individuali più potenti, dove UNIX costituisce ormai lo standard di fatto, si è creata una situazione diametralmente opposta (figura 9).

Le misure a consuntivo evidenziano le strategie dei produttori volte ad allargare il mercato verso il basso, con il contributo degli intermediari, e confermano le resistenze al mutamento delle fasce più alte della domanda, causate verosimilmente dagli investimenti più consistenti già effettuati sui sistemi di proprietà.

3. IL FUTURO

Gli avvenimenti politico-relazionali dello scorso anno, trovano le loro radici in preoccupazioni e problemi emersi fin dai primi anni '80. Utenti e produttori hanno infatti sempre considerato con preoccupazione la prospettiva di legare i propri investimenti ad un ambiente operativo le cui specifiche rimanevano nelle mani di un solo soggetto, che per giunta considerava riservata ogni informazione sugli sviluppi futuri. Parimenti, il problema di giungere a qualche forma di standardizzazione che assicurasse l'effettiva portabilità delle applicazioni fra sistemi diversi emerse fin dal primo momento dando luogo ad iniziative di rilievo: Lo *UNIX/usr/group Standards Committee* nasce nel 1981, e dal suo progetto di standardizzazione, sottoposto all'IEEE (*Institute of Electrical and Electronics Engineers*) nel 1984, nascerà il comitato *P1003 Working Group on Operating System Kernel Standard* (POSIX); nello stesso anno in Europa viene formato X/Open, con lo scopo dichiarato di promuovere un insieme di standard che definiscano l'ambiente applicativo (*Common Application Environment*) onde dar vita a sistemi aperti. X/Open sceglie UNIX come sistema operativo di riferimento, e in particolare lo standard POSIX. (Per contro non sembra inutile ricordare che AT&T annuncerà solamente nel 1987 l'intenzione di far convergere System V verso lo standard POSIX.)

L'annuncio di *Open Software Foundation* (OSF), dopo un primo momento di disorientamento, viene percepito come l'ingresso di un nuovo concorrente ed accolto favorevolmente dal mercato. Viene particolarmente apprezzata la dichiarazione che il prodotto sarà conforme allo standard POSIX e alla *Portability Guide* pubblicata da X/Open.
Avendo, in seguito, anche AT&T annunciato che la release 4 del System V sarà conforme ai criteri X/Open, il mercato viene a trovarsi nella felice condizione di vedere finalmente avviato a soluzione il problema più importante individuato fin dai primi anni '80. Parimenti la decisione di AT&T di mettere a parte il mercato degli sviluppi futuri, promuovendo *Unix International Inc.* (UII), porta ulteriore rassicurazione.

Lo scenario che inizia così a delinearsi lascia spazio ad aperture ulteriori, ancora nel campo delle possibilità, ma con un'alta probabilità di realizzarsi: OSF e UII, guidati dal mercato per la cui attenzione competono entrambi, potrebbero arrivare ad una convergenza dei loro prodotti almeno al livello dell'interfaccia utente e di quella applicativa, e X/Open potrebbe interpretare un ruolo importante in questo processo con i suoi test di conformità e verifica. A questo punto un robusto sistema di base, con il marchio di conformità X/Open, dovrebbe stimolare ulteriormente gli sviluppatori di software, che sono interessati alla massima portabilità, a investire in nuove applicazioni.
Rimarrebbe sempre spazio per estensioni e miglioramenti a scapito della portabilità, ma queste resterebbero pratica confinata alle strategie dei singoli produttori.
In questo scenario, la specializzazione nella produzione di strumenti complementari di software diventerebbe criterio degno di considerazione anche per i massimi protagonisti: in altre parole, difficilmente OSF e UII, in presenza di prodotti validi, avrebbero convenienza ad investire per implementare, ad esempio, un proprio Data Base relazionale o la propria System Network Architecture. Una maggiore chiarezza nell'assunzione concreta dei ruoli, e quindi nella distribuzione degli investimenti, si tradurrebbe in un'accelerazione del processo di standardizzazione di fatto che il mercato aspetta.

Rimane indubbio comunque che il futuro poggia ora su premesse più sicure.
Positive sono state anche le scelte tecniche annunciate

a valle degli avvenimenti dello scorso anno. Mi riferisco alla decisione di AT&T di consolidare, già nella release 3 del System V, Xenix e BSD 4.2 (le due deviazioni piu' popolari di UNIX) e soprattutto al lavoro intrapreso per permettere la distribuzione delle applicazioni in codice binario, come avviene già ad esempio per i sistemi MS-DOS. Questa limitazione ha avuto nel passato, e ha tuttora, un effetto paralizzante sul mercato al dettaglio delle applicazioni e inoltre, dirottando l'attenzione degli sviluppatori di software verso altri ambienti, ha di fatto frenato l'intero mercato dei sistemi UNIX.
UII, che distribuisce già il suo software con le *Application Binary Interfaces* (ABI) AT&T 3B/2, Intel 80386, e Sun Microsystems SPARC ha annunciato che *System V 4.0* verrà distribuito anche con ABI Motorola 68000, 88000 e Mips Computer Systems R2000 e R3000.OSF, da parte sua, ha già emesso una *Request For Technology* (RFT) per un *Architecture-Neutral Distribution Format* (ANDF), ed è noto che contemporaneamente tutti i maggiori produttori di microprocessori stanno lavorando, in coordinazione con lei, alle ABI di loro interesse.

I benefici potenziali di tutto questo lavoro iniziano già ad emergere: Gartner Group, la cui clientela consiste all'80% di utenti legati ai sistemi operativi di proprietà più diffusi, ha iniziato a consigliare i propri clienti di "*includere UNIX in ogni piano strategico in elaborazione*"[5], "*onde evitare di trovarsi in sicuro svantaggio nell'ambito di 18-24 mesi*"[6]. Aspettative parallele influenzano i piani di lavoro degli sviluppatori d'applicazioni.

Ecco che per il periodo 1988-1993 InfoCorp stima che il mercato dei sistemi UNIX cresca del 18.6% annuo composto in valore (contro il 7% dell'intera industria EDP), superando i 32 miliardi di dollari di Initial If Sold Value alla fine del periodo (figura 2). La penetrazione globale in valore dovrebbe superare il 18% (figura 3). La crescita in unità spedite dovrebbe aggirarsi intorno al 26% annuo composto, approssimandosi a 1,400,000 sistemi nel 1993, contro poco più dell'8% stimato per l'intera industria.

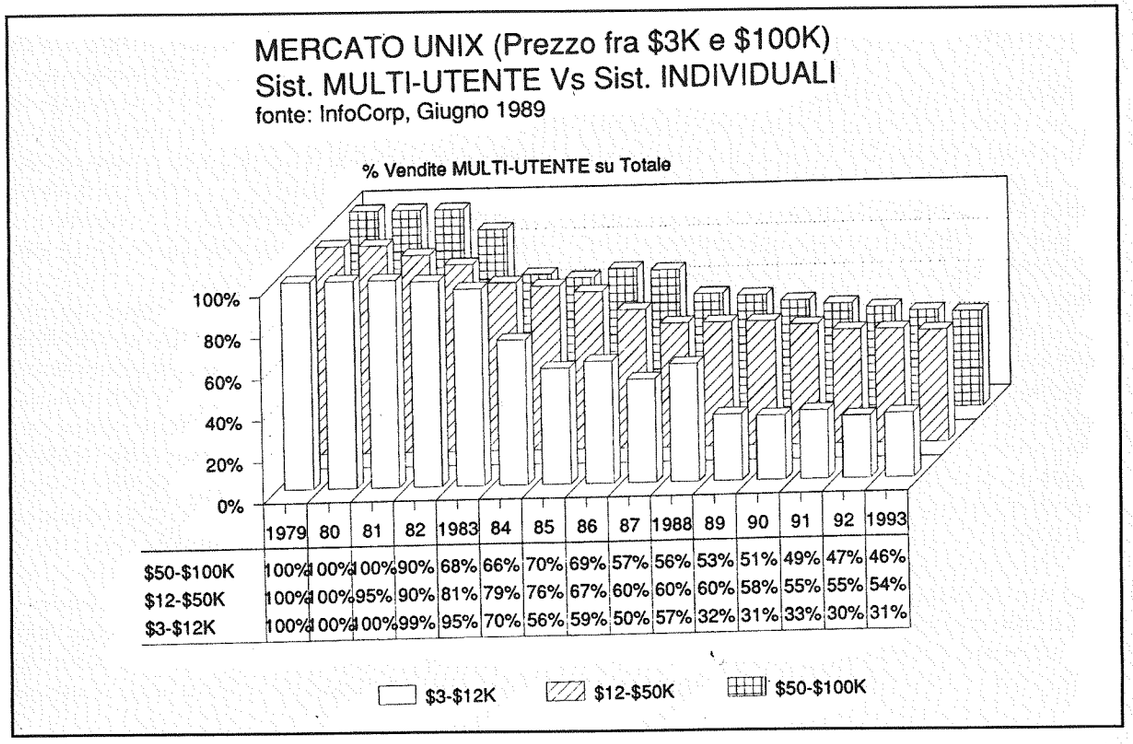


Fig. 6

[5] Gartner Group: Seventh Annual Small Computer Systems Conference, 13-15 aprile 1988, San Diego CA - Atti.

[6] Gartner Group: Eight Annual Small Computer Systems Conference, Midrange Computing in the 1990's, 7-9 giugno 1989, Monterey CA - Atti.

Nel segmento dei sistemi con prezzo tipico dai 12,000 ai 100,000 dollari, si prevede che già nel 1992, più della metà delle spedizioni siano UNIX.
In questo ambito la sottoclasse più ricca, in termini di ricavi aggregati, dovrebbe rimanere quella dei sistemi dai 12,000 ai 50,000 dollari di prezzo tipico: più di 12 miliardi di dollari previsti per il 1993 (figura 4). E la sottoclasse dovrebbe rimanere anche la piu' importante in termini di penetrazione globale. La stima vuole che il 53% dell'Initial If Sold Value totale spedito nel 1993 sia da accreditare ai sistemi UNIX (figura 5).
Nel loro ambito è previsto che si continui ad avere qui una prevalenza dei sistemi multi-utente (figura 6). Questi, rapportati a tutte le spedizioni di sistemi multi-utente nella classe, dovrebbero avere una penetrazione del 44% nel 1993 (figura 8). Le stazioni individuali, per contro, pur aggregando solamente il 46% dell' Initial If Sold Value dei sistemi UNIX nella classe, (il riferimento è sempre alla previsione 1993) dovrebbero raggiungere, rapportate a tutte le stazioni individuali nella classe, una penetrazione del 69% (figura 9).

La seconda classe di prezzo per importanza, dell'intero mercato UNIX, è quella dei sistemi con valore tipico iniziale dai 50,000 ai 100,000 dollari. La penetrazione globale dovrebbe raggiungere qui il 42% nel 1993 grazie soprattutto alle stazioni individuali più potenti che si prevede porteranno a UNIX, alla fine del periodo di previsione, il 77% delle vendite di tutte le stazioni individuali nel segmento. Per contro i sistemi multiutente UNIX dovrebbero aggiudicarsi nello stesso anno il 27% di tutte le vendite di sistemi multi-utente nella classe.

La figura 4 ipotizza una buona crescita anche per i

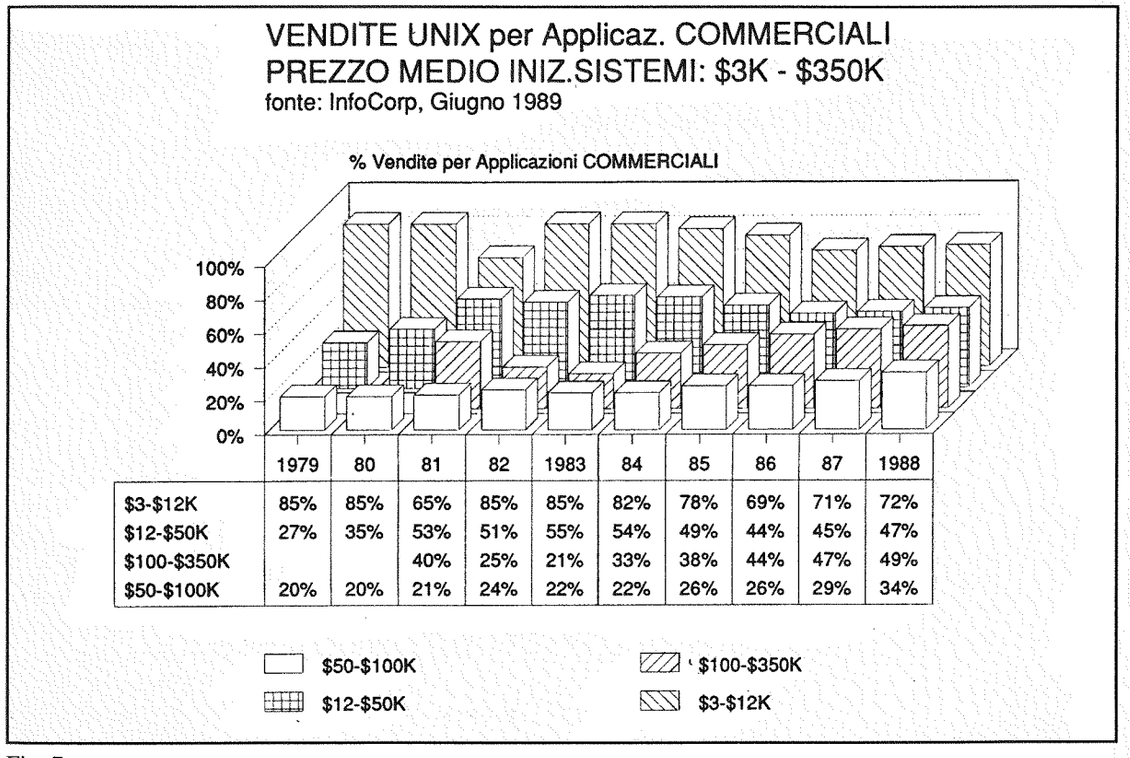


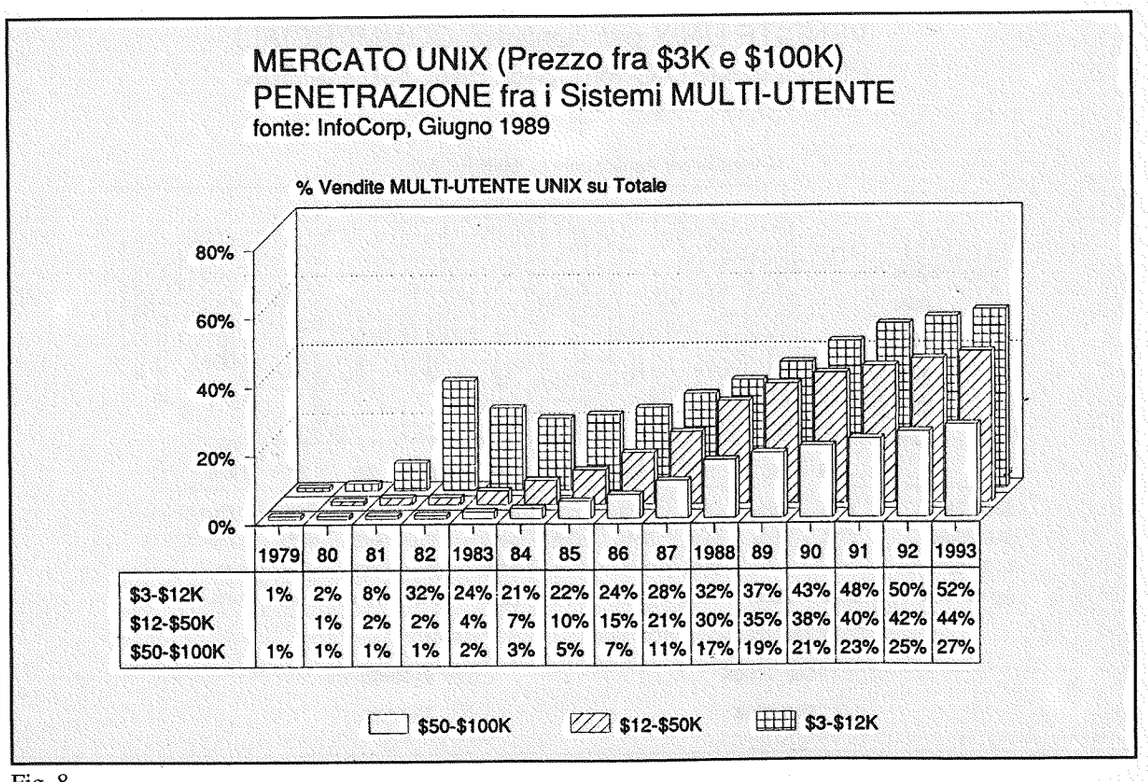
Fig. 7

sistemi UNIX con prezzo tipico dai 3,000 ai 12,000 dollari. In termini di composizione della domanda, dalla figura 6 si desume anche che InfoCorp ha previsto, per questi prodotti, una netta inversione di tendenza: a partire da quest'anno le stazioni individuali dovrebbero aggregare una percentuale di ricavi decisamente più alta dei sistemi multi-utente. A complemento le figure 8 e 9 chiariscono ulteriormente la posizione del consulente: i sistemi UNIX multi-utente in questa classe sembrano destinati a prevalere su tutti gli altri sistemi multi-utente di pari prezzo, mentre le stazioni individuali UNIX, pur essendo in forte crescita, non dovrebbero superare, come penetrazione in valore, il 7% alla fine del periodo di previsione.L'assunzione esplicita è che gli ambienti MS-DOS, OS/2 e Macintosh continueranno ad essere di gran lunga i dominanti, fra le stazioni individuali, in questa classe di prezzo. (Di qui, anche la scarsa penetrazione globale che emerge dalla figura 5). Ma le analisi degli sviluppi tecnologici in corso suggeriscono anche ipotesi ragionevoli diverse, e le previsioni di InfoCorp, per il mercato dei sistemi UNIX in questa classe di prezzo, potrebbero rivelarsi estremamente conservative.

Oggi, in effetti, con OS/2 e Macintosh System 7.0, si cerca di dotare i Personal Computer di funzionalità proprie delle stazioni UNIX, e con le interfacce utente tipo *Open Look e Motif* si cerca di dare a queste ultime le caratteristiche più proprie dei Personal Computer. Da questo processo di convergenza, UNIX non potrà che avere vantaggi essendo il solo sistema operativo oggi sul mercato ad offrire un ambiente di lavoro unico ed omogeneo, dalle stazioni individuali ai sistemi dipartimentali, fino ai supercomputer: l'unico che prometta di far scegliere la macchina, in funzione della potenza richiesta dall'applicazione, al momento di operare (*scalability*).

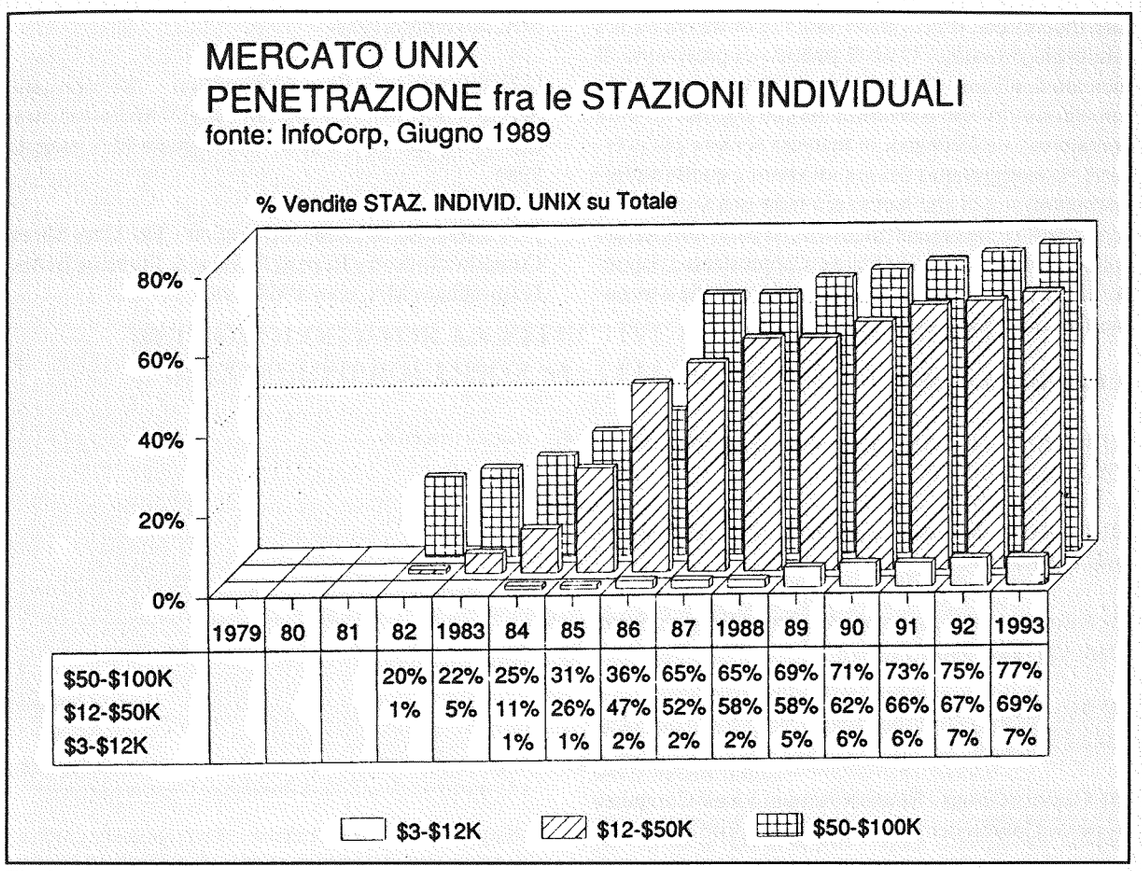
Comunque, indicazioni più sicure sulla reazione del mercato al processo di convergenza in corso non dovrebbero tardare molto.

Le previsioni per i sistemi UNIX più potenti e costosi non sembrano esaltanti nè come valore delle vendite nè



	1979	80	81	82	1983	84	85	86	87	1988	89	90	91	92	1993
$3-$12K	1%	2%	8%	32%	24%	21%	22%	24%	28%	32%	37%	43%	48%	50%	52%
$12-$50K		1%	2%	2%	4%	7%	10%	15%	21%	30%	35%	38%	40%	42%	44%
$50-$100K	1%	1%	1%	1%	2%	3%	5%	7%	11%	17%	19%	21%	23%	25%	27%

Fig. 8



	1979	80	81	82	1983	84	85	86	87	1988	89	90	91	92	1993
$50-$100K				20%	22%	25%	31%	36%	65%	65%	69%	71%	73%	75%	77%
$12-$50K				1%	5%	11%	26%	47%	52%	58%	58%	62%	66%	67%	69%
$3-$12K						1%	1%	2%	2%	2%	5%	6%	6%	7%	7%

Fig.9

come penetrazione, anche se il tasso medio annuo composto di crescita del valore del venduto è più del doppio di quello dell'intera industria EDP. In effetti sopra i 100,000 dollari di prezzo i sistemi di proprietà possono offrire ambienti di lavoro e di sviluppo ormai affermati, sfruttabili in tutte le loro funzionalità più sofisticate, e il peso degli investimenti effettuati dagli utenti su di essi non facilita il mutamento. Ma UNIX ha mostrato di incorporare le tecnologie emergenti più velocemente. Penso qui specificamente al *multiprocessing* e all'adozione dei microprocessori *Reduced Instruction Set Computing*, che insieme potrebbero mutare significativamente la curva del rapporto prestazioni/prezzo nella fascia alta del mercato, ma anche alle tecnologie di *Computer Aided Software Engineering* e agli *Expert Systems* ormai d'interesse per una fascia d'utenza sempre più ampia. Purtroppo il mondo UNIX manca ancora di standard per alcune funzionalità importanti nella fascia alta del mercato. Mi riferisco a *On Line Transaction Processing*, al *Real-Time* (anche se su questo argomento una prima pubblicazione di specifiche, da parte di IEEE, è prevista in tempi brevi), alle funzionalità di *security e* alla tecnologia dell'*High Availability*. Offerte UNIX di prodotti con queste funzionalità esistono già', ma le soluzioni disponibili sono ancora di proprietà, e gli sforzi di standardizzazione richiedono, per loro natura, tempi lunghi onde aggregare il consenso. Anche in questo segmento però il fermento è notevole e una piattaforma omogenea di sistemi aperti, come UNIX s'avvia a diventare, non può prescindere da una soluzione rapida dei problemi ancora aperti.

4. CONCLUSIONE

Il mercato ha finora risposto in maniera incoraggiante alla diffusione dei sistemi UNIX. Le previsioni pubbli-

cate dicono che il progresso sarà continuo anche nel futuro più prossimo. Oltre il periodo di previsione, il mercato degli anni '90 sarà verosimilmente caratterizzato dai sistemi aperti costruiti intorno a questo sistema operativo: lo sollecitano, in maniera sempre più pressante, la maggioranza dei grandi utenti e quelle amministrazioni statali che hanno già fatto una scelta in tal senso. I protagonisti dell'industria, da parte loro, hanno colto la sfida in maniera chiara e s'apprestano a rispondere con la rilevanza delle risorse economiche e tecnologiche che li caratterizza.

5. BIBLIOGRAFIA

[1] InfoCorp - Midrange Systems Service, Santa Clara CA, anni 1984-1989

[2] InfoCorp - Microsystems Service, Santa Clara CA, anni 1984-1989

[3] InfoCorp - Executive Market Trend Service, Santa Clara CA, anni 1984-1989

[4] InfoCorp - Software Service, Santa Clara CA, anni 1987-1989

[5] Gartner Group - Seventh Annual Small Computer Systems Conference, San Diego CA, aprile 1988, Atti

[6] Gartner Group - Eight Annual Small Computer Systems Conference, Monterey CA, giugno 1989, Atti

[7] Gartner Group - Seventh Annual Personal Computing Conference: Implementing the User Workbench, Monterey CA, giugno 1989, Atti

[8] Dataquest - Business Computer Systems Industry Service, San Jose CA, anni 1979-1989

[9] Dataquest - Technical Computer Systems Industry Service, San Jose CA, anni 1986-1988

[10] Dataquest - Software Industry Service - UNIX, San Jose CA,anni 1986-1987

[11] International Data Corporation - Software and Services Information Program - UNIX, Framingham MA, novembre 1984

[12] International Data Corporation - Software and Services Information Program - UNIX Migration Strategies of Supermini Users, Framingham MA, maggio 1985

[13] International Data Corporation - The Gray Sheet, Computer Industry Repert, Review & Forecast Issues, Framingham MA, anni 1968-1986

METHODOLOGY FOR AN EFFECTIVE COST/PERFORMANCE UNIX MULTIPROCESSOR SYSTEM IMPLEMENTATION.

Vittorio Cajani *, Angelo Ramolini **

RIASSUNTO

Questo articolo descrive l'architettura di un innovativo sistema UNIX multiprocessor, recentemente posto dalla BULL sul mercato mondiale.
Vengono brevemente descritte soluzioni architetturali alternative e illustrati i criteri che hanno portato a formulare questa scelta. Lo scopo è duplice: far meglio comprendere la portata innovativa della soluzione adottata e fornire i razionali per cui il rapporto costo/prestazioni di questo sistema si sta dimostrando estremamente competitivo.
Per necessità di sintesi l'articolo riporta solo una minima parte dei dati, degli algoritmi e delle metodologie utilizzate. L'intero materiale è comunque a disposizione di chi volesse approfondire l'argomento, con le limitazioni legate a copyright e brevetto.

ABSTRACT

This paper introduces and describes the innovative architecture of a UNIX Multiprocessor System recently introduced by BULL on the world-wide market.
Different architecture implementation alternatives are briefly described with all the related criterias used to compare, define and select the proposed implementation.
The objectives pursued throughout the paper are to clearly point out the intrinsic innovation of the proposed solution and precisely define its aggressive cost/performance ratio with the related competitiveness in the open market.
For conciseness sake, only the highlights of the methodology, algorithms and related data used in the analysis are included herein.
The full set of data is available, within copyright and patented restrictions,to verybody interested to have an in depth visibility of the subject.

* BULL HN Information Systems Italia - Milano
** BULL XS - Milano

[1] UNIX is a trademark of Bell Laboratories. In this paper we use the name UNIX to indicate an operating system which provides an interface which is POSIX or SVID or BSD4 and X/OPEN compliant.

1. INTRODUCTION

Till a couple of years ago, only few UNIX[1] boxes suppliers were offering multiprocessor systems[2]: these were mainly start-up companies, addressing niche market segments.
Today almost all the UNIX players have in their catalogue (or at least have plans to offer in a near future)

[2] In this discussion the term multiprocessor is used for systems which, in addition to dedicated (or functional) processors (designed to execute specific functions, like front-end processors to execute communication protocols and drive the communication lines and back-end processors, to carry out disk I/O operations and interface-disk channels), provides more than one central processing unit (CPU), able to simultaneously execute calculations for concurrent programs.

UNIX multiprocessor systems, which start having a key role in the UNIX market place.
The reasons for these UNIX upward evolution will not be discussed in this paper: the only consideration, which is relevant for the following discussion, is that the major target of UNIX multiprocessor systems is the connection of a large number of terminal users and/or the connection of a large number of client workstations, acting then as a powerfull computation or file server in a distributed system architecture.
Consequently, the UNIX system has to execute a great amount of concurrent processes, to carry out many simultaneous services and terminal applications.
On the other hand, the UNIX operating system was originally designed for monoprocessor H/W architectures and, even today, AT&T is licensing UNIX sources only for monoprocessor H/W architectures, so that each producer has to properly modify the UNIX [kernel] to make it efficiently executable on a multiprocessor system[3]. Incidentally, UNIX kernel can be enhanced to a large variety of multiprocessor architecture implementations, being designed for none of them. This is the today status. In a (near?) future the source code of a multprocessor kernel will be licensed by OSF and possibly also by AT&T. Already today the Carnegie Mellon University is delivering the MACH sources for multiprocessor systems. Some producers have already ported MACH [on commercial multiprocessor systems], implementing the UNIX interface on top of it[4].
For the above reason (lack of a UNIX S/W preferred multiprocessor architecture) the implementations of the UNIX multiprocessor systems today available/ announced differ very deply among themself. This paper will discuss some of them, focusing on the efficiency and cost/preformance trade-offs of the CPU/ Memory subsystem architecture.

2. ARCHITECTURE OBJECTIVES

Multiprocessor architectures have two major objectives:

scalability: a multiprocessor architecture allows to build-up, on the same H/W technology base and within the same CPU architectecture, a family of products that cover a large spectrum of possible configuration (computation power), providing ful compatibiltiy across the family [5]

configurability: the performances of a multiprocessor system are tunable according to the application specific needs and may be upgraded (up to the upper limit of the architecture) when the application re quirements are growing.

Ideally, the scalability of the system performances should then be achieved with the best granularity for every type of application processing requirement (calculation throughput, mass storage I/O throughput, communication throughput, etc).
To achieve that, multiprocessor [UNIX] systems typically provide specialized (or functional) processors, i.e. processors dedicated to execute specific functions, namely:

- CPU processors, to execute calculations
- Disk processors, to efficiently carry out mass storage I/O operations
- Communication processors, to connect communi - cation lines and execute communication protocols, downloading the CPUs from this low-level task.

[3] The major problem to be solved in order to let the UNIX run on a multiprocessor H/W, is related to the parallel execution of kernel routines (the UNIX delivered by AT&T provides only a serial execution of kernel services and disables the interrupts whenever a kernel routine accesses a data structure which could be inconsistently accessed by an interrupt handling routine). Refer to [1] for a description of the problem and of the most convinient way to solve it.

[4] MACH design and implementation overcomes the weak points of the AT&T UNIX kernel, still being capable to provide UNIX compliant interfaces.
Its major differantiating factors (with respect to AT&T and Berkley UNIX implemetations) are the improved portability, the capability to run on multiprocessor H/W, the suitability to distributed processing and the main memory desing (suited for large memories and programs with a very large address space). It was already ported on many systems. In particular it is the O.S. of the NEXT work station.

[5] Full compatibility means that all the products of the family have the same CRUs (customer replaceble units - likePWAs, drives, power supply modules, etc) and that, in addition to the source compatibility which is guaranteed via the adherence to the various official and defacto standards (POSIX, X/OPEN, etc), object programs can run on any machine of the product line (i.e. a recompilation is not required) and that the data rapresentation rules are identical on all the models, so that data files can be exchanged between the different models without any need of conversion.

The capability to provide more than one processor per each type of processing function, allows to optimize any system configuration. The system is assembled as an *incremental puzzle* , putting together the functional processors which provide the aggregate throughput that better approximates the application [throughput] requirements.

The following discussion will address the architecture of the CPU/memory subsystem, i.e. the subsystem which provides the calculation throughput and briefly highlights the modularity of the supporting environment (i.e. I/O subsystems and packaging). Being the objective to provide a competitive commercial system, the focus is on theprice/performance ratio, so that the most convinient architecture is that one with the best $/ MIPS figure (providing that the functional requirements are satisfied).

2.1 CPU/memory subsystem architecture objectives

The most advertized objective of a multiprocessor CPU subsystem is the capability to reach an aggregate CPU throughput which is almost linearly proportional to the number of CPUs connected to the system[6].
The first and obvious consideration is that to properly load n CPUs, at least n processes (or process threads) should be simultaneously executable on them. The following will assume that at any time there is enough processing threads to load all the available CPUs in the system; this assumption is generally true, due to the multiuser and client/server distributed environment targeted by theUNIX multiprocessor systems (see§1).
The second consideration is that a UNIX process executes both user code and kernel code. Considering that the ratio of kernel calculations [versus total calculations - i.e. user plus kernel] is application type[7] dependant and that typically has the following values:

Scientific Application		*OA, TP Application*
10%	⟶	**40%**

the UNIX kernel must provide a *multithreading* facility (i.e. the capability to be concurrently executed on several CPUs)to profit of a more than 2 CPU multiprocessor H/W, otherwise the serialization of kernel routines will be such a bottleneck that additional CPUs will not increase significantly the aggregate calculation throughput.
Nevertheless, due to the fact that this implies a major knowledgeable redesign of the UNIX kernel [1], some system builders have provided only a "*master/slave*[8]" implementation, with very negative results in many application environments.
The same considerations are applicable to the [horizontal] applications: they must be designed to be efficently executed on a multiprocessors architecture[9].
In one word, both the UNIX kernel and the application has to be *symmetric* , i.e. executable in parallel by many identical calculation processors. The following discussion is based on the hypotesis that the applications satisfy this requirement (as they generally do!).
Even with this basic hypotesys, the increase of performances (expressed in terms of aggregate CPU throughput) can not be directly proportional to the

[6] Expressing the throughput of a CPU in Mips (i.e. million of instructions [executed] per second), if one CPU is able to execute *y* Mips, a system configured with *n* CPUs will get up to *mxy* Mips, being *m* as closed as possible to *n* .

[7] In the following pages we will use the acronym OA for [advanced] Office Automation applications and TP for Transaction Processing and Information System applications

[8] In a master/slave approach, only one CPU processor (called *master*) is enabled to receive interrupts from the I/O processors and to execute [some] kernel routines.The other processors (*slave* , usually called also application processors) execute only user code, so that they require to the master processor the execution of [some] kernel services: in this way the modifications to the UNIX kernel are minimum, because the *semaphorization* of the kernel routines is not required.The brackets around "some" are indicating that some producers (above them ARETE' and EDGCORE) started offering master/ slave multiprocessor systems and lateron haveenhanced this approach by allowing the slave processors to execute some kernel routines and moving the execution of some driver code from the CPU into the I/O processor, thus limiting the master processor bottlenecks.

[9] Usually the time-sharing applications are suitable for multiprocessing, because for each user one or more independent processes are executed. Viceversa for TP or advanced office applications, when many concurrent users areaccessing the same set of resources (usually database files), the application must be carefully designed. If the application synchronizes the accesses to the shared resources via a serializationof the services (for example providing a single server process to execute all the database access), it will be facing the same problem as for the UNIX kernel master/slave multiprocessing: the CPU which executes the server process clearly becomes a bottleneck.

number of CPUs:

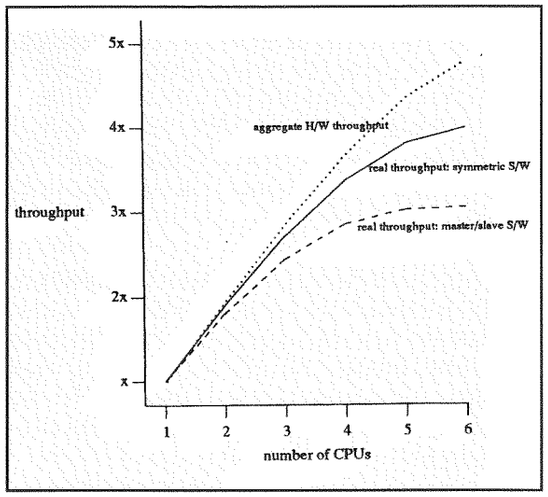


Fig.1 The aggregate CPU throughput

In the above figure the dotted line reflects the achievable aggregate H/W thrughput, i.e. it takes into account the H/W interferences between processors due to the concurrent access to shared H/W resources and the intrinsic communication bandwidth limits of the processor interconnetc network. The other two lines indicate the real throughput, i.e. the processing power really available for the execution of the application: it takes into account also the S/W interferences[10].

Additional penalizations must be considered, like limitations to the capability to properly schedule tasks on the available CPUs (see load balancing at § 4.2).
The discussion will be focalized on the achievable real throughput in a fully symmetric S/W implementation.

An other major requirement for the architecture is to provide *sharing of main memory data (i.e. shared memory segment facility of UNIX System V) across concurrent processes*[11].

The multiprocessor system architectures can be splitted in two basic categories, depending on the type of the CPU/memory subsystem interconnection network:

- bus connection based

- hypercube connection based.

To fully profit of a cube-based multiprocessor architecture, an application must be able to take advantage from a fine grain processing parallelism and must be designed for it. Such types of applications are not common at all today in the UNIX marketplace. Consequently, this type of architecture will not be discussed.

Multiprocessor bus based systems can be grosly divided into three categories:

low-end , up to 4-16 processors (1-4 of them CPUs)
midrange, up to 16-64 processors (2-30 of them CPUs)
massively parallel systems , more than 64 processors.

[10] The major origin of S/W interferences is the serialization ofmutually exclusive kernel sequences. For example, while a processor is executing a kernel routine which scans a linked list, an other processor can not execute a routine which modifies the links between the items of the list. To avoid that, [Dijkstra's] semaphores (complex structures that can handle a queue of resource requestors) are used: a semaphore is associated with the list itself and a routine, before to access the list must get the semaphore. If not, due to the fact that a routine running on an other processor has already gotten [and not yet released] the semaphore, the execution of the routine is halted and the current process *goes to sleep* till the semaphore will be released (in some cases, when the locking of the semaphore is expected to be very short, the routine executes a *spin lock* loop). If the current process goes to sleep or if the spin lock loop is executed, there is a S/W overhead (i.e. the context switch processing sequence or the spin lock loop itself) which results in a loos of processing power.To minimize this problem, a *fine grain semaphorization* is programmed,meaning that the number of semaphores is increased (for example, referring to the above linked list example, the list items are groupped according to an hash value and a semaphore is associated with each hash chain), so thata semaphore is locked for a very short time to minimize the probability of interference. But also with this approach there an impact on the real throughput, due to the fact that the execution of the synchronization primitives requires extra processingpower. This extraprocessing, not needed in a monoCPU environment, has to be considered as a decrease of the global system available CPU throughput.

[11] The shared memory segment facility originally was not supported by UNIX. In fact time sharing applications (the initial UNIX target environment) are sharing between concurrent system users (i.e. processes) the central H/W resources (main memory, CPU, etc) and the peripherals (disk, printers, etc). But the sharing of objects like files is not a major requirement for time-sharing applications.
But as soon as UNIX has started its evolution toward other application environments (like departemental systems), its basic facilities were extended to provide the interfaces for supporting a concurrent access to a common data base and in general to common data structures. These exetnsions were named IPC (inter process communication facilities) and they include the shared memory segment facility. Incidentally these are the basic interfaces needed to implement a symmetryc multiprocessor database management subsystems.

The discussion will not address massively parallel systems. To-day, these systems are addressing niche market segments, like hypercube based multiprocessor systems.

3. THE MAIN BUILDING BLOCKS

These paragraphs provide a general description of the S/W environment to be supported and defines the basic system building blocks.
Focusing the discussion on the CPU/main memory subsystems, the most relevant S/W requirements are the main memory management requirements: they will be discussed in the following paragraph.
Whereas for the S/W the issue is to make the best possible porting/enhancements of the UNIX kernel to efficently support a multiprocessor H/W, for the H/W the issues to be addressed are quite wider and have as basic the definition of the building blocks. This definition will, first of all, try to use standard building block, due to their obvious advantages in terms of required R&D effort and intrinsic learning curve. Where a standard off-the-shelf component is not yet available, like for a CPU/memory bus, a specific paragraph will discuss what can be reasonably achieved using lower level off-the-shelf technology.
Considering than the characteristics of the H/W building blocks and of the S/W requirements, we will try to identify the architecture implementation that provides the best cost/performance ratio for a given scalability of system performances (in this discussion the CPU number).
It is important to underline that we are not shuting for something which has only a theorical validity, i.e. "*The Architecture*" , but for *the architecture which provides the most convinient compromise for a product defined in the 88 and with a lifecycle in the 89-90-91 timeframe (i.e. at a precise spot of the technology evolution) between cost/performance (including R&D costs)*[12]. As any timed H/W architecture implementation is somehow technology driven, most likely at the next product evolution a different implementation could be preferrable; *what will not change are the criteria and the methodology for the selection of a [multiprocessor] architecture.*

The key point to get a succesfull product (from the engineering point of view), is to precisely and properly define the design point on which optimize its cost performance through tuned architecture definition and clever engineering implementation. The usage of the above methodology in all system areas (even if it seems quite obvious - the so called good sense approach) is not at all a common pratice, as it requires a great deal of knowledge, collaboration and integration between different engineering skills.

3.1 The S/W framework

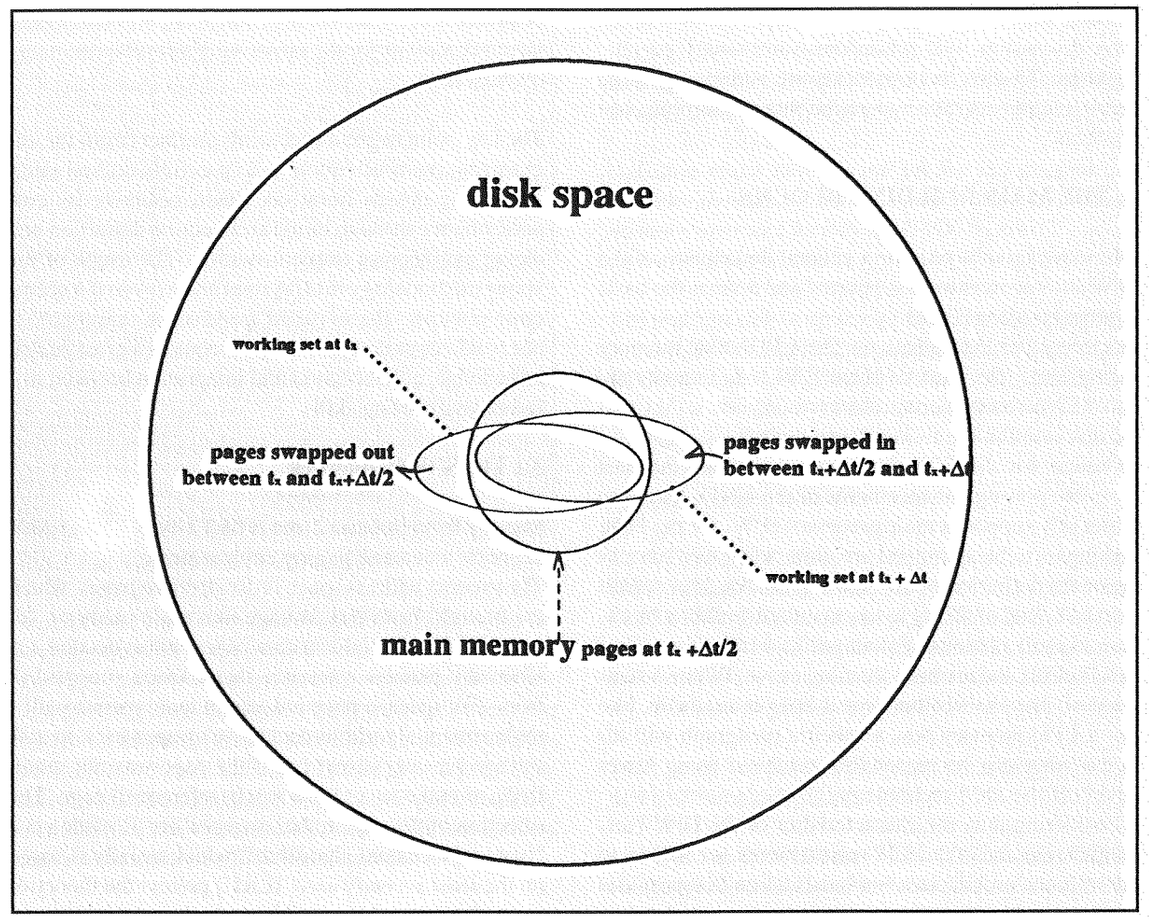
Starting from Release 2, the AT&T UNIX SYSTEM V supports a demand paging environment.
The process address space is divided into pages, which are brought from disk storage into main memory (or created, if not initialized data pages) when needed, i.e. when the process reference them. Once *swapped in* from disk space, a page is keept in main memory till a replacement algorithm decides to swap it back to disk storage (or only discard it, if the page was not modified), to make room for a newly referenced page. The selection of the page to be *swapped out* is made via a "page replacement algorithm", which usually is based on the *least recently used* (LRU) policy: the theory of the "process working set" is applied.

The objects stored into the disk space and main memory that are relevant for the following discussion are:

disk space:	**memory space:**
program sections	program pages:
- text	- user & kernel text
- initialized data	- user stack and data
- etc	- kernel data:
	- - per process stack and page tables
	- - global tables
data files	device drivers
swapp space	I/O buffers
	shared memory segments

The two way swapping between main memory and disk space is:

[12] In this evaluation it will be taken into account also the forecasted volume distribution of the possible achievable configurations, i.e. the cost/performance ratio must be optimized for the models with the largest expected volumes).



The main memory pages are classified as follow:

page type	page contents	characteristic
X (exec only)	program text	sharable among processes
r/w (read/write)	program stack and data	private to a process
R/W (read/write)	shared memory segment global kernel tables	data share among processes

The ratio between the references to private (r/w) pages and the references to shared data (R/W) pages is application dependent. The following two figures will be assumed within this discussion[13]:

- 15% of data references are to shared data pages in Transaction Processing applications
- 5% of data references are to shared data pages in Office Automation and Scientific applications

[13] the ratio of *global* versus *local* data references was analyzed also for finding a solution to other type of architectural problems, so that a lot of figures are available on the subject (refer in particular to [2]). In any case we have measured in our labs a MC68020 based machine, running UNIX System V release 2 and the most popular horizontal UNIX applications. Via an H/W facility, able to count the number of references to the various page types, we have gotten the figures of the ratio of references to the shared data pages: these figures basically confirm the values published in the literature.

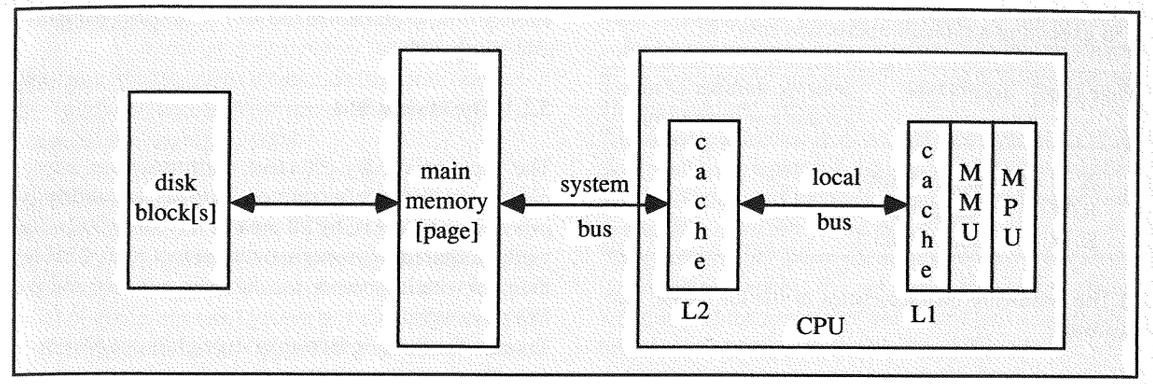


Fig. 2 Memory hierarchy

It must be underlined that the above considerations and figures are not dependent from the particular main memory management implementation. The features relevant for the following discussion are that the process virtual address space is mapped onto a paged memory, the process pages are loaded into mein memory on demand and swapped out according to a replacement algorithm, and that [some of] the target applications are based on the shared memory segment facility (i.e. the capability to access from different processes common data, providing that the access time to common data is comparable to that one of the local data[14]). A part from that, the considerations done in the following paragraphs are applicable to many different [UNIX] memory management implementations, including AT&T UNIX System V, IBM AIX version 3, Carnegie Mellon MACH, etc.

3.2 The architecture building blocks

This paragraph defines the H/W building blocks used for the system implementation. Obviously the building blocks selected are those available within today technology and its foreseen evolution in the product lifetime.

[14] This requirement implies a *tightly coupled multiprocessing* capability

[15] The hit ratios of the MMU *Address Translation Cache* will be assumed:
99.68% for the Intsruction ATC
98.5% for the Data ATC.
For an analysis of the MMU *translation buffer* performances, refer to [14].

As already mentioned, furhter evolution, that improves and/or modifies the characteristics of the assumed building blocks, can also change the results of this discussion. Nevertheless what will not change is the mothodology used to define the preferred architecture.

The memory hierarchy, implemented on most of the today systems, is the following:

1) An MPU with the following characteristics is considered:
- a so called *single chip CISC* engine (being Intel X86 and Motorola 680X0 the major players)
- integrated MMU[15] and first level cache (~8 KBytes - splitted or unified - see following paragraph for the assumed hit ratio) and on-board basic FPU engine
- splitted 32 bit address/data bus
- 25/33 MHz (in the 1989/90) and 40/50 MHz (in the 1991/1992) operating frequency and 1.8 average cycles per in struction (cpi) with a 100% integrated cache hit ratio
- The number of references to the different page types

[16] These figures, as the previous one, was gotten running in our labs UNIX Syetem V Release 2 and the most popular horizontal applications on a MC68020 based machine and making some extrapolations on the technology (H/W, for the MPU, S/W for the compilers) evolution. Refer to [6] for a detailed description of the characteristics of the MC68XXX interiore decore. For other standard chips, like the INTEL X86, the figures changes mainly for what concerns the AIT but not very significantly for data reference ratios. Finally, for RISC architecture based MPUs, the average number of data references per instruction is lower of that one assumed for CISC MPUs (depending on the implementation, mainly on the *register window* facility), being still 1 for the instruction fetch.

in executing a typicall instruction mix[16] is:

Page type	access type	average number of references per instruction
X	instruction fetch	1
R/W	data read	.4
R/W	data write	.2

2) The following characteristics of the L1 cache[17] are assumed:

- access from the MPU to the cache is at *zero wait states*[18]
- the cache line size is of 16 bytes[19]
- the cache provides a *snooping* facility, capable to guarantee consistency between the cached data and data transferred into main memory through the bus to which the cache is connected (the local bus of fig. 2)
- the cache policy is selectable, on a per page basis, between *write through* and *write once*
- the *dirty reads* are assumed to be 50%, but an H/W assist avoid any delay in fetching data from main memory into the cache (*deferred write* after miss recovery)

The paragraph § 3.2.3 will discuss the L2 cache, i.e. the second level cache of figure 2.

3) A unique system bus to properly interconnect the CPUs with their server (main memory and I/O processors) and with enough bandwidth to sustain the generated traffic. The following paragraph will define its characteristics.

4) A main memory with high modularity, full EDAC capability, designed with 1 Mbit chips but open to 4 Mbit evolution, capable to sustain block bursting at the bus frequency and with enough level of interleaving to reduce the access time streching factor and the overall latency time to guarantee the best achievable computation bandwidth.

3.2.1 The system bus

The skeleton of any effective multiprocessor implementation that can guarantee a linear scalability of processing power is, by all means, the communication media required to properly interconnect each CPU in the pool with its servers, i.e. the main memory and the I/O processors.
Two are the limiting factors in any tightly coupled multiprocessor environment:

- the required *communication bandwidth*[20],
- the achievable *computation bandwidth*[21].

We will analyze both of them, with the objective to properly trade off the possible achievable performance with the related cost to get, to approximate at best an ideal linear growing of processing power.
In other words, we will armonically exploit at its best the engineering, technology and design knowledge to get the most effective(cost/performance) multiprocessor system.
Given a processor type (RISC/CISC), an application environment (e.g. transaction processing or office automation) and a defined objective of scalability granularity, we can calculate the required communication bandwidth.
As the maximum achievable thorughput of the given pool is the ratio between the technologically possible bus bandwidth and the number of bus cycles required to satisfy the CPU *external requests* (i.e. the communication bandwidth), we are shurely facing the classic "cicken and egg" dilemma.
Too many parameters can strongly influence this ratio and consequently the so called $ per Mips but, as the result here presented will validate, the key to reach the objective of a real "*optimal multiprocesso rimplementation*" are:

- the proper tune of the system architecture partitioning and the available of-the-shelf technology
- a smart packaging solution.

3.2.2 The bus bandwidth

The first limiting factor to be properly weigthed is the possible bus bandwidth, that is the absolute safe and produceable minimum system bus clock cycle, achievable with a given technology in the product timeframe.
The *bus cycle time* is mainly the sum of four effects:

- the intrinsic gate delay of both the needed driver and receiver
- the number of bus cycles for a given hand-shake protocol
- the driving capability derating factor (physically and technology limited)
- the intrinsic media (etched line) delay.

Being these effects all technology dependent, their cost sensitivity is quite high, so their tuning can strongly influence the product cost/performance objective[22].
One interesting thing that is worth to point out is the almost linear dependency of cycle time on the number of connected loads (i.e. processors), which is creating a degenerative positive feed-back between load capacity, driving current and bus delay.
Considering the above and the foreseen product availability, we decided to safely use for our analysis off-the-shelf TTL FAST MSI for bus interface and two level of handshaking per bus cycle (bus grant-data).
This lead to a possible cycle time for an eight slot cage (4 CPU + 4 Memory controllers) of 60/50 ns (16/20 MHz).
Now, knowing the bus cycle within a given technology and processor pool physical constrains, the bus protocol and the bus width have to be defined: both of them will determine its throughput.
Considering the bus protocol, the most promising possibility in term of bus efficiency (latency time) between *circuit switched* and *packet switched* is surely the last one. Being the packet switched bus not locked during the memory access, the bus occupancy of this protocol is for sure the minimum achievable, with the drawback to be more costly and complex to implement than the circuit switched approach.
The other parameter that strongly influences the bus occupancy, throughput and cost is its width (i.e. number of bytes) and/or its splitting in separate paths for read and write (*multistructured* bus).
The cost is mainly due to the number of driver/receiver, the board real estate, the pin number in an integrated solution and the number of connector pins required (at board level) to support the high parallelism.
An other dangerous positive feedback can be created if we trade off this parameter on the following:

decrease the cache miss ratio via an higher line size (mainly for instruction miss)

⇩

higher bus occupancy due to the cache line burst filling

⇩

bus occupancy due to a defined bus width, given burst filling and line size.

Above this technical issue, another important factor to be considered is the objective to keep the processor board interfaces within a given standard (with obvious production advantages): the exploiting of this advantage results in enforced constrains on the board size and the connector type.
To get a reasonable bus occupation (assuming a cache line size of 16 bytes to keep the same line size of the cache integrated into the CPU) and shooting to achieve as much as possible a low cost solution, the decision was to analyze different busses, with a data width of 8 bytes, a clock frequency of 16/20 MHz, multiplexed address/data or separate address read/write and read data[23].

In first approximation we also did not consider the bus overhead activity due to processor synchronization

[17] With the acronym L1 (i.e. first level) cache we are referring to the cache integrated in the MPU (see figure 2).

[18] zero wait states means that the MPU does not loose any cycle in fetching/storing data from/into the cache

[19] This assumption is arbitrary: but selecting a different line size the following considerations will not change from a qualitative point of view.

[20] The communication bandwidth is the number of bus cycles (for a given bus width) needed to sustain without a significant degradation the CPUs access frequency to their servers (in this case memory and I/O).

[21] The computation bandwidth id the maximum number of instructions executable by the system in the unit of time. This figure depends from the single CPU Mips, the number of CPUs and the average latency for each CPU external request.

22 In this context "cost" does include both the intrinsic technology cost and the indirect costs deriving from the time-to-define/get/support of a proprietary system backbone like the "system bus".

[23] We forsee and welcome an 8 byte synchronous packed switched multiplexed bus, at 25/33 MHz in a standard 96 pin connector: this seems a product for release in the 90 time frame, so out of the scope of the planned implementation.

(*test&set*) and the related bus locking constraint[24].

3.2.3 The cache

Both the computation bandwidth and the required communication bandwidth are strongly influenced by the miss ratio of the distributed caches.
We define the bus traffic reduction factor as the ratio of the native CPU bus traffic and its real frequency after cache filtering. This parameter is very cost sensitive due to its dependancy on cache parameters and policy, i.e. the schema used to guarantee consistency between data stored in the distributed caches and in main memory [25].
Different cache policies strongly influence the system bus traffic, with relative consequences on the throughput.
Within three viable write policies (*write through* , *copy back* and *write once*), the one which achieve the best performance is some form of write once [3] [4]: we will use the overhead induced by this protocol to calculate the bus occupation in the defined environments.
All the possible cache consistency protocols try to reduce the bus traffic and to improve both the local hit rate and coherency overhead through the definition of multiple local data statuses and an intelligent separation between shared data and private data.
Even if some sophisticated protocols like DRAGON or FIREFLY can guarantee in most environments the best performance and an optimun bus traffic reduction factor, we decided, for shake of simplicity, to consider in our analisys a simple form of write-once, which can anyway guarantee good performances at lower cost.
A plenty of protocols has been studied and many of them (e.g. Synapse, Berkeley, Illinois, Firefly, Dragon, etc) has been somehow implemented in today available bus based multiprocessor systems. Some of then are (or will) even be supported by proposed standards like FUTUREBUS or will be integrated in future implementation of some RISC architecture to effectively support multiprocessor with a standard integrated solution.
The general trend in both RISC and CISC implementation is to fully exploit the available process technology and integration capabilities, fitting on the same peace of silicon as much as possible architectural functionalities[26]. Given that and as the performance disparity between main memory DRAM and the new processors widens, it becomes necessary to implement both an on chip cache (L1 of figure 2) and a second level cache (L2 of figure 2) implemented with SRAM devices and possibly tightly couple their architecture to specific processor interface. In fact the speed of standard SRAM technology seem to become incompatible with processor cycle time and required throughput. Due to that, the only solution that has chances to take advantage of the foreseen processor power is to integrate a first level cache on processor silicon and to use a second level cache, to further filter the external required access per instruction. In this way we can achieve an optimum and balanced (cost/perf) environment, in which a relative big L1 cache operates at the processor speed (50-60 MHz) within the same integrated environment. The accesses to the L2 cache are made at half the operating frequency (25-30 MHz), to be compabile with the second level of technology and integration (SRAM 256/1 M - 15/25 ns).
This solution can guarantee the best compromise on cost/speed/size of the memory hierarchy.
Any memory location that have image on L1 cache will also have copies in the associated second level cache L2. As L2 will see the miss ratio of L1, its effectiveness strongly depends on its parameters. Its size (and consequently hit ratio) needs to be tuned, to guarantee effective bus traffic reduction (consistently superior to the given by L1) and increased computation bandwidth at reasonable costs.
In conclusion it becomes more and more important:

- the ratio between the cache access time and the main memory access time
- the size of L2, to guarantee a worthwhile reduction in miss rate and consequently on the bus traffic[27].

In some cases the L2 cache can decrease the system performances (increasing memory cycle cost on miss) when the size of L1 is so high, respect to L2, to strongly reduce the hit ratio. On the other hand the look-ahead memory cycle will reduce this problem, at the cost of increased bus traffic (we can somehow get a better teoretical computation bandwidth reducing the available communication bandwidth).
For what concerns the write strategy on the two level caches, different overall performance and complexity can be achieved. The best performance are given by the write-back/write-back policy (the first refers to L1 cache, the second to L2) and than in the order write-back/write-through, write-through/write-back, write-through/write-through.
The best global implementation in terms of performances, complexity and costs is the write-through/write-back. In this case, considering L1 as an inclusive subsetof L2, only the L2 directory need to snoop the bus and propagate proper invalidation down to L1.
Some other techniques like look-ahead fetching on miss (i.e. read given address and next) can guarantee an overall improvement of system performance, improving the hit ratio (specially on instruction miss), reducing global bus occupancy (burst reading) and taking advantage of intrinsic DRAM features (page mode or static column). Obviously all these features will add complexity and cost to the implementation and some of them will have questionable results, so that in our discussion we will consider an effective and simple L2 cache implementation (see § 4.3), assuming that L1 cache is 4K+4K, four set associative.

Due to sensitivity of the architecture to the overall (L1+L2) cache hit ratio and due to the lack of measurements on which we can be confident, two figures will be assumed:

page type	hit ratio (figure 1)	hit ratio (figure 2)
program text	98%	99%
program private data	94%	97%
shared data	70%	80%

Fig.3 The assumed cache hit ratios

It must be remarked that the above figures provide the "global" hit ratio, i.e. the hit ratio resulting from the hit ratios of both L1 and L2 caches.

4.THE ARCHITECTURE MODELS AND THEIR MEMORY AND BUS BANDWIDTH REQUIREMENTS

In the previous paragraphs we have discussed and selected the system building blocks. This selection was done independently from the target architecture, being the selection criteria the achievement of the best cost/performance ratio from the off-the-shelf technology. Now we have to define on the basis of the above building blocks the most convinient (i.e. effective from a cost/performance point of view) architecture. This definition is crucial, *because it is the differentiating factor with respect to oder standard systems*[28].
This definition is also an exciting game: it remembers

[24] Some of today multiprocessor implementations (e.g.Sequent and S.G.I.) use a separate, integrated bus for the above function, providing a given amount of test&set variables available for allocation to the applications (i.e. mapped in their virtual space) and to the kernel. The intended use of these variables is for fine grained locks and proper interrupt distribution to the processors. In this case the processor bus can be fully dedicated to efficiently support coherent data sharing between processors.
Again the optimum possible achievable performance using separate busses for these different functions comes at some extra cost. Considering the kindof system we were shooting for, the overhead of cost to optimize the synchronization was not judged to be worth the gained advantage and the decision was to support test&set on the native system bus.

[25] Process communication and its special form known as synchronization (data are control information), can guarantee consistent parameter passing between processes and allow multiprocessor system to execute concurrent, parallel operation enforcing protection against asynchronous access to common shared resources.
In a shared memory multiprocessor implementation, this is achieved through controlled access to shared data It becomes then mandatory to guarantee memory consistency, so that at any memory read the returned data is the most updated version available within the possible memory hierarchy in the system.
We can than envision a "consistency continuum" in which implemenation complexity, performance and cost are strongly influenced by the chosen approach.
We can generally state that the best approach (versus cost/performances) for bus based multiprocessor systems is the "when needed, on the fly" coherency schema supported through fully distributed, per processor caches (with the needeed H/W on the bus interface to guarantee consistency).

[26] Today technology (1/.8 CMOS/BICMOS) and design stile can guarantee quite high clock operating frequency (33-60 MHz) with cpi fast approaching the 1 and below limit in RISC implementation (multiscalar approach and strong compiler technology) and the 1.5-2 in (soon to come standard) compatible CISC (RISC design methodology on top of CISC compatibility value).

[27] With on chip integrated caches in the range of 8 Kbytes multi-set-associative (to become by end 90 16-32 Kbytes) a L2 cache at least 8-16 time larger is needed. 1 Mbyte or even 2 Mbyte L2 cache sizes will be common in 91/92 timeframe.

[28] For what concerns the comparison with proprietary systems, it has been proven that the standard systems provide quite better cost/performance figures than proprietary systems:refer to the numerous reports of Dataquest and Infocorp on the subject.

somehow theselection of the way to play a card hand at a bridge tournament, when the same cards (the building blocks) were distributed at all the tables and you know that the other players are trying to get the best from the same cards.

These paragraphs will compare three different architecture alternatives:

GMA *global memory architecture*
LMA *[shared] local memory architecture*
MMA *multiplexed [local] memory architecture.*

Once described them (very shortly the GMA and the LMA, because are very traditional solutions, in some detail the MMA, which is an innovating solution) and evaluated their main memory and system bus bandwidth requirements, a comparison of the achievable performances and related cost/performance will be done.
The following table provides a schematization of some multiprocessor CPU/memory architectures:

COMMUNICATION & SYNCHRONIZATION

MEMORY ARCHITECTURE	SHARED VARIABLES (tightly coupled)	MESSAGE PASSING (loosely coupled)
GLOBAL MEMORY	**GMA/sv** (ENCORE/SEQUENT)	**GMA/mp** (ELXSI 64K)
DISTRIBUTED MEMORY	**LMA** (BUTTERFLY) **MMA** (SEE ARTICLE)	**DMA/mp** (HYPERCUBE)

Fig. 4. Synoptic tableof multiprocessor architectures

4.1 The global memory architecture

The characteristic of the global memory architecture is that all the CPUs are connected via the system bus to a unique shared main memory: it is the traditional, simplest way to provide tightly coupled multiprocessing, so the most used and speculated one.

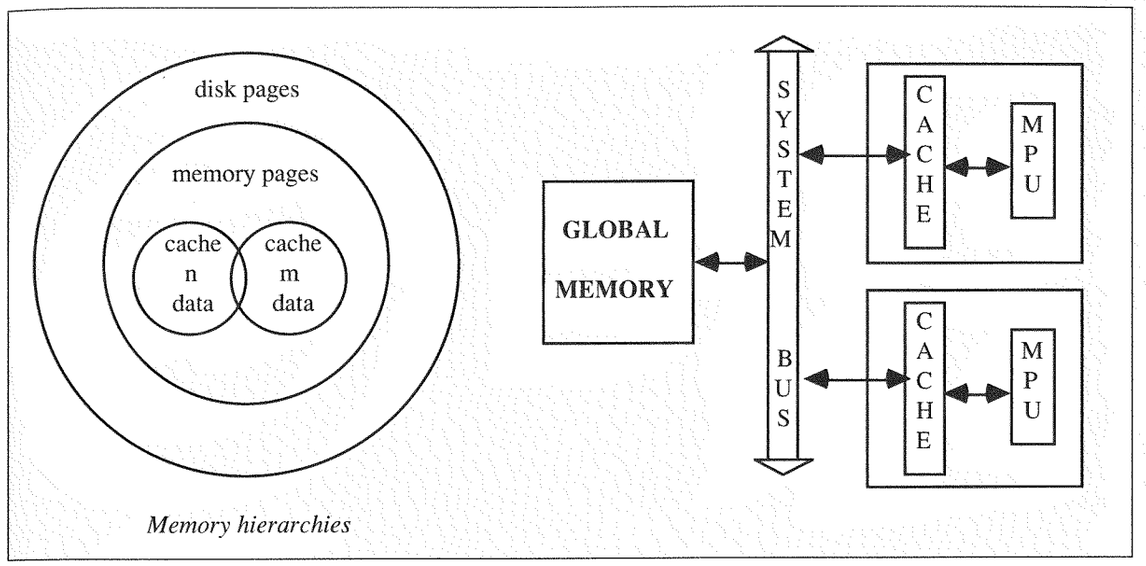


Fig. 5. The global memory architecture block diagram

Each CPU concurrently recovers cache misses fetching data through the system bus from the main memory. The write once policy guarantees the consistency of the main memory data distributed on the various caches (the same data stored into more than one cache can be writable only in one of them). Without going into a detailed description of this very well known architecture model, it must be underlined that its most critical building block is the cache and its write policy (as shown in the figure 5 memory hierarchies , the caches of different CPUs can store common data: the cache policy must guarantee consistency, as discussed in 3.2.3). In fact this is the item of the architecture that provides the mean to limit the system bus and memory bottlenecks (which throughout, as discussed in the previous paragraphs, has technology limits - even considering the processor and main memory interleaving efficency).
In any case, with the assumptions made in the previous paragraph, the number of system bus cycles originated in average by an instruction, is in the following range:

Scientific, OA Application	*TP Application*
.061 ⟶	**.077**

With the above figure, the required bus throughput and the required main memory cycle time (cumulative cycle time, in case of interleaved memories) are a linear function of the target aggregate H/W CPU throughput, expressed in millions of [aggregate] instructions per second (Mips).
Making the assumption that the S/W can support any number of CPUs, being the mutually exclusive sequences defined with the proper granularity, this architecture provides the maximum level of symmetry , that is every CPU can run at any time every type of processing sequence.
This symmetry provides the S/W with the capability to maximize the utilization factor of the CPU, i.e. to get the maximum aggregate real system throughput.
A critical point for this architecture is the way to dispatch interrupts. The criteria is very simple (dispatch the interrupt to the CPU running the process[ing sequence] with the lowest priority), the implementation is quite complex and/or expensive.

4.2 The local memory architecture

As mentioned in the previous paragraph, the limiting factors of the global memory architecture are the system bus and main memory bandwidths. The objective of the local memory architecture is to effectively mini

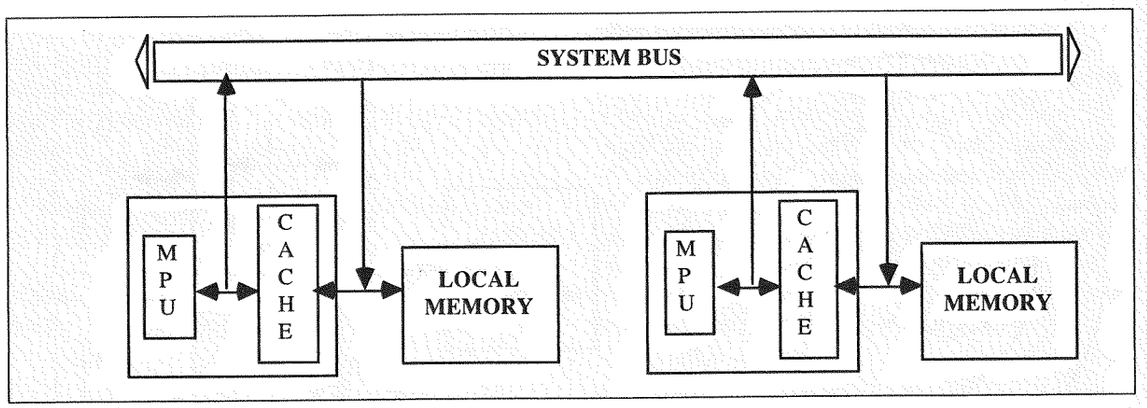


Fig. 6. The local memory architecture block diagram

mize both of them, to get a more competitive cost/performance ratio[29].

In this model each CPU can access:
- its own local memory through a local CPU/ memory bus
- the local memories of the other CPUs via the system bus.

To guarantee consistency of cached data, each CPU can cache only its own local memory data. Due to the above and to the fact that fetching shared data across the system bus is quite more expensive than fetching data from the local memory, this architecture is valid only if the CPU memory references to shared data are quite low, being most of the memory references to the local memory (refer to § 3.1 for the [page] data classification).

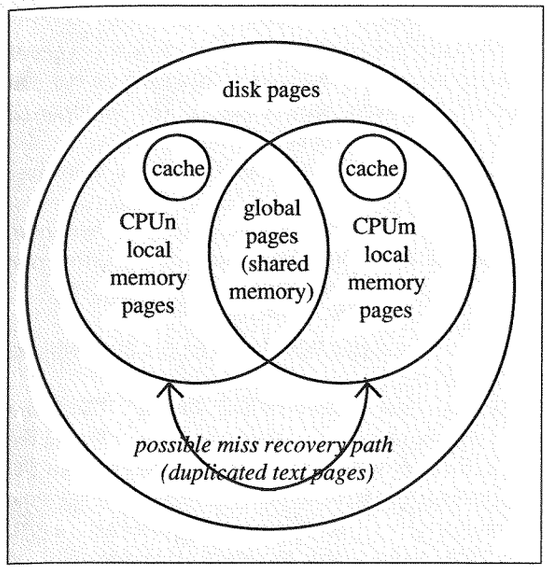
To achieve that, the entire execution of a process is assigned to a specific CPU[30], so that all the pages accessed by a process are loaded into the local memory of the same CPU which is running it. This approach is applicable to all the page types, but to the shared memory segment pages. In fact these pages are shared between different processes: if two processes, which have attached the same memory segment, are running on two different CPUs, one of the two accesses the shared memory segment on the pages of the other CPU local memory, crossing the system bus.

The hypotesis is made that shared memory segment pages are scatterly allocated on the various local memories.
Summarizing, the following rules are applied to the various page types:

page contents	*characteristic*	*allocation rules*
program text	sharable among processes	possibly ennuplicated on LMs
program stack and data	private to a program	on one LM only (no cross references - local)
shared memory segments global kernel tables	shared among processes	on one LM only (cross references through system bus)

[29] During the past two years many UNIX system builders have announced multiprocessor systems based on a local memory architecture. Among them:
- AT&T 3B4000
- NCR TOWER 32/8x0
- Counterpoint S19
- Texas Instruments System 1500
- etc.

[30] When the process if created (or better at exec time, in order to minimize the number of process pages to be eventually migrated from one CPU local memorty to an other one) the kernel evaluates which is the CPU with the lowest utilization factor (in terms of the two basic CPU local resources - MPU busy time and local memory page swapping activity) and assigns to it the execution of the process.



Note that text pages are duplicated, i.e., if the same process is executed on two CPUs, the text page is loaded on both local memories. In this case, the page miss recovery occurs between two local memories with a miss recovery time which is two order of magnitude lower than the recovery of a page miss from the disk.
The duplication of the text pages causes some waste of main memory, but strongly reduces the need to fetch highly referenced data from one another CPU local memory through the system bus, thus avoiding a degradation of the CPU performances and an increase of the system bus bandwidth requirements and/or occupation.
Global pages (i.e. pages of shared memory segments created with read/write access mode) are not duplicated, because they contain R/W data that must be shared and kept consistent for all the CPUs.
The access to a shared data page (from the CPU that has not allocated it on its own local memory to the local memory of an other CPU via the system bus) is the most critical factor for the performance of this architectural model. To avoid that, some systems (like the AT&T 3B4000) set the limitation that the processes that attach the same shared memory segments have to be executed within the same CPU. This solution will guarantee performances but it is not always applicable (i.e. not all the application can accept this constrains) and more is a deviation from the de-facto standard interfaces: we considered it not acceptable.

With the assumptions made in the previous paragraphs, the number of system bus cycles, originated in average by one instruction, is dependent not only by the application type, as for the global memory architecture, but is also a variable of the CPU number, as expressed by the following formula, where n is the number of CPUs:

Number of system bus cycles originated in average by an instruction

Scientific, OA Application	*TP Application*
.06 x (n-1)/n	.18 x (n-1)/n

Number of memory references in average per instruction

Scientific, OA Application	*TP Application*
.138/n	.1925/n

Comparing this architecture with the global memory architecture, the number of system bus and memory accesses per instruction is linearly proportional to the number of CPUs and is significantly lower than in the GMA (see figure 11).

From the S/W point of view, this model differs from the unique global memory model for the fact that less resources are shared between the CPUs: the main memory is mainly a local resource and the process management has to deal with a local resource (the MPU allocation is arbitrated between the set of processes whose execution was assigned to the CPU itself).
This has two types of implications:

- positive, because the kernel routines which execute the process management and memory management need no/less synchronization, sothat the S/W inter ferences are reduced

- negative, because, according to the queue theory, if α is the number of processors (servers) and β is the number of processes to be executed, the average number of processes (γ) which are in the ready to

run queue is greater is LMA than in GMA:

$$\gamma = f\left(\frac{\alpha}{\beta}\right) \quad =< \quad \gamma = f\left(\frac{1}{\frac{\beta}{\alpha}}\right)$$

It must be remarked that in the typical UNIX processing environment several processes (per active users) are created. That means that in general has an high value with respect to , reducing the impact of the above drawback on the local memory architecture performances.
In any case, to minimize the problem, the S/W provides a *dynamic load balancing* facility: the execution of a process can be switched from one CPU to an other one, in the case that the load of a CPUs become significantly unbalanced, due to the termination of many processes executed by the same CPU[5][31] .
In any case the negative effect of the process to processor allocation (i.e. probability of lower CPU utilization and the dynamic load balancing overhead) is greater than the positive one (i.e. less kernel routines need synchronization). The measurements activities done on a local memory based machine indicates a 10% of throughput degradation in the worst cases. To be conservative, we will use this figure: usually the degradation is quite less, meanly when the system is significantly loaded, i.e. when the aggregate CPU throughput is really needed.

4.3 The multiplexed memory architecture

Even if the local memory architecture seems interesting because it requires less memory and bus bandwith [with respect to the global memory model], it has a major drawback. In fact the shared data, not only have to be accessed through the system bus, but can not be cached (to guarantee consistency). Due to that the AIT[32] is increased, with the consequence that the number of processors needed to reach a target aggregate throughput his higher than in the GMA architecture.
To solve this problem, still keeping the advantages of the local memory architecture (i.e. minimum bus and memory bandwith requirements), the *multiplexed memory architecture* has ben defined.
The characteristic of this architecture is the following:

a CPU references only data stored into its own local memory, till providing tightly coupled multiprocessor environment.

This is the best possible set-up to get the optimum performances!
To achieve that, whenever two or more CPUs are accessing the same shared memory segment [page], the shared memory segment page is duplicated on the CPU local memories and is keept consistent (i.e. at the same updating level) via an H/W "write broadcast" assist. This H/W assist is able to detect write operations to shared memory segment pages which are allocated on more that one CPU. These write operations are broadcasted (via the system bus or equivalent communication network) to the other CPUs, which in turn update the referenced shared segment pages located within their local memory.
For read references, the shared memory segment data are always read from the local memory, with the following advantages (with respect to the LMA):

- the shared memory segment data are cached (consistency is guarenteed through the snooping of the write operations broadcasted on the system bus)
- the shared memory segment read operations do not use system bus bandwith
- the cache miss recovery time is lowered
- the computation bandwidth is higher as latency time is lower.

Obviously all the references to the local data and text (reads and writes) are kept within the optimized CPU local memory environment (as for LMA).
For what concerns the shared memory segment write

[31] The "load" of a CPU is evaluated weighting different factors:

- the CPU busy time in kernel and in user mode
- the process run queue
- the swapping activity
- etc

An algorithm allows to minimize the switching of the execution of a process from one CPU to another one (executed in order to achieve a better load balancing). because this process switching implies the overhead of the process page copying (executed on demand) across two local memories

[32] AIT is the average istruction execution time.

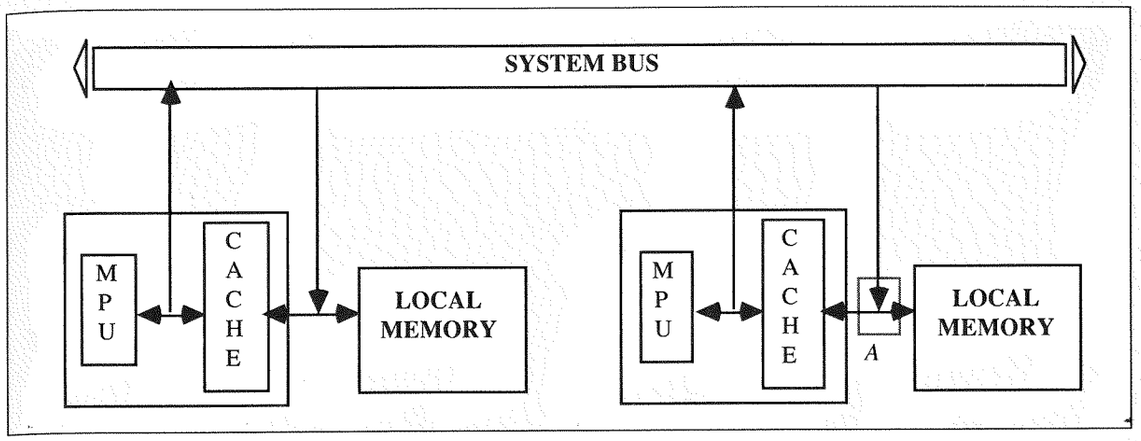


Fig.7. The multiplexed memory architecture block diagram

operations, these are still affecting the bandwith of the system bus (i.e. the *communication bandwidth*), but the impatc on the throughput of the CPUs (i.e. the *computation bandwidth*) is keept minimum by the *deferred (or posted) write technique* . A CPU does not wait for the completion of a shared memory segment write (which could require a significant time, because the write command has to be broadcasted on the system bus and all the local memories have to complete their updates), except when a test&set or a similar synchronous indivisible cycle is executed.
That is done to guarantee consistency of data across memories when actually needed[33].

The block diagram of the multiplexed memory architecture is thein Fig. 7.

In order to distinguish between global and local data, two address spaces are defined:

- a global address space, to which belongs all the shared memory pages which are allocated on more than one CPU local memory (see below)
- a local address space, to which belong all the other pages.

[33] A similar approach is followed by SEQUOIA multiprocessor systems, to guarantee consistency of information in a fault tolerant environment: between a lock and and unlock primitive, data are keept in a cache; the main memory is updated only when the synchronized sequence is successfully completed.

The size of each address space is such that each one can address the entire physical local memory, so that each local memory page can belong to either one of the two address spaces. A bit of the most significant part of the *real address* (in this discussion with real address we mean theaddress generated by the MPU MMU after the translation of the virtual address) is used to distinguish between the two address spaces. Whenever a write operation is done in the global space, the H/W recognizes it and broadcasts the write command [and data] via the system bus to the other CPUs, which update their local memory global page consistently.
Finally, the real addresses are converted into physical memory addresses via a second level translation (the *translation memory* - see following figure)[34].
The next picture outlines what stated above, giving a logical detail of the system behaviour in the box A of the figure 7.

[34] In reality a local space address corresponds to a physical address, so that local addresses (recognized for having the above real address bit off) do not need any further translation. Viceversa, global space addresses actually need a further translation, achieved via the translation memory. Note that the translation memory could be avoided, assuming the convention that a global real address is identical to a physical memory address but for the above bit, so that the conversion from real global address to physical memory address can be done very simlpy just turning off one bit. But in this case the same shared memory page should be allocated at the identical physical memory address on all the local memories, with a consequent higher complexity in the main memory management S/W. To avoid that, we have decided to introduce the traslation memory facility.

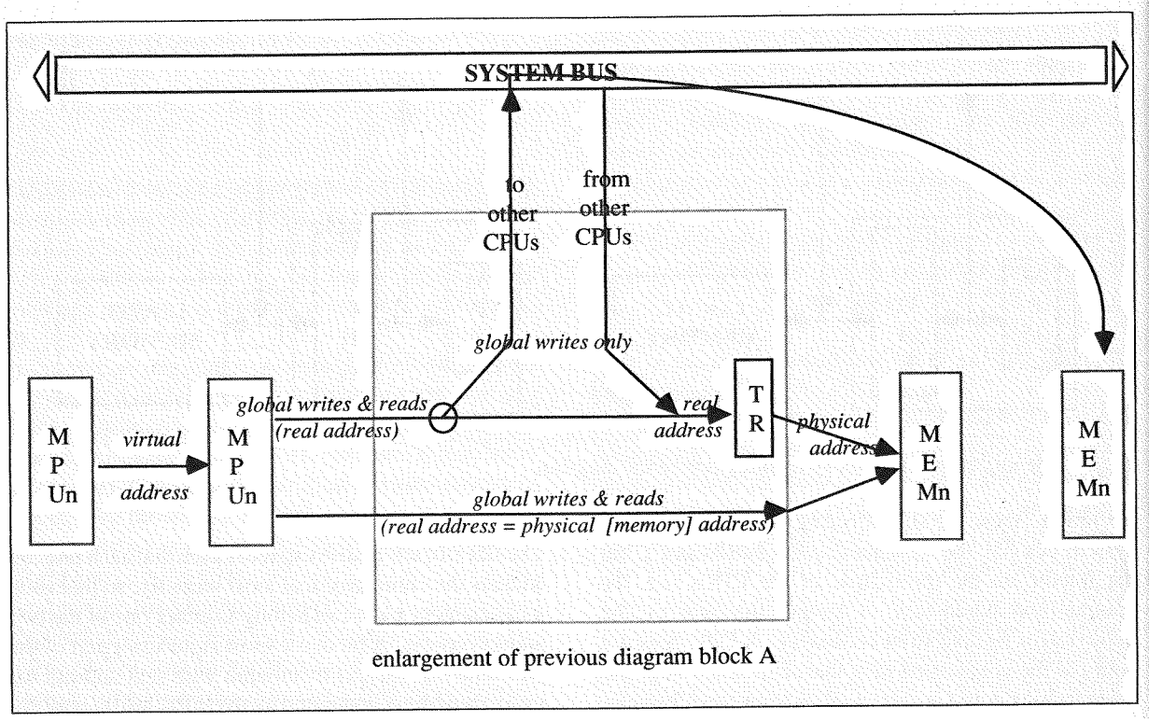


Fig. 8. Addresses and data in the multiplexed memory architecture

Figure 8 provides a logical diagram. Actually, in the proposed physical implementation, all addresses and data are exchanged through the same CPU/memory local bus.

It is worth to remind (see previous footnote) that the translation from real to physical address is required only for global data references.

A basic consideration: with the MMA architecture that waste of main memory due to the ennuplication of global pages [and the system bus traffic overhead due to global write broadcasting] can be keept minimum, because a page is included into the global space only on demand (i.e. when actually referenced by more than one CPU) and it is moved out from the global space [into the local space] whenever the page replacement algorithm allows to do that (i.e. when it is deallocated from the last but one local memory).

That means that when a shared memory segment is created by one process, the shared memory segment pages are allocated into the local space and are moved into the global space only when a second process running on an other CPU references them[35].

The life cycle of a shared memory segment page is the in Fig. 9.

Each page of a shared memory segment has its own individual life cycle. In particular, if a shared memory segment includes frequently used pages and seldom used pages, only the former have high probability to be

[35] For the CPU which has already allocated the shared memory segment page, the "moving" of a page from local to global space and viceversa, only requires to change a bit in the PTEs (the process address translation tables) and to compile an entry of the global addresses translation table. The overhead results from the fact that when a page of a shared memory becomes global, it has to be physically copied via the system bus from the local memory of the CPU which has the page already allocated, to the local memory of the CPU which is inserting it in its working set.

When a page becomes local (the event is detected by the CPU which is removing it from its working set), the CPU which has still the page must be notified by the CPU which is deallocating it, so that the notified CPU can change the page real address [space] from global to local (and the write operations for this page are not broadcasted any more).

Note that a global page does not need to exist on all the local memories, but only on those referencing it. In fact the global writes are broadcasted to all the CPUs, but are discarted by those CPUs which do not have the global pageinluded in their working set.

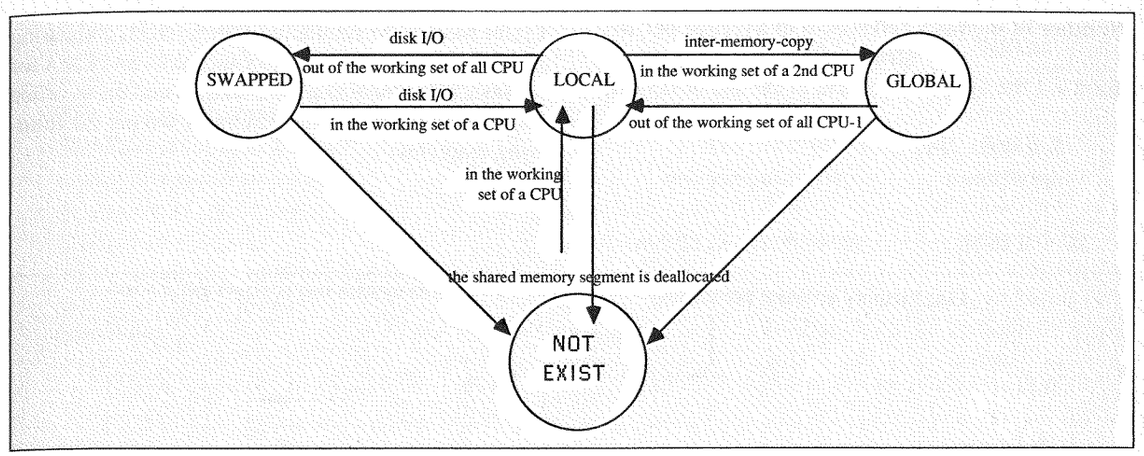


Fig. 9. Lifecycle of a shared memory page

ennuplicated (i.e. allocated in all CPU memories). This is the typical case of the shared memory segment of a Data Base Management System, which includes.

- frequently referenced *control structures* ,
- seldom referenced *buffers* .

Typically, even if the number of shared memory pages which contain *buffers* is two orders of magnitude greater than that of *control structure* pages, a local memory will have statistically allocated more control structure pages. The result is that buffers tend to spread across local memories without being ennuplicated, with the advantage that the recovery of a "buffer miss" from one local memory to an other one is about two orders of magnitude faster thanthe recovery from disc space !

The main consequency is that even if the multiplexed memory architecture requires more main memory with respect to the global memory model (due to the duplication of some pages), the inter-local-memories page copy facility achieves such better performance in term of miss recovery that these extra memory requirements are limited: in fact, to achieve a given level of performances, the size of a buffer pool (and for these aspects the main memory is a buffer pool of program pages) is a function of the buffer miss recovery time.

The average number of bus and main memory accesses per instruction is dependent not only by the application type as for the unique global memory architecture, but is also a variable of the number of CPUs, as expressed by the following formula, where n is the number of CPUs:

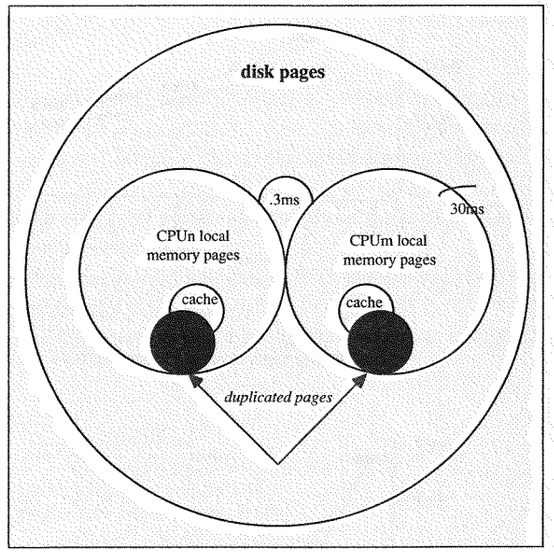


Fig 10. The MMA memory hierarchies

Number of system bus cycles originated in average by an instruction
(is 0 for a monoCPU)

Scientific, OA Application	*TP Application*
.02	.06

Number of memory references in average per instruction
(for n = 1 the figure is identical to the one of GMA):

Scientific, OA Application	*TP Application*
.01 + .1169/n	.03 + .1295/n

On the basis of the simulation of the system, comparing the above figures with those of the GMA and LMA for a transaction processing environment and for an office automation processing environment we get the following diagrams:

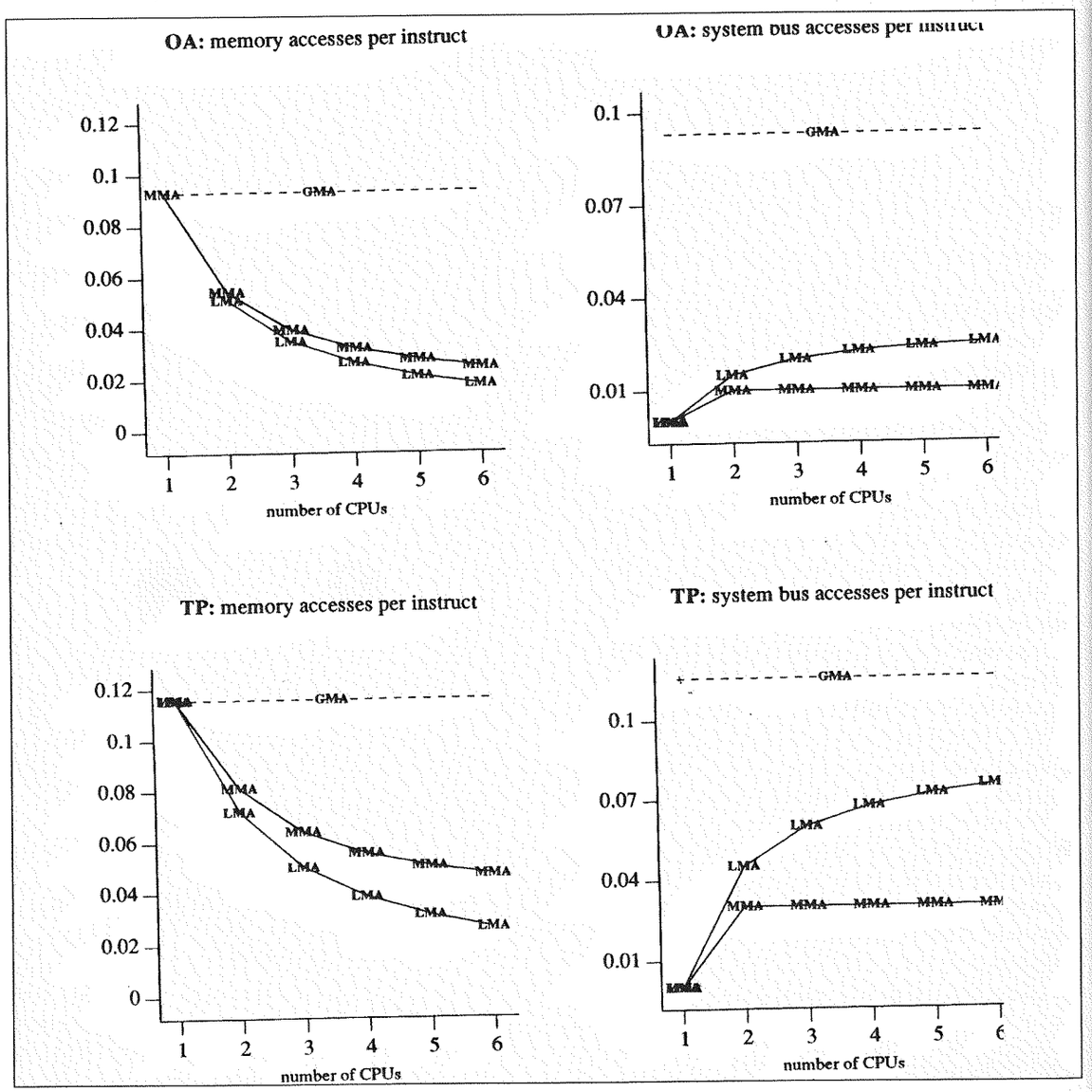


Fig. 11. Comparative figures for GMA, LMA and MMA

	GMA	LMA	MMA
BUS bandwidth required	high⁻	medium	low⁺
Memory bandwidth required	high⁻	medium	low⁺
Cache hit ratio requirements	high⁻	low/med	low⁺
Strong cache coherency	high⁻	low⁺	low⁺
Memory access time	symmetric⁺	asimmetric⁻	symmetric⁺
Performance degradation when shared data references increase	low⁺	high⁻	medium/low
Scalability	medium	medium/high	high⁺
Cost	high⁻	low⁺	low

Fig. 12. Comparison the three architectures

management, the same consideration done for the local memory architecture are applicable. Again, we will assume that with respect to the GMA a 10% of performance degradation due to [extra] S/W overheads (the load balancing efficiency and the inter-memory page copy) as to be payed.

An other consideration: from an implementation point of view, the MMA is more challenging than the other two architectures, due to the need to implement a more sophysticated main memory management (to hadle the lifecycle of a shared memory page). Viceversa the problem of interrupt dispatching disappears, because the terminations of I/O operations are notified to the CPU which is running the process waiting for that termination[36].

The of Fig. 12 table provides a synthetic comparison of the three architectures (some of these figures will be analyzed in detail in the following paragraphs).

See note[37] for some remarks about the above comparison criteria.
Note that taking advantage from the characteristics of the memory hierarchy, the L2 cache use for the MMA can be very simple, having the following characteri-

[36]A biunivoc association between a process and an interrupt is not always possible. For example, more processes can wait for the termination of the I/O of the same disk block or only after the execution of some [streams] modules it is known the process waiting for a packet received from the network.
Consequently, the case that an event notified to a CPU changes the status of a process running on an other CPU must be supported.

[37] **High bus bandwidth requirements imply:**
- high frequency clock
- high speed technology
- packaging real estate (board and connectors)
- electrical constrains
- long line size/ bus queueing support
- complex interface

all these mean high costs to be payed also in the entry configurations.
High memory bandwidth requirements imply:
- high speed parts
- multiway interleaving/multimodule
- queueing/buffering support
- high parallelism
- difficulties in exploiting modularity

i.e. high costs and low flexibility.
High cache requirements imply:
- high speed SRAMs
- high number of parts (big cache size) to reduce bus traffic
- long line sizes/blocks to serve bus requests
- board real estate and electrical problems

these imply high costs again.
Strong cache coherency protocol requirements imply:
- complexity in design a low overhead coherency protocol
- complexity in debugging a low overhead coherency protocol

this means high R&D costs and schedule risks.
Asymmetric memory access time means that application performances are unpredictable or at least can vary very deeply.
Performance degradation when shared data references increase implies: that the system is not well performing in some application environments.
Low scalability means difficulties in exploiting efficient scalability (i.e. cost/performances) within all possible system configurability.

stics:

- 64 Kbytes
- direct mapped
- 16 byte line size
- write through for global data
- write allocate on L1
- deffered writes
- alias tag for bus watching
- data parity
- memory cycle start on miss detection

so that it is fitted in the CPU board real estate (which is a standard double Eurocard format) along with the CPU and the bus interfaces.

According to the figure 11, it is evident that the LMA don't has any " plus " toward the MMA. The following discussion will thereforefocus mainly on two possible alternatives:

- the unique global memory (GMA)
- the multiplexed memory architecture (MMA).

5. THE PERFORMANCE ACHIEVABLE WITH GMA AND MMA

At this point we have enough information to calculate the possible aggregate system throughput, given a bus bandwidth, the CPU bus request frequency in a defined environment (TP/OA) and with different bus traffic reduction factors (i.e. with two different figures for the cache hit ratio).
A bus utilization factor of the 60% is enforced, to reasonably guarantee a minimum overhead in media usage and occupancy, and consequently limiting (specially in a packed switched bus)the latency time in getting to servers.
To summarize, a 20 MHz, eigh byte multiplexed bus and a cache with the maximum hit ratio (refer to figure 3) are needed with the global memory approach, to properly connect 4 CPUs at about 10 sustained Mips each in the TP application environment.
An other major limiting factor in reaching an H/W aggregate system throughput which increases quasi linearly in a GMA is the computation bandwidth , that is the processing power reduction due to the intrinsic latency in the service unit (memory) in satisfing the incoming requests[38].
Now we can start the discussion of the multiplexed memory architecture and start to
understand the cost/performance trade offs between the implementations of
the two architectures.
From the physical H/W point of view, the main difference between the two approaches is the presence of a unique system bus with its related global memory in the GMA and the local bus with per processor local memory and global update dispatching media in the MMA.
Giving the same memory access time, the overall latency of a CPU request is quite lower in a closely coupled local memory set-up (as in MMA) than in a classic unique global memory, linked with the CPUs through a system bus (as in GMA). In fact, while the clock frequency of the system bus is limited to 16/20 MHz by phisycal constraints (considering the analized technology and the given timeframe), the local bus can easily trak the processor frequency due to its limited and lightly loaded environment (33/40 MHz). The difference in frequency between the CPU and the system bus puts an extra cycle penalization on the GMA latency, to properly synchronize the incoming requests with the synchronous bus protocol.
Given all that, the access times that can be achieved are[39]:

MMA	GMA
340 ns	500 ns

If the memory access time is not a multiple of the system bus cycle, an other synchronization clock is needed in GMA at first data in, to properly synchronize the media with the processor. Considering all this and starting with a CPU running at 33 MHz and with a cpi of 1.8, with the same cache structure and bus traffic reduction

[38] Having considered a 60% utilization factor in the evaluation of the communication bandwidth, we can assume that the bus will not generate any significant delay, while the only effective latency is due to memory access and cycle times and the contention of multiple requests on the same memory module (memory access time streching factor).

[39] Assuming that the processors in both cases will not wait for cache burst filling, but will start the processing at the arrival of the first block of the line size.

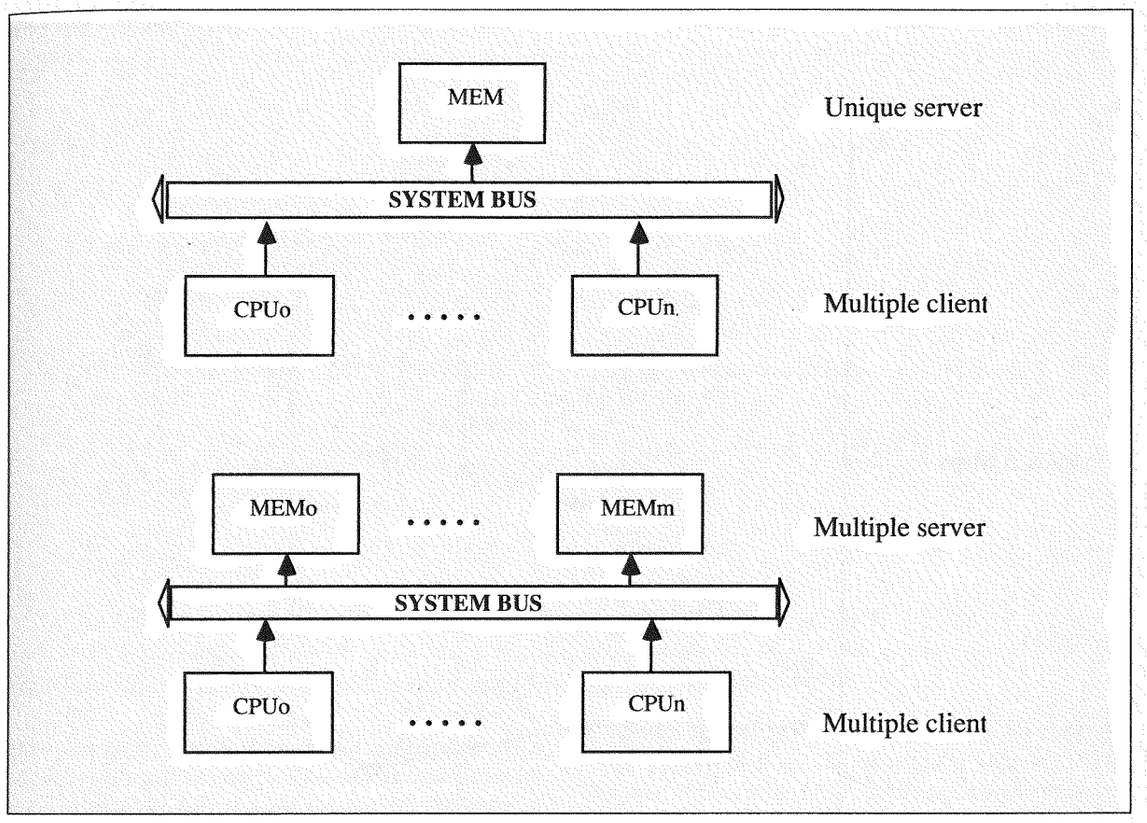


Fig. 13. The interleaved memory schema

factor, we can ultimately get less processing power degradation in a local memory set-up than in a global memory set-up (i.e. the price of misses is quite lower in MMA than in GMA).
An other factor already mentioned, but still to be considered, is the memory contention: CPU request rate and memory service rate discrepancy results in a degradation of the CPU efficiency, proportional to both the CPU and the memory module number, up to constitute an upper limit to the aggregate CPU throughput[40].
The problem of CPUs waiting in queue due to heavy main memory contention can be addressed with multimodule memory interleaving (Fig.13).

[40] The visible effect of this bottleneck is to significantly reduce the process efficiency, creating a queue of waiting CPUs on which the streching factor effect on memory access time will be quite high.

The sequential distribution of addresses among memory modules can achieve somincrease of the memory bandwidth, through simultaneous operations on different memory modules.
This concept can be somehow stretched to its limit, creating (within the global memory) a so called per processor *home memory* , located in different modules.
Anyway the described memory bottleneck is not relevant in the MMA, were the only extra contention in a CPU local memory is due to the broadcasted global writes and the I/O activity (which however, due to its low impact, we decided to not consider in this discussion). For sake of consistency, we have anyway calculated also for MMA the processor efficiency related to contentions in CPU local memory accesses due to global write broadcasting. In this case the request rate is the

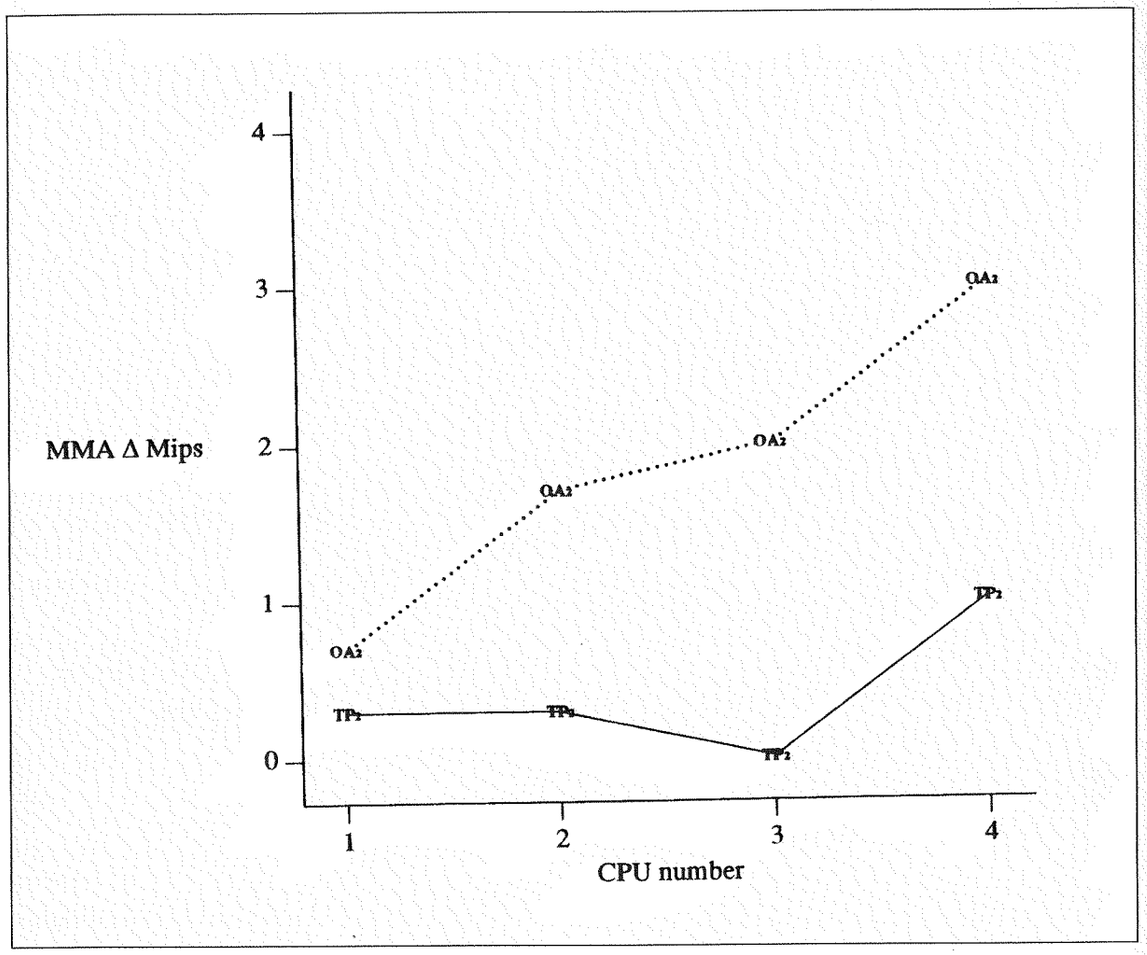


Fig. 14. MMA versus GMA delta MIPS

frequency of the write broadcast, while the service rate has been considered the frequency of bus interface pipeline spopping.

The figure 14 provides the synthesis of the performance calculation done.

What is very important is the fact tha MMA performances are a little bit better than GMA performances *even if the bulding block of the MMA are less complex (e. g. cache and system bus) than GMA building blocks!* .

6. COST/PERFORMANCE COMPARISON

Is there something magic which makes MMA more performant than GMA, even if a system based on the GMA utilizes building blocks which are equally or more (e. g. the system bus and cache) sophysticated (i.e. expensive) than the building blocks used for the MMA?

Very simply MMA outperforms GMA because make an intelligent usage of extra memory to minimize the system bus and memory bandwidth requirements.

The vive! picture sketches that:

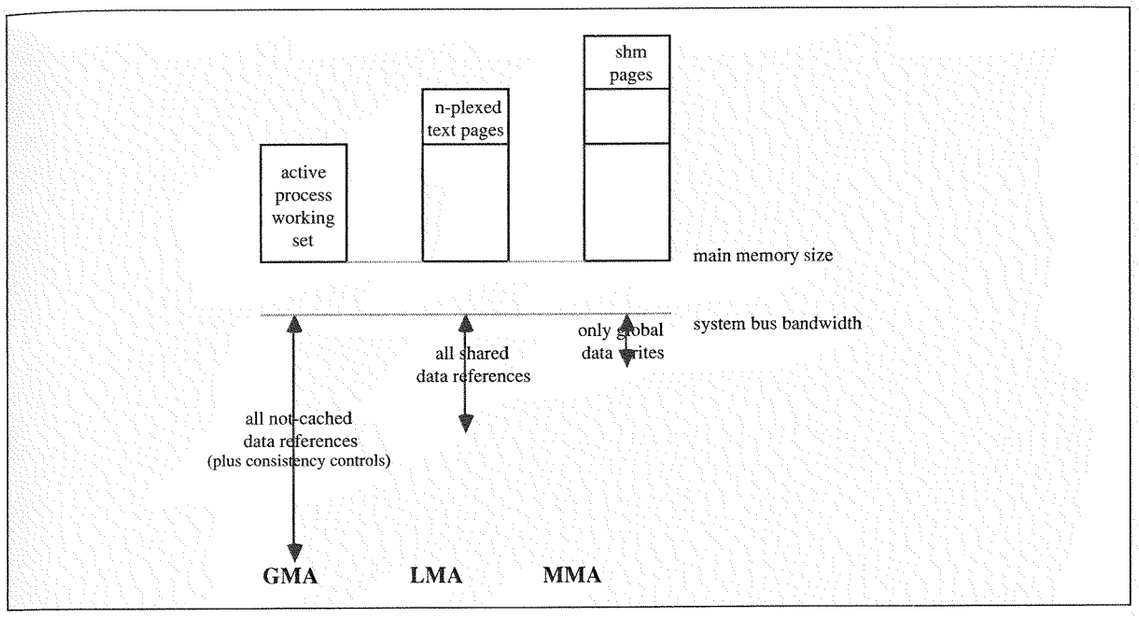


Fig. 15. Comparison different resource requirements between the three architectures

Due to the fact that the performances figures are different but also the main memory reqiurements and the cache hit ratio/bus bandwidth requirements are different, a cost evaluation is needed to understand which is the most convinient architecture, i.e. which is the architecture which exhibits the best $/Mips figure.
To compare the costs of the two solutions, we will evaluate both the cost of the main memory ($ 6.1) and the MMA memory requirements ($ 6.2). After that we will make the comparison, on the basis of two possible implementations of the MMA and GMA architectures.

6.1 The DRAM costs evaluation

The DRAM prices are/have been driven more by political implication then by real product issues.
Even if the required investiments to get up a facility for a new DRAM technology/integration generation are becoming quite high (see figure 16), and the process related problem to guarantee production can easily be underestimated, a quick calculation shows that once the technology is mature and the political implications are either setted up or overcome, the price of the product is quite reasonable and more, using standard parts, we can sure take advantage of all the benefic effects on prices, due to very high voume production (see fig 16).So trading off again the high performance and integration per chip, and so the related high cost required to build up big coherent caches in the GMA approach, against the price of standard high volumes DRAMs that need to be duplicated in the MMA set up and the intrinsic relative performances achievable in the two architectures, we will use the following figure in our evaluation:

Cost estimate for 1 Mbit Memory chip (DRAM) 100 ns	
Wafer cost	~300 $
Die size	~90K sq.mils
total die on 6 inc wafer	260
Probe yield	~60%
(assuming learning curve at 1 def/cm²)	
good die	156
Pkg cost	.2 $
Assembly yield	92%
Final test cost	.2 $
Final test yield	80 %
Total fab cost	~3 $
Selling price (assuming 50% margin)	~6 $
Trade/Political effect	~1.5/2 selling price

Fig. 16. DRAM cost

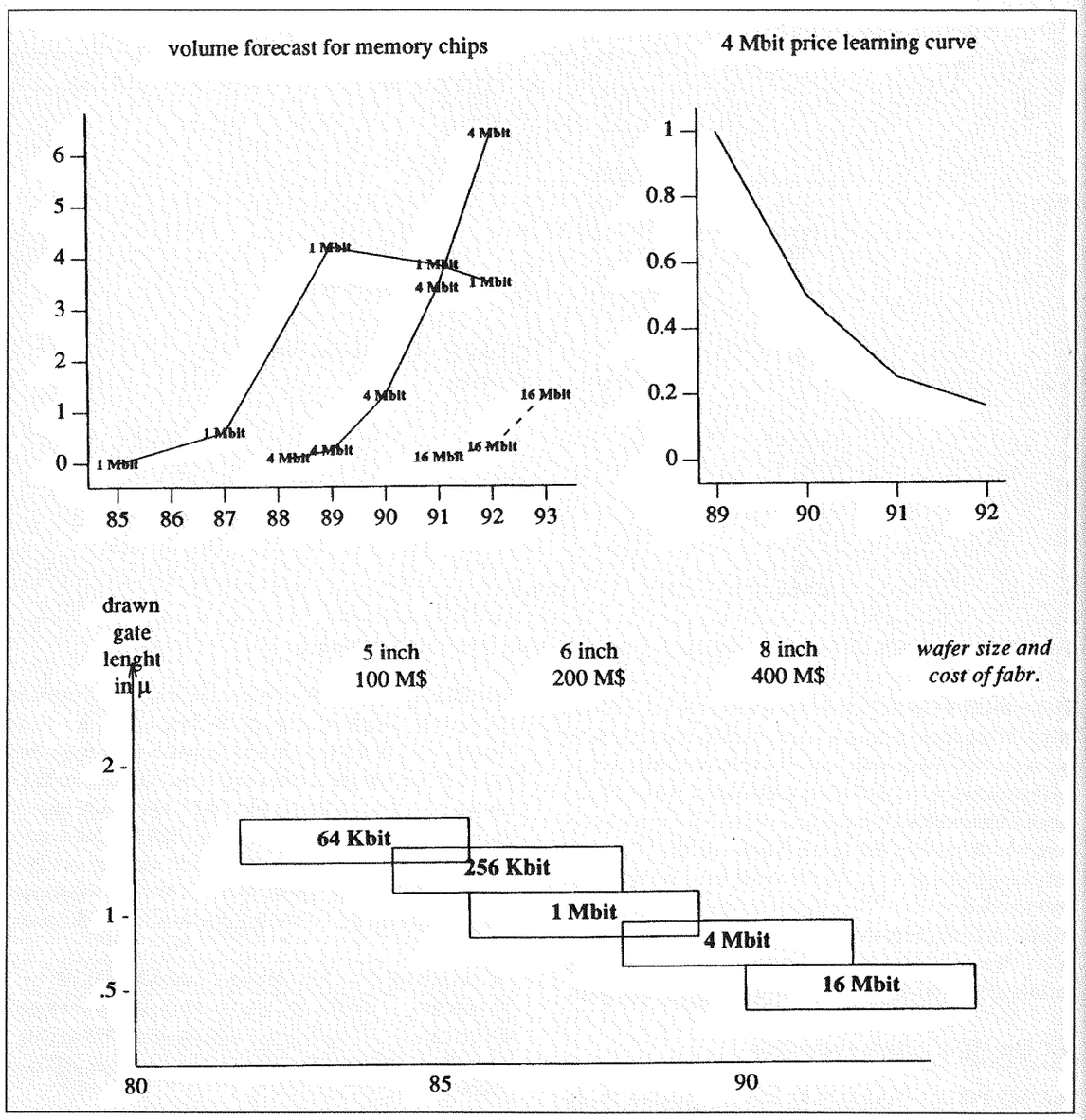


Fig. 17. Trends for DRAMs

6.2 The MMA delta memory requirements (versus GMA requirements)

The MMA extra memory requirements have the following characteristics:

- are almost proportional to the number of cPUs
- depend from the processing environment
- are negligible for a mono-CPU system configuration

We have evaluated the extra memory requirements for two processing environments (OA and TP), with the following results:

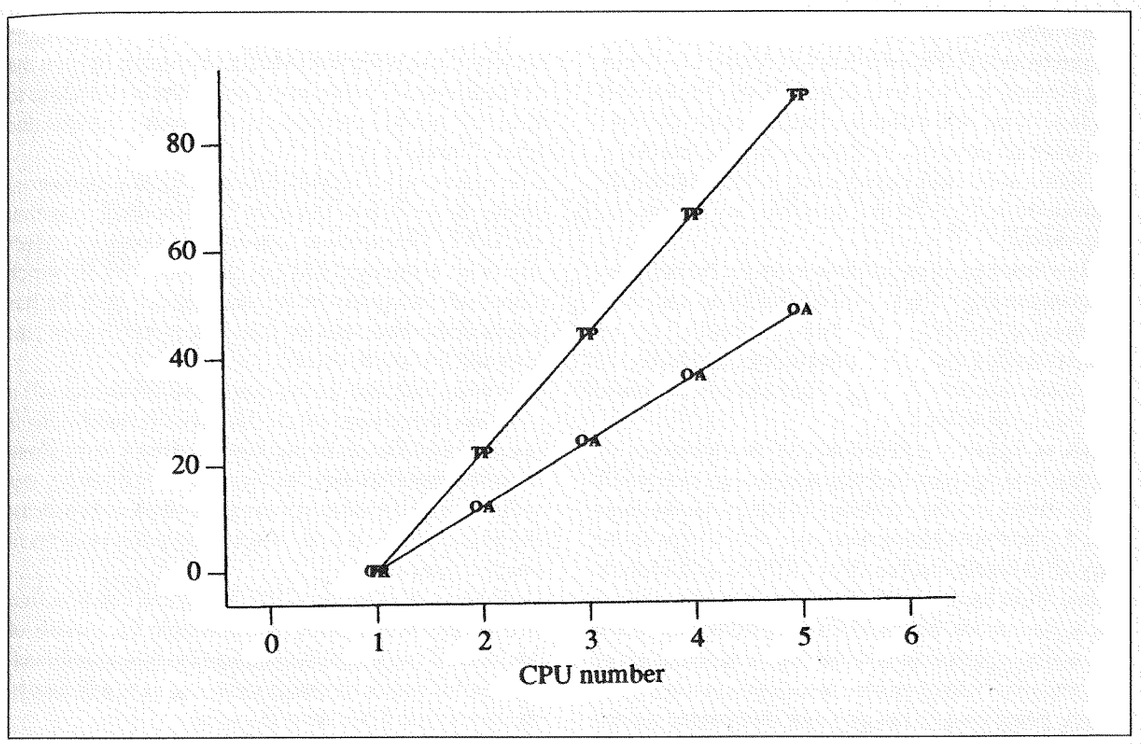


Fig. 18. MMA extra memory requirements as percentage of the GMA)

To be conservative, in the following evaluations we will use the figure:

MMA memory requirements = GMA memory requirements x (1+.22 x (n - 1))

where n is the CPU number.

6.3 An MMA based implementation: a good opportunity

All the consideration made in the previous chapter and the related support data, have shown the importance and the economic implication of the architecture and/or the selection of a "system backbone" like the system bus.
Also in this area, as well as all over through the system, we were facing a big dilemma. That is: do we use a standard bus publicly available, or a proprietary closed implementation ?
Generally speaking, using a well known standard bus we are surely making a safe decision, but with the feeling (in the future this will be more and more just a feeling) to compromise somehow the performance of the system.
On the other side, a fully custom bus could indeed maximize the system performance (this will be anyway less and less true in the near future) but at the expenses of a very costly and risky approach.
Anyway, at the time of the architecture definition (1987/1988) of the system described in this paragraph, no such a "standard system bus" to effectively sustain with the needed throughput a GMA was available on the market. The major players on the standard arena

were (and still are) VME and MBII, which can be considered high performance I/O busses[41].
At that time FUTURE bus was in its premature infancy and the NGAWG (New Generation Architecture Working Group) was not even a possibility. So the only viable solution, still within the concern outlined above, was to set-up some form of proprietary system bus to link CPUs with main memory and to fully exploit the intrinsic advantages of the standard bus for the I/O subsystems.
Is at this time that, in our effort to shoot for the best cost/performance at [sub]system level and for an optimized function modularity to support a fast technology evolution, we applied the clever methodology that lead us to the architecture definition described and rationalized in this paper: *the MMA* .
In fact, as repeately outlined before, its advantages are the drastical reduction of the system bus bandwidth requirements and the capability to achieve at the same time the best possible performance and $/Mips in the entry level configurations (1 or 2 CPUs), still keeping a quasi linear increase of CPU power at least up to four CPUs. And more, MMA gives us another real *good opprtunity: a further cost reduction through the use of the available I/O bus to broadcast global writes* .
In fact, going back to the previous discussion, wa can add that in a "full custom bus approach", the engineering can (and must) select all parameters.
But this is both an advantage and a drawback: the need to specify, qualify, document, evolve and maintain everything. Not to mention that bus problems are quite often extremely difficult to diagnose and that in the real world there is nothing worst than a back-plane design that works just most of the time.
But MMA, with the strong reduction in the system bus bandwidth requirements and with the availability within the system of an integrated, powerfull I/O bus like MBII, offers the possibility to effectively get a mixture of the two alternatives, getting the best of both. That is the usage of the available standard bus media (getting all its advantages) and add the extra feature needed to support the already optimized MMA architecture.
What we mean is to add further extensions to the definition of the standard bus, still retaining its native compatibility.
That is exactly was has been done: use available and underutilized resources (MBII), to properly sustain the defined system feature and exploit the architecture to tune the needed parameters that make feasible the above fitting.
The easiest way to handle I/O activity in a symmetric multiprocessor, is to integrate the same I/O bus within each CPU. This solution is quite more viable and effective with a standard I/O bus like Multibus II, which effectively integrates in the same silicon both the bus physical interface and the data link protocol (low part cost and low board estate).
Given the fact that in the considerd environment[42] a very low percentage of the Multibus II bandwidth is taken by the I/O activity and that its intrinsic throughput is quite large (10 MHz, 32 bit size, 40 MBytes/s), this media can be considered more than viable to broadcast the global writes: it is an effective solution in terms of cost/performance.
A 40 Mips of aggreagte CPU throughput can besustained in the MMA using the available bandwidth of Multibus II (with a 60% of utilization factor) to broadcast global writes (a global write dispatch will take in the worst case 5 Multibus II cycles).
The concept is based on a smart architecture partitioning, that implies an effective balance of cache filtered processors requests and an high speed local bus to fetch/write local data and read global data and the use of the available bandwidth of the integrated I/O bus (Multibus II) for broadcasting global writes.
Thediagram of Fig. 19 outlines the busses used by the CPU/memory subsystem.

Even within the CPU/memory subsystem defined in the MMA, critical design decision have to be made, to get

[41] Some form of loosly coupled multiprocessor can be implemented on MBII. VME was defining some powerfull CPU/memory bus extension: VSB.

[42] The profile of application environment for what concems the I/O activity can be summarized (for up to 256 connected users):

- up to 400 disk I/O per second (4 Kbyte average I/O block size)
- up to (active LANs
- up to 500 messages per second due to point connected terminal I/O (in general 1 character I/O operations).

In this environment about the 5% of the MultibusII bandwidth is used for the I/O activity

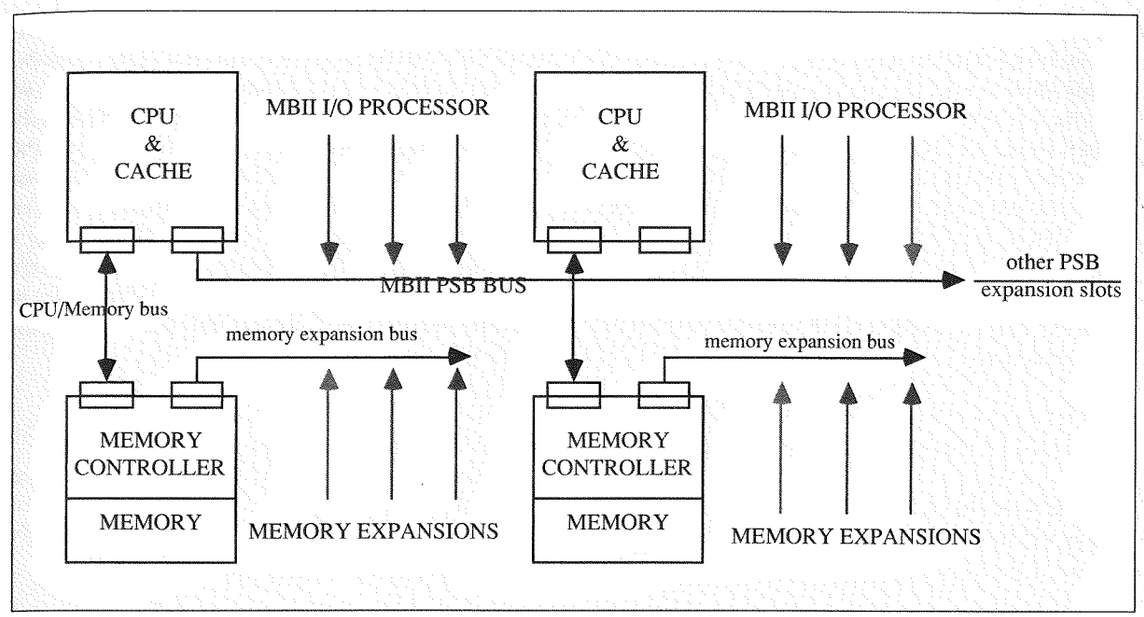


FiG 19. MMA system block diagram

the optimum price/performance ratio.
Three solution have been studied:

1) alocal, passive backplane to connect CPU/memory modules (with I/O integrated in the CPU)

2) a single board design, with on board memory expansions

3) modular memory and mother board design

Even if the first approach seems quite straightforward, the implication to integrate a CPU/memory bus with multi-slave capability (different memory controllers to be plugged for memory capacity extension) generate quite an overhead in terms of cost, complexity and packaging constrains.
The single board with mezzanine on board bus expansion seems to require the least global space and the memory access time appears to be faster. But the complexity to fet all the needed resources on a single board lead inevitably to a very high cost in mother board design and manufacturing (considering also the quite high board real estate requirements, that can also have serious impacts on packaging).
The best approach is then to effectively split the CPU function and the memory facilities (i.e. interface, timing, EDAC, etc - basic main memory) in two different standard size boards (double Eurocard size), linked by an high throughput bus (optimized on the foreseen CPU MPUs), integrating the I/O bus interface on the mother board and putting memory array expansions on a single array-bus, integrated into the memory controller. With this solution the design and manufacturing cost are lower and the system costs are variable, with additional costs reflecting higher performance and/or feature.
The great modularity of the desing allows to easily upgrade the system performance, simply changing the CPU motherboard with a more powerfull one (better operation frequency and/or throughput - RISC engine) and to fully exploit the technology evolution of DRAMS (i.e. 4 Mbit chips), just changing the memory array.
In conclusion the lower design and manufacturing costs, the incrementally tunable and balanced subsystem costs, the improved global reliability (board complexity well within manufacturing technology capability), easily offset the appearent packaging complexity and the slight performance degradation (with respect to the single board solution) due to the memory/CPU splitting on separate boards.

The so called *Modular Memory and Mother Board* approach combination provides the best flexibility in solving local (i. e. CPU - local memory, in the MMA) performance bottlenecks at reasonable cost and the needed *Mixed and Match* environment to design systems that properly fit the design needs and targets, taking full advantage of the technology evolution of both CPUs and memories at the minimum design effort.
Definitely this is the best approach to effectively fulfill the time to market need with the best up to date technology .

For what concerns then the cost implication of design complexity, we can summarize the following.

In an ideal world, the problem complexity[43] would be the unique "linear" determinant of the cost of a system.

In the real world instead, the implementation complexity strongly effects costs in a quite non linear manner. One of the major contribution to complexity [and so costs] is the "coupling" of needed functions. Generally the greater the degree of coupling, the greater is the complexity of the design andworst its structure. Even minor changes in a complex-coupled module can have unpredictable consequences, propagating through many modules and forcing changes in a unpredictable way. Considering trade offs in complexity, it is important to consider that tightly coupling introduce major problems like:

- difficulties to find errors in design
- difficulties to add new features to a system
- difficulties in integrating functionalities to enhance system performances.

Some forms of loose coupling[44] (like MMA), simplify

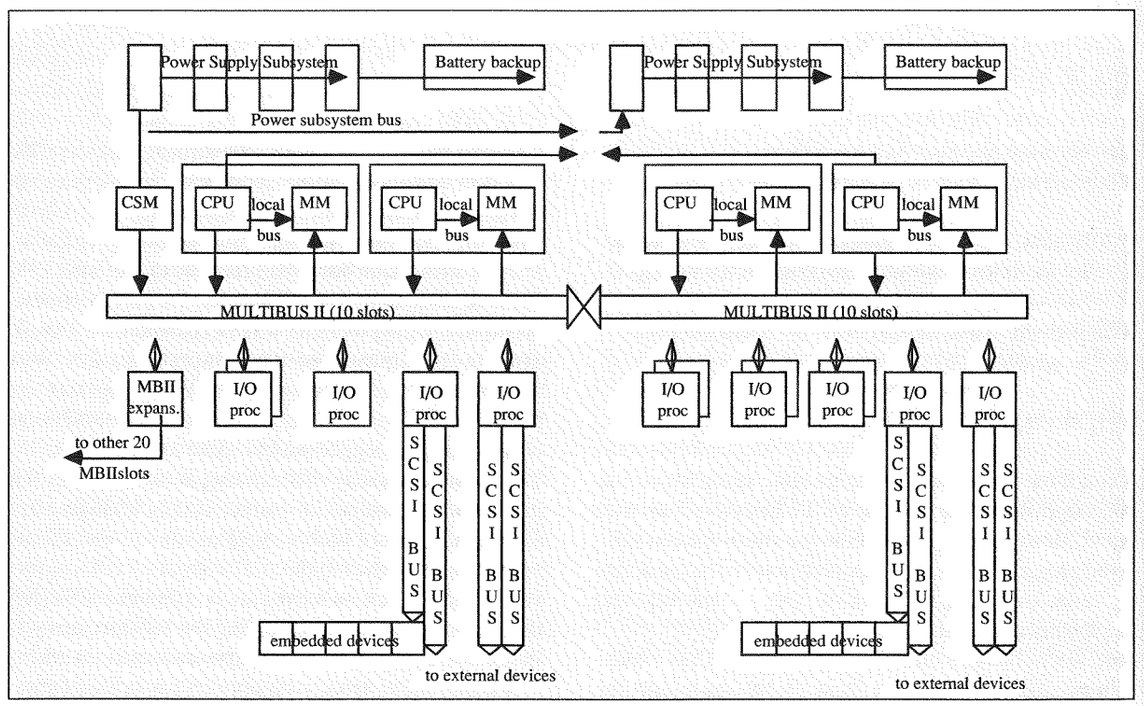


FiG 20 A. MMA based system

[43] Generally there are two different notion of complexity:
- problem complexity, that determine a minimum complexity of a possible solution
- implementation complexity , that underline that some implemen tations are more complex than others, even if they try to solve the same problem.

[44] In this context loose coupling refers to the characteristics of the module design, it is not an attribute of the multiprocessor architecture (which is tightly coupled for MMA!).

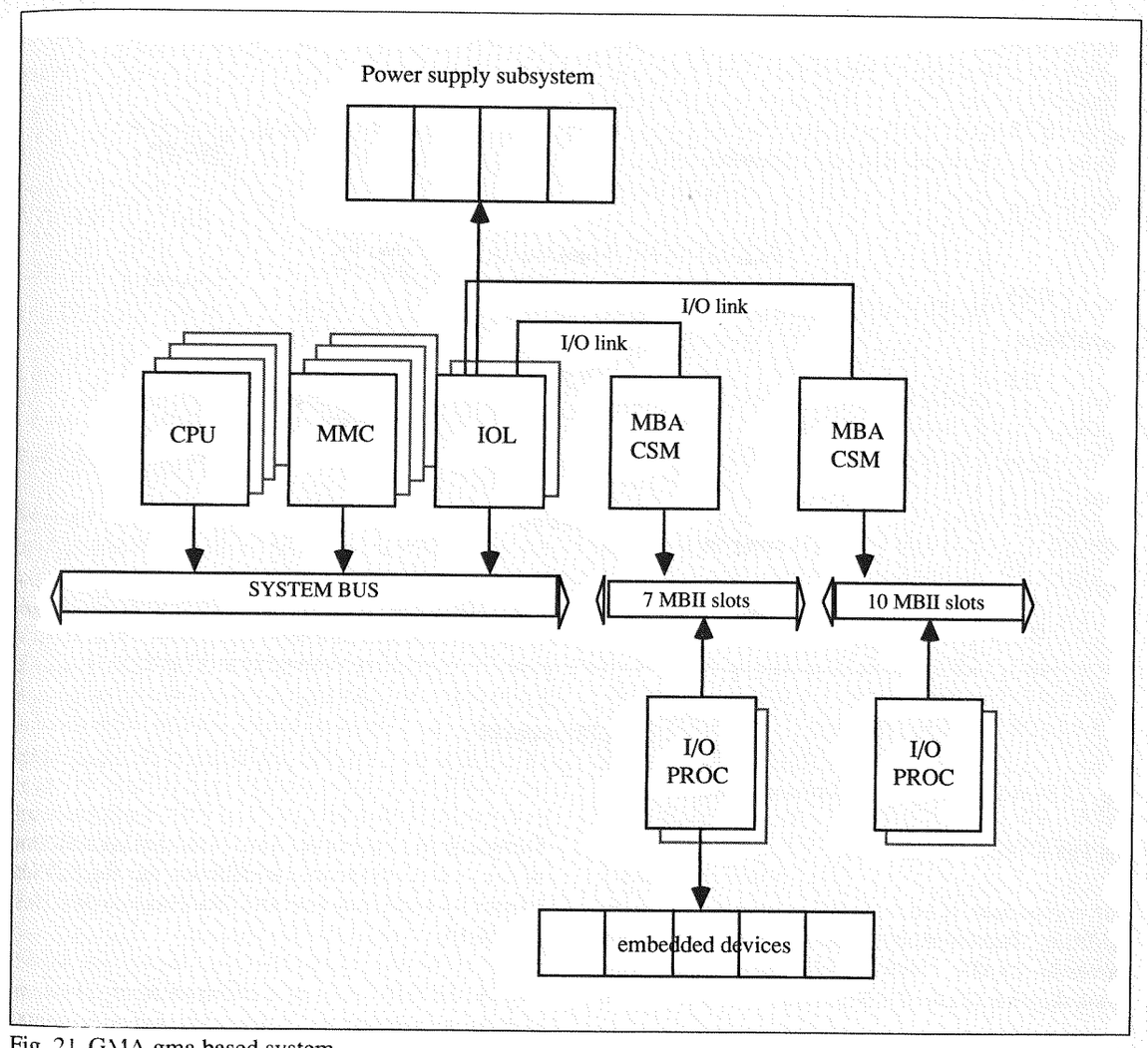


Fig. 21 GMA gma based system

maintenance, addition of new features and computer power upgrades.
One of the most valid methods to limit system complexity and related risks/costs is modularity , with its recursive (inclusive), scalable notion and its communication and sysnchronization need.
The MMA architecture is more tuned to a finer and effective implementation of modularity with the related effects on cost/performance and complexity.

The GMA instead reflects better a more complex product, with high risk in time ot get and costs overhead, specially in a not-integrated and not-standard solution.

Theof Fig.20 block diagram provides an overall picture of a system implementedon the basis of the MMA architecture.

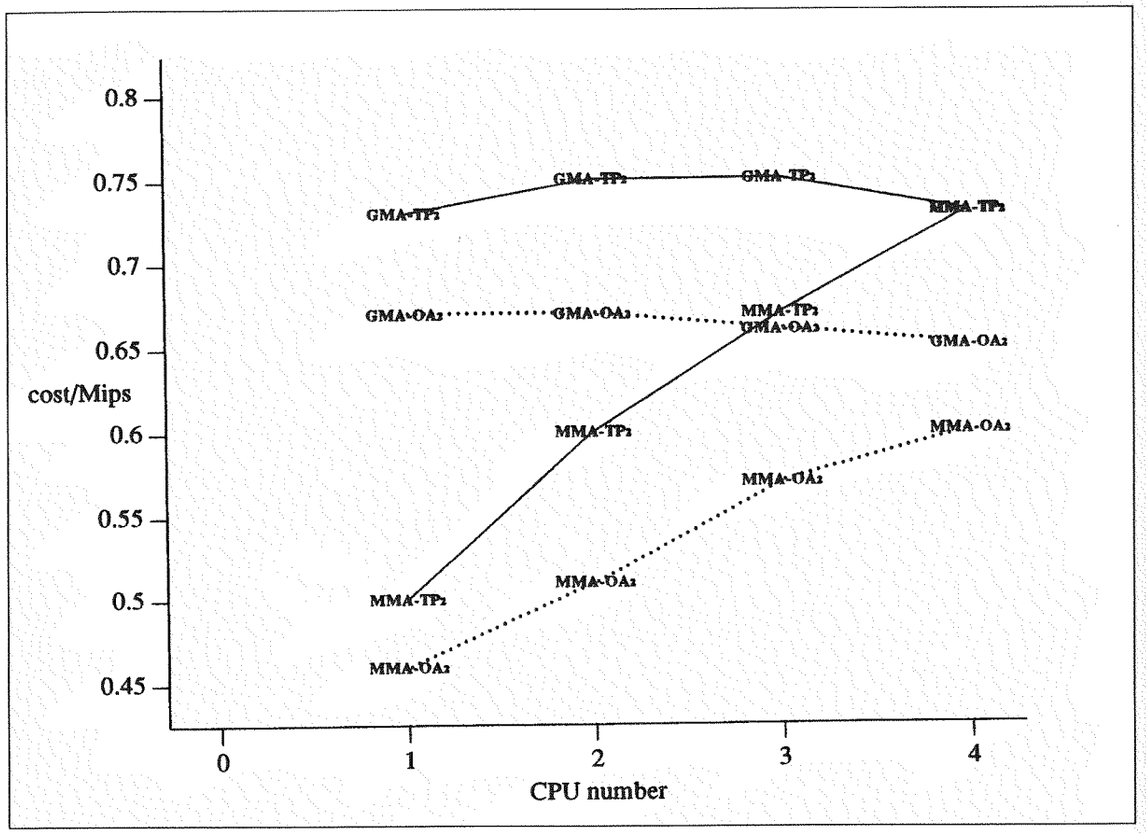


Fig. 22 MMA/GMA cost/performance comparison

6.4 The $/Mips comparison

The comparison is between the system based on the MMA architecture sketched in figure 19 and a system based on the GMA architecture, with the following block diagram illustred at Fig. 21.

notes:

- CPU and MMC are triple Eurocard size format
- each MMC has 32 Mbytes of on board memory (with 1 Mbit chips) (some modularity can be achie ved at the expenses of packaging volume)
- for the power supply subsystem is assumed the same modularity as for the MMA, but with more power per module

Designing the system, we have evaluated an indicator of the cost/performance (we did not use the $/Mips figures for marketing reasons). The diagram provides a synthesis:

One of the real values of a multiprocessor set-up, as seen by a customer, is its gracefull and incremental growth of processing power and related I/O configurability. Its need is then to have a more than optimum cost/performance in the entry point configuration and all the related advantages brought by a very modular/ scalable system resources.
The MMA represent the best fitting of this requirement, being cost/performancecompetitive in mono/dual set-up against optimized monoprocessor implementations, exploiting in all possible ways the modularity and the system resources incremental growth, and being very convinient against GMA.

From the system vendor point of view, the major objective is to optimize cost/performance of the multiprocessor offer on a design point centered on the configurations which represent the bulk of the product volume:

Percentage of MMA advantage
versus GMA (cost/performance)

CPU#	OA	TP	Volume distribution
1	31%	31%	50%
2	22%	20%	30%
3	13%	10%	20%
4	7%	0%	10%

7. CONCLUSION

We sure can not be mistaken in saying that there is no application that is as much technology driven as the computer business.
Aggressively and constantly driving its learning curve with an accelerating factor that still seems to be far from a saturation point, this technology evolution will more and more influence the methodology of standard system design and their related economy (very often the technology evolution of the computer building blocks is driven both by high volume profit application, that have nothing or little to do with the dedicated computer market, and by the intrinsic economic convinience of the technology possibilities.
Even trying strong benefits from customer inputs (system houses), silicon vendors will continue to push at its limits the produceable technology and offer to the market high quality and high integration building blocks. That will apparently constrain the possible design freedom.
So the only possible approach to fully take advantage of this promising learning curve is to effectively adapt the system architecture to the available most up to date building blocks and partition it to make use of its fast learning curve at minimum R&D costs.
For what concerns the multiprocessor/multiuser system implementation, an other key point is to precisely define the design point[s] on which optimize the architecture (within the possible configurations) and properly tune the system modularity to fulfill the product range coverage characteristics of multiprocessors. The above considerations and methodology where applied in our product definition and lead us, in the timeframe defined, to the MMA architectureherein described. Probably in the 91/92 timeframe the new technology building blocks will lead to different or improved implementations. What will not change are again the driving criterias and the methodology described in this paper.

8. REFERENCES

[1] AT&T BELL TECHNICAL JOURNAL, Vol. 63 N 8, october 1984

[2] *David R. Ditzel and H.R. McLellan* REGISTER ALLOCATION FOR FREE: THE C MACHINE STACK CACHE, ACM 1982

[3] Cedell Alexander, *William KEshlear and Furrokh Cooper* CACHE MEMORY PERFORMANCE IN A UNIX ENVIRONMENT Texas Instrument Faye Briggs, Rice University

[4] *James Archibald and Lean-loup Baer* , University of Washigton CACHE COHERENCE PROTOCOLS: EVALUATION USING A MULTIPROCESSOR SIMULATION MODEL, ACM 1986

[5] *J.Kent Peacock,* APPLICATION DICTATES YOUR CHOICE OF A MULTIPROCESSOR MODEL, EDN, june 87

[6] *Doug MacGregor, Jon Rubinstein* , A PERFORMANCE ANALYSIS OF MC68020 BASED SYSTEMS, IEEE MICRO dec 85

[7] *M.Naderi* MODELING AND PERFORMANCE EVALUATION OF MULTIPROCESSOR ORGANIZATION WITH SHARED MEMORY
IKAM UNIVERSITY, 1989

[8] THROUGHPUT ANALYSIS OF CACHE BASED MULTIPROCESSORS WITH MULTIPLE BUSSES IEEE transaction on computers, january 1988

[9] *Eric E.Johnson* COMPLETING A MIMD MULTIPROCESSOR TAXONOMY, ACM Transaction

[10]*Maurice v.Wikes,* SIZE/POWER/SPEED, ACM 1983

[11] *Peter.C.Pattom* MULTIPROCESSOR: ARCHITECTURE AND APPLICATION IEEE Computer, june 1985

[12] *Edward F.Gehrimger, Janne Abullarde, Michael H.Gulyn* A SURVEY OF COMMERCIAL PARALLEL PROCESSORS

[13] *Faye A.Briggs* SYNCHRONIZATION, COHERENCE AND EVENT ORDERING IN MULTIPROCESSORS

[14] *Cedell A.Alexander & William M.Keshlar* TRANSLATION BUFFER PERFORMANCE IN A UNIX ENVIRONMENT,Texas Instruments Faye Briggs, Rice University

EASYTUNE (*)
UNO STRUMENTO DI ANALISI IN TEMPO REALE DELLE PRESTAZIONI DI UN SISTEMA UNIX (**) MULTIPROCESSOR

Nicoletta Airenti, Carlo Leonardi, Paolo Zavatarelli
BULL HN Information Systems Italia
Pregnana (MI)

RIASSUNTO

In questo articolo viene trattato il problema dell'ottimizzazione delle prestazioni di un sistema UNIX e viene presentato uno strumento interattivo che permette al system administrator di analizzare l'attività del sistema al fine di definire una configurazione hardware e software ottimale in relazione ad uno specifico workload L'attività del sistema,intesa come utilizzo delle risorse più significative,è visualizzata in modo grafico su terminale in modo da fornire all'utente la visibilità di quello che sta facendo l' elaboratore in un dato istante.
Le stesse informazioni possono essere stampate al termine della sessione di campionamento allo scopo di permettere un'analisi più accurata dei dati rilevati. Particolarmente interessante è la possibilità in ambiente multiprocessor (2,3,4 cpu) di tracciare e visualizzare l'attività di ciascuna cpu con l'uso delle relative risorse (memoria locale, tabella dei processi, ecc...).
Tale strumento, noto col nome di EASYTUNE, èstato sviluppato in ambito UNIX System V Rel. 3.1 ed è ufficialmente distribuito sui sistemi BULL.

ABSTRACT

In this paper we discuss the problem of performance optimization of a UNIX system and we present an interactive tool which allows the system administrator to analyse the system activity in order to define an optimal hardware and software configuration under a specific workload.
The system activity, viewed as the utilization of the most significant resources, is shown in a graphic way on a terminal. In this way the user can see the computer activity in a given instant. The same informations can be printed at the end of the sample in order to allow a more accurate analysis of the data. The possibility in a multiprocessors enviroment to trace and view the activity of each CPU with the usage of its resources (local memory, processes table,...) is particulary interesting.
This tool called EASYTUNE was developed on UNIX System V Rel. 3.1 and it is distribuited on BULL systems.

1. L'OTTIMIZZAZIONE DELLE PRESTAZIONI SUI SISTEMI UNIX

La crescente diffusione dei sistemi UNIX nell'ambito delle attività produttive induce a studiarne e ove possibile migliorarne le prestazioni. Quest'ultimo obbiettivo può essere raggiunto attraverso un corretto dimensionamento delle risorse hardware e software delle quali il sistema necessita per soddisfare nel minor tempo possibile le esigenze del carico utente.
Le risorse hardware che tipicamente vengono coinvolte sono la dimensione della memoria, il numero, la capacità e il tempo di accesso delle unità disco, il numero e la potenza delle cpu.

(*) EASYTUNE è un trademark di Bull HN Information Systems Italia S.p.A.
(**) Unix è un trademark della AT&T Bell Labs.

Per quanto riguarda le risorse software, il sistema UNIX [1] è molto flessibile e dà all'utente la possibilità di stabilire i valori di un insieme di parametri che controllano le principali attività di sistema. Il system administrator, ad esempio, può definire il time slice dei processi, dimensionare l'ampiezza delle tabelle usate dal kernel per memorizzare le informazioni su processi, files, ecc..., intervenire sull'attività di I/O attraverso il dimensionamento della buffer cache. E' chiaro quindi che definire con accuratezza i valori dei parametri configurabili è essenziale per ottenere buone prestazioni e sfruttare appieno la potenzialità della macchina.
Tale operazione tuttavia è molto complessa in quanto implica una conoscenza approfondita dei meccanismi di sistema e delle relazioni che intercorrono tra le risorse, nonché una conoscenza del significato e dell'effetto indotto dai parametri che UNIX permette di modificare.
Nel caso che il sistema abbia un'architettura multiprocessor le difficoltà del tuning aumentano in modo considerevole e diventa indispensabile essere in grado di ottenere un insieme di informazioni caratterizzanti lo stato del sistema e delle singole cpu.
Queste motivazioni ci hanno indotto a sviluppare uno strumento in grado di tracciare l'utilizzo delle risorse hardware e software del sistema durante l'esecuzione di un determinato workload e di fornire quindi informazioni al system administrator su come configurarlo correttamente.
Il flow chart contenuto in fig.1.1 mostra come la configurazione ottimale del sistema sia strettamente connessa ad un'attività di campionamento e analisi dello stato del sistema stesso durante l'esecuzione di un determinato workload.

2. L'ARCHITETTURA

Dal punto di vista architetturale il prodotto è costituito da un driver, ovvero un modulo del sistema operativo che generalmente è adibito alla gestione delle periferiche ma che può essere utilizzato come canale di comunicazione tra kernel e programma utente, e da due

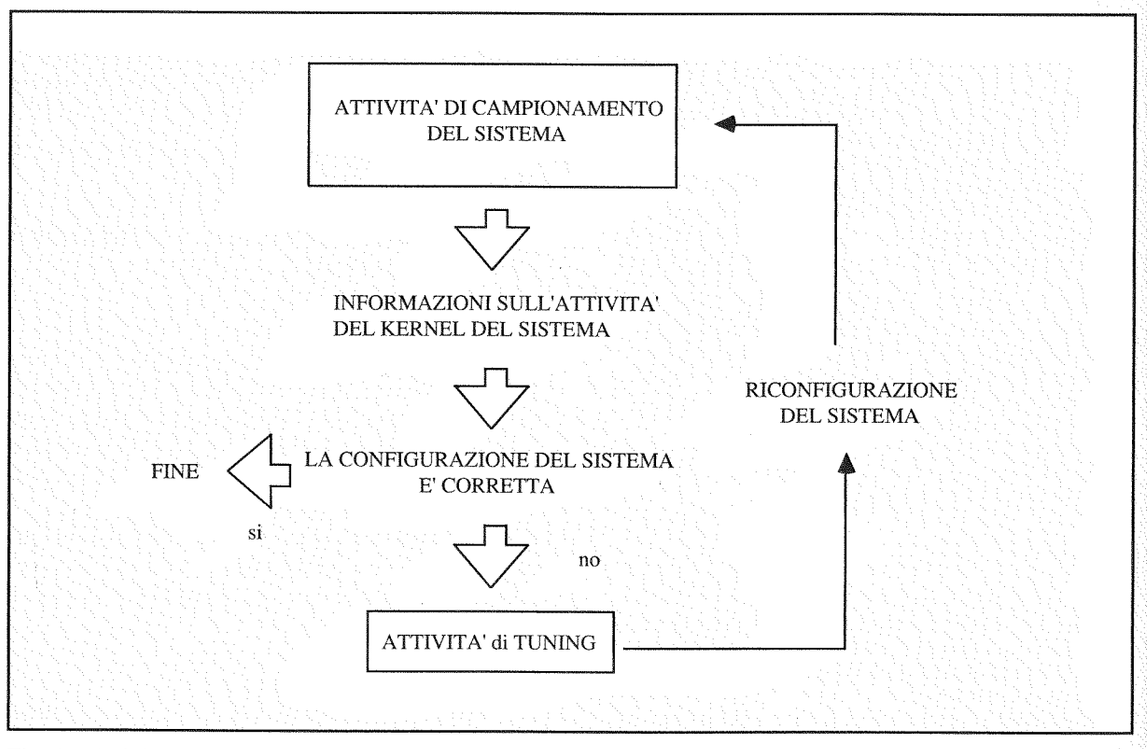


Fig. 1.1 Attività di tuning del sistema

processi utente aventi rispettivamente il compito di estrarre ed analizzare i dati (fig. 2.1).
Questi due processi sono quelli direttamente visibili all'utente ed interagiscono con il driver attraverso le normali primitive di sistema read,open,close,ioctl.
L'insieme dei dati estratti è formato da informazioni contenute nelle varie tabelle di sistema e da valori assunti da contatori, inseriti appositamente nel codice del kernel, in grado di registrare particolari eventi di sistema.
Il driver viene attivato ad intervalli regolari di tempo dal processo estrattore che provvede alla memorizzazione parziale o totale dei dati su disco ed eventualmente al passaggio dei dati al processo analizzatore che a sua volta ha il compito di effettuare elaborazioni statistiche sui dati e di renderli visibili all'utente tramite grafici o listati.
L'intervallo intercorrente tra due campionamenti può essere definito dall'utente in funzione dell'overhead che vuole introdurre con l'attività di misurazione e del grado di affidabilità che vuole raggiungere nei risultati.
L'intervallo di campionamento minimo è un secondo e questo tempo minimo corrisponde al tempo di ciclo di alcune attività di kernel quali le operazioni di ricalcolo delle priorità dei processi o la schedulazione di particolari processi kernel.
EASYTUNE è stato sviluppato in modo modulare allo scopo di mantenere disgiunta l'attività di campionamento da quella di analisi.
Un utente quindi può decidere di utilizzare solo il processo estrattore per un'attività di campionamento con un overhead minimo sul sistema, può richiamare solo il processo analizzatore per esaminare dati già rilevati in precedenti sessioni di misura oppurepuò usare contemporaneamente entrambi i moduli richiedendo un'analisi in tempo reale dello stato del sistema [2].

3. LE FUNZIONI DI EASYTUNE

Le funzioni di EASYTUNE possono essere sintetizzate come segue:

- campionamento interattivo
- campionamento in background
- analisi differita

Selezionando la prima funzione l'utente innesca il campionatore ed ha contemporaneamente la possibilità di analizzare in tempo reale lo stato del sistema in un determinato momento in quanto vengono forniti i livelli di occupazione istante per istante delle singole risorse. I dati vengono, su richiesta dell'utente, memorizzati su

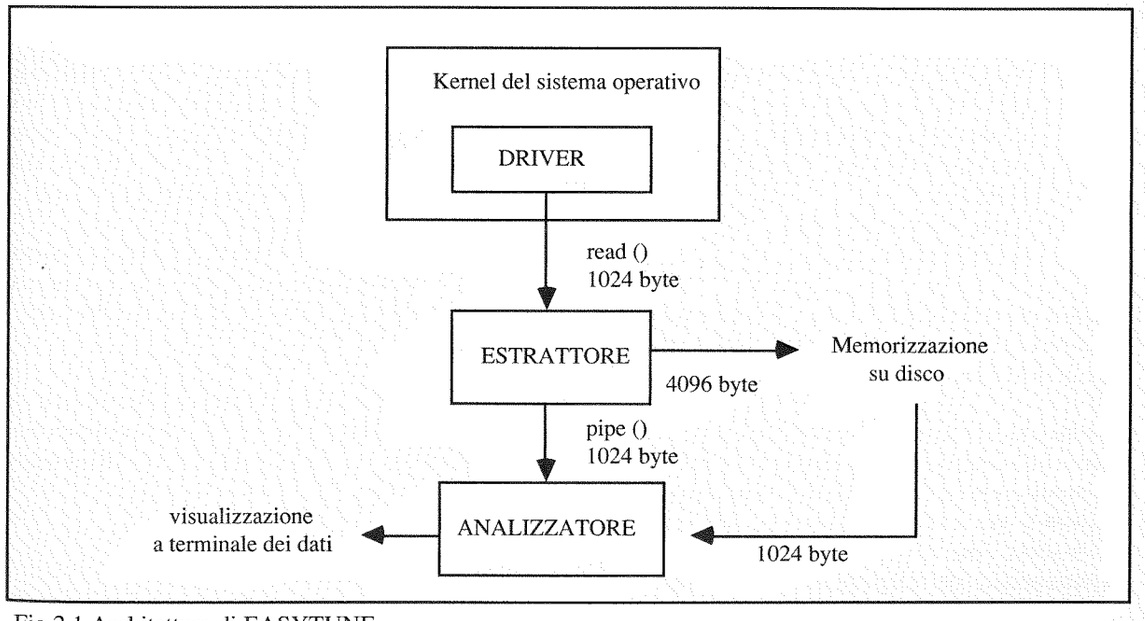


Fig.2.1 Architettura di EASYTUNE

file e possono essere riutilizzati all'interno della corrente sessione di campionamento o possono essere analizzati utilizzando la modalità di analisi differita.
Con la seconda opzione invece viene attivato unicamente il processo estrattore, i dati vengono memorizzati su disco e all'utente viene riproposto il prompt della shell in modo che sia libero di effettuare altre attività in attesa che termini la sessione di campionamento. Solo in un momento successivo l'utente potrà vedere tali dati attraverso la visualizzazione su terminale o la produzione di stampe. Come già accennato in precedenza questo tipo di funzionalità comporta il minimo overhead sul sistema. L'analisi differita permette di rivedere i dati memorizzati in precedenti sessioni di campionamento. L'analisi dei risultati può essere effettuata a terminale oppure può essere generato un insieme di stampe in grado di caratterizzare lo stato del sistema.

4. RISORSE CAMPIONABILI

I dati estratti ed analizzati da EASYTUNE sono strettamente collegati alle problematiche di configurazione e quindi alla possibilità di riuscire a caratterizzare nel modo più preciso possibile il workload a cui è sottoposto il sistema.
Essi possono essere suddivisi in cinque categorie:

- risorse software (buffer cache, inode table, file table, process table)
- risorse hardware (pagine di memoria, occupazione di cpu, occupazione area di swap)
- attività di paginazione
- attività di I/O su device disco e sui controller di linea
- eventi di sistema utili alla caratterizzazione del workload

Vediamo brevemente di inquadrare la funzione di questi dati nella determinazione delle caratteristiche del sistema e quindi nella ottimizzazione del suo utilizzo.

Il ruolo giocato dalle risorse software è soprattutto relativo all'occupazione di memoria del kernel, quindi determina quale parte della memoria fisica installata è utilizzabile per contenere i processi dell'utente. In questo senso, i dati sull'utilizzo delle tabelle di sistema possono portare alla riduzione delle loro dimensioni, a tutto vantaggio della quantità di memoria centrale a disposizione dell'utente.

Il secondo gruppo logico di dati è invece relativo all'uso delle risorse hardware. Poter sapere in ogni istante quanta parte della memoria fisica è utilizzata può essere utile ad esempio a stabilire se la configurazione hardware della macchina è insufficiente e quindi se sia necessario potenziarla per poter sostenere efficientemente il carico.

L'attività di paginazione dovuta al traferimento di pagine di memoria da e verso disco per sopperire a mancanza di memoria centrale (attività di swap), crea nel sistema già carico un ulteriore appesantimento, che non è di per sé legato allo svolgimento dell'attività utente. Questa funzionalità di sistema inoltre utilizza piuttosto pesantemente primitive di I/O, che sono molto lente se paragonate alla velocità di calcolo interno.
Il tuning ottimale di un workload dovrebbe perciò limitare al minimo la presenza di attività di swap nel sistema.

Un altro "collo di bottiglia" che può presentarsi in una applicazione è l'utilizzo non bilanciato delle periferiche ed è per questo motivo che un altro gruppo di dati si riferisce al traffico di I/O sulle periferiche locali (tipicamente i dischi) e al traffico di linea sui controllers.

L'ultimo gruppo di informazioni riguarda particolari eventi di sistema come context switch, page fault, fork (creazione di un nuovo processo), ecc.. Queste informazioni sono utili per riuscire a caratterizzare con precisione il tipo di workload presente nel sistema.

L'insieme di tutti questi gruppi di dati fornisce quindi un quadro completo sulla situazione istante per istante dell'utilizzo della macchina da parte di una data applicazione.

Le tabelle 1 e 2 riportano in dettaglio i dati rilevati sia per quanto riguarda le variabili legate alla configurazione software del sistema che quelle relative all'intervento di particolari funzioni di kernel.

5. VISUALIZZAZIONE DEI DATI

L'utilizzo delle risorse e le attività di sistema più

significative vengono visualizzate tramite istogrammi allo scopo di rendere la lettura più immediata e intuitiva.
La maschera creata per visualizzare tali istogrammi contiene due assi cartesiani sui quali viene effettuata la proiezione dei risultati dei campionamenti.Sull'asse verticale viene proiettata in percentuale l'occupazione della risorsa presa in esame rispetto al valore di configurazione,sull'asse orizzontale viene riportato il numero di campionamento e il valore della risorsa relativo (fig.5.1). Nel caso l'utente stia effettuando un'analisi in real time la visualizzazione avviene mediante "scroll automatico" verso sinistra dei dati.Ciò non accade nell'analisi differita in quanto i campionamenti sono visualizzati in modo statico.
Alcune risorse o attività di sistema sono analizzate per ogni singola cpu. In questo caso la maschera contenente gli istogrammi sarà suddivisa in 2,3,4 quadranti a seconda del numero di cpu presenti sul sistema (fig.5.2).
Tutti i dati relativi alle risorse di sistema analizzate da

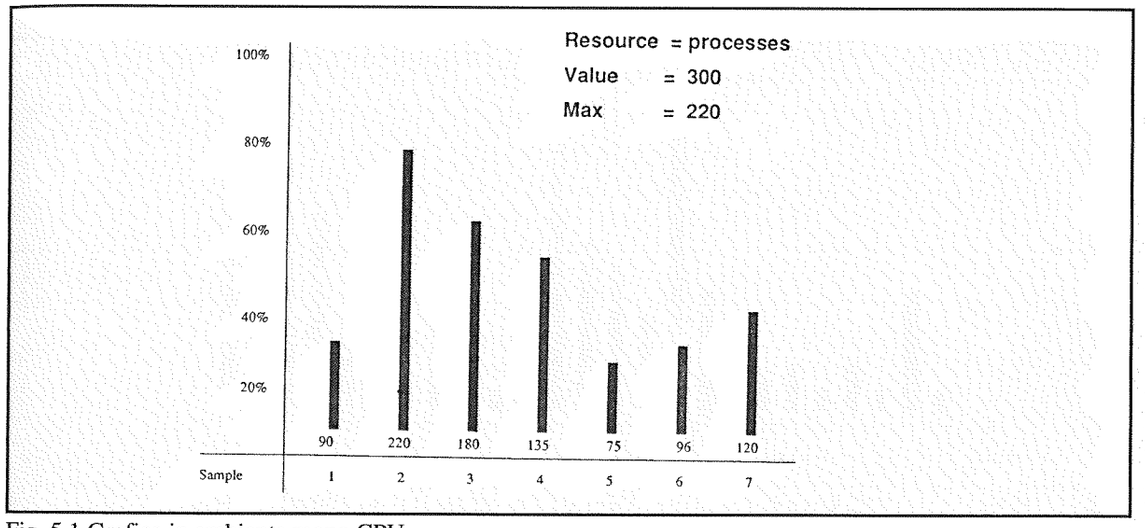


Fig. 5.1 Grafica in ambiente mono CPU

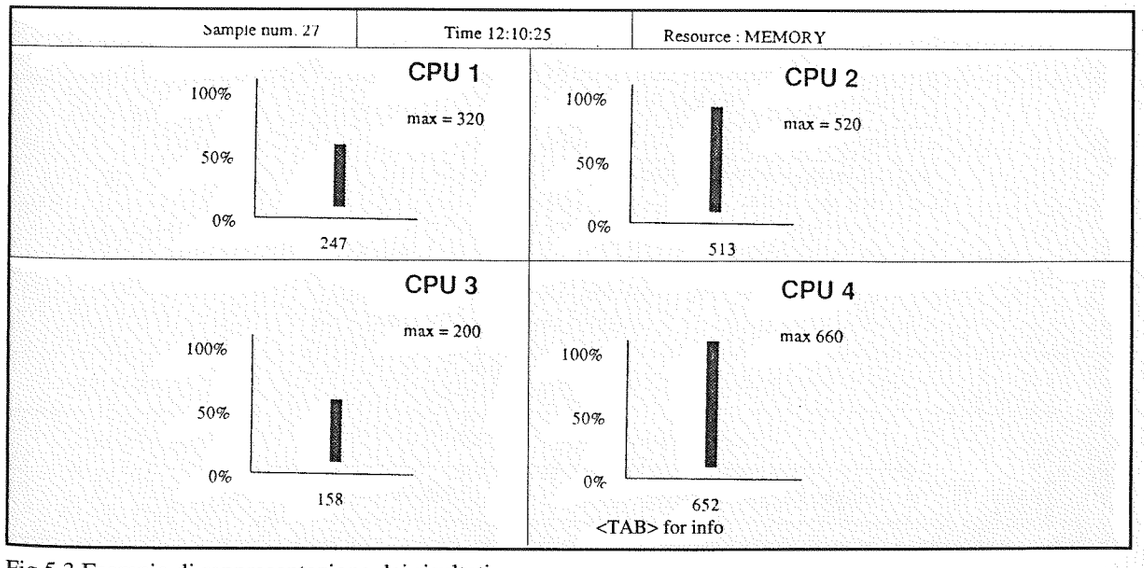


Fig 5.3 Esempio di rappresentazione dei risultati

EASYTUNE possono essere rappresentate in modo statistico attraverso un algoritmo di clustering.In questo caso nella maschera sono presenti due assi cartesiani che riportano sull'asse delle ordinate l'utilizzo percentuale rispetto al valore di configurazione o nel caso che non ci sia, rispetto ad un massimo fissato dall'algoritmo di clustering, sull'asse delle ascisse il periodo di campionamento diviso automaticamente in gruppi di intervalli uguali.Tali grafici, oltre ad essere visualizzati su terminale, possono essere stampati (fig.5.3).

6. L'INTERFACCIA UTENTE

L'interazione tra l'utente e EASYTUNE avviene mediante una serie di menu gerarchici realizzati tramite finestre che via via vengono a sovrapporsi alle precedenti in modo parziale in modo che l'utente possa sempre ricostruire la successione delle operazioni compiute.
In figura 6.1 è visualizzato un esempio di utilizzo di EASYTUNE. L'utente ha selezionato nel primo menu

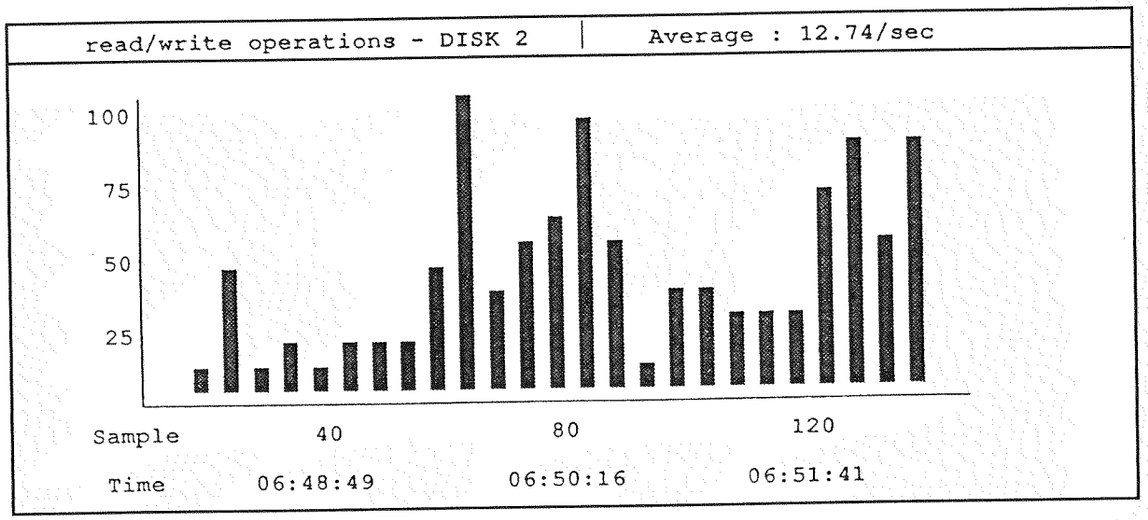


Fig. 5.2. Grafica in ambiente con quattro CPU

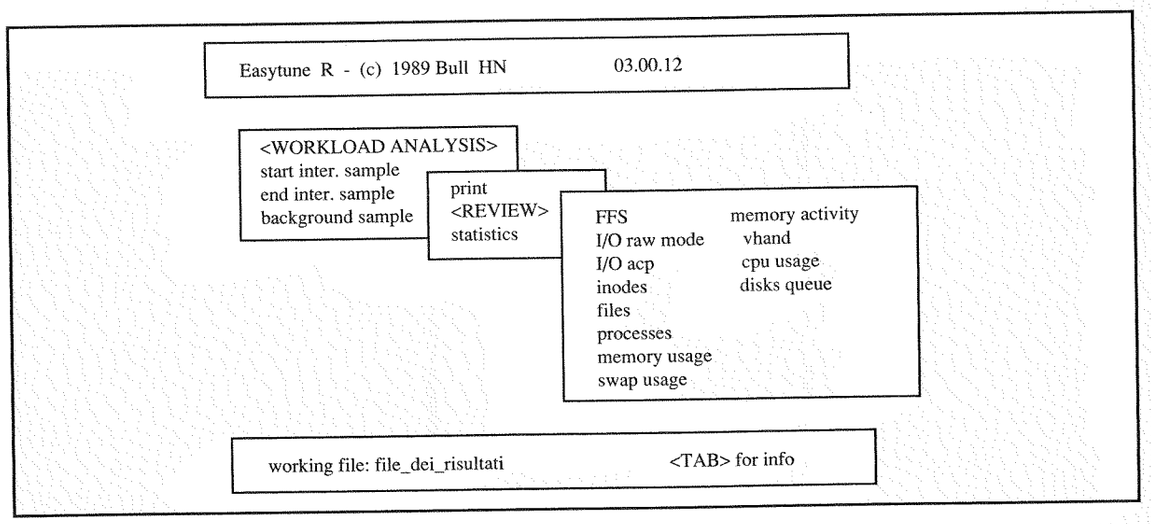


Fig 6.1 Esempio di interfaccia utente

l'opzione di "workload analysis" che permette di analizzare un file di dati creato in una precedente sessione di campionamento. Nel terzo menu compaiono invece le risorse che possono essere analizzate. da notare che la presentazione di menu gerarchici per la scelta delle varie opzioni e la rappresentazione mediante istogrammi dei risultati rendono l'utilizzo di EASYTUNE estremamente semplice e quindi adatto anche ad utenti con minime conoscenze dei sistemi Unix.

7. RINGRAZIAMENTI

Si ringraziano il Prof. Gianni Degli Antoni, il Dott. Le Van Huu e la Dott.ssa Paola Zocco Ramazzo per i contributi apportati alla realizzazione di questo lavoro.

8. BIBLIOGRAFIA

[1] Bach M. J., THE DESIGN OF THE UNIX OPERATING SYSTEM, PRANTICE HALL, ENGLEWOOD CLIFFS, NJ, 1986

[2] Zavatarelli P., ANALISI DI DATI SULL'UTILIZZO DI RISORSE ORIENTATA AL TUNING DI UN SISTEMA UNIX IN AMBIENTE MULTIPROCESSOR, Tesi di Laurea, Scienze dell'Informazione, Milano Anno Accademico 1986-87.

buffer cache	numero di elementi in DELAYED WRITE numero di elementi BUSY percentuale di "hit" nella buffer cache relativamente alle operazioni di I/O
inode/file	numero di "entries" occupate
processi	processi attivi per cpu lunghezza della run-queue per cpu numero di fork eseguite per cpu numero di processi trasportati tra cpu numero di contex switch eseguiti per cpu
memoria	numero di pagine occupate per cpu
uso delle cpu	percentuale di utilizzo totale percentuale di utilizzo in kernel/user
attività di swap	numero di pagine occupate su disco numero di pagine liberate con/senza I/O per cpu numero di pagine lette dall'area di swap per cpu
attività di I/O	numero di operazioni logiche/fisiche numero di operazioni in raw mode numero di richieste in coda per device numero di operazioni di read/write per device
I/O di linee	utilizzo percentuale di ogni processor di linea
gestione memoria	numero di page fault per cpu numero di pagine lette da file per cpu numero di pagine lette da swap per cpu numero di pagine trovate in cache

Tab. 1 Valori di rilevati ad ogni campionamento

numero di buffer di sistema
numero di entries della FILE TABLE
numero di entries della INODE TABLE
numero di entries della PROCESS TABLE
numero di pagine di memoria utilizzabili
numero di dischi connessi al sistema
valori di configurazione utilizzati per regolare l'attività di swap

Tab. 2 Valori di configurazione rilevati

UN ALGORITMO DI LOAD BALANCING PER SISTEMI UNIX MULTIPROCESSOR

Le Van Huu (*) - Paola Rossaro (*) - Carlo Leonardi (+)

RIASSUNTO

In questi ultimi anni, si è evidenziata una crescente diffusione delle architetture multiprocessor.
Un problema rilevato dallo studio delle prestazioni fornite da tali architetture, riguarda la distribuzione del carico di lavoro.
Accade di frequente che una o più CPU risultino sovraccariche mentre altre siano quasi o completamente inattive.
In questo articolo è descritto un algoritmo di bilanciamento del carico utente attraverso il trasferimento di processi tra le varie CPU.
Si sono evidenziate le principali problematiche legate alla definizione dell'algoritmo, le politiche adottate e gli aspetti implementativi.
Vengono infine forniti dei risultati esemplificativi, ottenuti utilizzando l'algoritmo.

ABSTRACT

Recently, the increasing development of multiprocessor architectures has given system engineers new problems to solve.
One of these concerns performance evaluations with respect to workload distribution.
In fact, it may frequently happen that one CPU is idle, while another is very busy.
This paper presents a load balancing algorithm that distributes the workload by transferring processes among different CPUs.
We describe the main problems regarding algorithm definition, adopted solutions and implementative aspects.
Finally, some results obtained using the algorithm are given.

(*) Dipartimento di Scienze dell'Informazione - Università degli studi di Milano

(+) System Engineering - BULL HN italia - Pregnana Milanese

1. INTRODUZIONE.

Le prestazioni di un sistema multiprocessor sono influenzate da diversi fattori, riguardanti sia l'architettura hardware, sia le politiche adottate dal sistema operativo.
Sebbene sia praticamente impossibile ottenere un throughput complessivo pari alla somma dei throughput delle singole CPU, occorre sfruttare al meglio le potenzialità che esse offrono.
E' stato dimostrato [1] per i sistemi distribuiti, con reti formate da nodi omogenei, utilizzati a medio carico, che accade di frequente che alcuni nodi si trovino sovraccarichi mentre altri sono quasi o totalmente inattivi. Si è verificato sperimentalmente un risultato analogo anche per le architetture multiprocessor.
Un carico ben distribuito consente quindi un miglioramento delle prestazioni del sistema con una conse-

guente diminuzione dei tempi di risposta.
Gli algoritmi di load balancing hanno lo scopo di bilanciare il carico di lavoro attraverso il trasferimento di processi utente tra le CPU.
Da molti anni il problema del bilanciamento dei carichi di lavoro ha interessato l'area dei sistemi distribuiti; di conseguenza gran parte degli algoritmi di load balancing esistenti riguardano solamente questa area, mentre per quanto concerne i sistemi multiprocessor, le soluzioni sono ancora poche.
In questo articolo presenteremo una di queste soluzioni, rappresentata da un algoritmo realizzato presso i laboratori di Ricerca e Sviluppo della BULL HN Italia di Pregnana Milanese, in collaborazione con il Dipartimento di Scienze dell'Informazione dell'Università degli Studi di Milano.
Sono stati utilizzati degli elaboratori della BULL HN Italia con una architettura multiprocessor con configurazione massima di quattro CPU, completamente simmetriche tra loro, con memoria locale condivisa, connesse attraverso il bus di sistema Multibus II. L'architettura del sistema è rappresentata dalla Figura 1.
Il sistema operativo utilizzato è UNIX System V, adattato per ambienti multiprocessor, dato che l'UNIX originale della AT&T Bell Laboratories non prevede, al momento, la gestione di architetture a più CPU.
In questa versione, ogni CPU possiede una copia del sistema operativo; la gestione delle risorse condivise e i problemi di concorrenza sono risolti da strutture e primitive semaforiche.
La definizione dell'algoritmo in esame ha richiesto attenti studi volti a risolvere i seguenti problemi basilari:

1) scelta della politica di bilanciamento (per esempio statico o dinamico);
2) caratterizzazione del carico;
3) definizione dei criteri per decidere del trasferimento di un processo;
4) scelta del processo da trasferire.

Nei successivi paragrafi cercheremo di presentare, in dettaglio, le problematiche elencate, insieme alle relative soluzioni. Per ciò che riguarda la terminologia, faremo riferimento alla letteratura esistente sui sistemi distribuiti poiché alcuni dei problemi ad essi legati sono analoghi a quelli riscontrati sui multiprocessor. In particolare sono utilizzate le definizioni relative alle varie politiche di bilanciamento e ai principali criteri adottati per il trasferimento dei processi.

2. PROBLEMATICHE

Politiche di bilanciamento.

Gli algoritmi di load balancing si suddividono in due categorie principali [2], in base alle politiche di assegnamento seguite.
La prima categoria è composta dagli algoritmi di load balancing ad *assegnamento statico* (static load balancing); agiscono indipendentemente dalle condizioni attuali del sistema e applicano una distribuzione dei processi in base a studi statistici. In pratica, ciò avviene stabilendo per ogni utente, in modo più o meno permanente, lo specifico processore dove il suo lavoro deve essere eseguito. Un secondo metodo è quello di dedicare particolari classi di processori a determinate classi di processi. Quest'ultimo meccanismo può applicarsi ad esempio ad un'architettura con processori front-end o back-end, dedicati a specifiche funzioni. Ovviamente una volta stabilito il criterio per la distribuzione del carico, esso rimarrà invariato per un intervallo di tempo relativamente lungo (giorni, settimane, ecc.). Il costo di questi algoritmi non è elevato e non incide sull'overhead del sistema. Le prestazioni, però, non sempre risultano ottimali, poiché non si basano sulle conoscenze dello stato reale del carico.
La soluzione a queste limitazioni ha portato alla definizione di nuovi algoritmi che potessero rispondere in modo più "fedele" alle esigenze del sistema.
Infatti, la seconda categoria degli algoritmi di load balancing è composta dagli algoritmi ad *assegnamento dinamico* (dynamic load balancing), che decidono la distribuzione dei processi in base alle condizioni attuali del carico, effettuando dei controlli a run-time.
Questa categoria si suddivide a sua volta in due sottoclassi principali. La prima prevede il trasferimento dei processi solo durante la fase di creazione degli stessi. Si parla quindi di algoritmi di dynamic load balancing a *piazzamento iniziale*. Quando il processo è creato, è il sistema operativo che lo assegna a qualche processore in modo del tutto trasparente all'utente. Quindi le i dati, utili per decidere gli assegnamenti dei processi, devono essere a disposizione del sistema operativo.
Del nuovo processo non si possono avere molte informazioni; non si conosce, perciò, quanto esso inciderà

sulle attività future del sistema, e non è possibile stabilire con esattezza se il suo trasferimento migliorerà o peggiorerà la distribuzione del carico.
Una soluzione a questo problema è fornita dalla seconda classe di algoritmi; essi permettono, infatti, il trasferimento di un processo anche in una fase successiva a quella della sua creazione. Si definiscono algoritmi di dynamic load balancing a trasferimento, e risultano i più affidabili, almeno da un punto di vista teorico, in quanto permettono una notevole dinamicità nelle decisioni. Anche per questo tipo di load balancing, come per il precedente, occorrono informazioni globali sullo stato del sistema per ciascun processore; inoltre il trasferimento di un processo già in esecuzione comporta un maggior overhead rispetto agli schemi precedenti.
Per questo tipo di algoritmo la letteratura distingue tre principali meccanismi di attivazione [3]:

1) *sender initiated*: è la CPU con carico massimo che decide ed esegue il trasferimento di un proprio processo.
2) *receiver initiated*: è la CPU con poco carico che segnala il suo stato di parziale o totale inattività alle altre, e si dispone in attesa di processi.
3) *centralized controller*: una CPU prestabilita, detta anche CPU Master, esegue periodicamente dei controlli sullo stato globale del sistema e si occupa degli eventuali trasferimenti.

Nel primo caso tutte le operazioni dell'algoritmo sono demandate alla CPU che segnala un sovraccarico. Ciò comporta un aumento dell'overhead della CPU che è già in difficoltà.
La seconda soluzione prevede invece l'utilizzo della CPU poco carica. Questa situazione viene rilevata, generalmente, quando il valore del carico è al di sotto di una determinata soglia. Una volta segnalata la sua disponibilità, sono le altre CPU a dover verificare la propria situazione e, eventualmente, a decidere del trasferimento. In questo modo, in realtà, la CPU poco attiva è parzialmente utilizzata e la maggior parte delle operazioni sono eseguite dalle altre.
Queste due politiche richiedono, per ciascuna CPU, una fase di analisi sullo stato del proprio carico, prima di eseguire l'algoritmo vero e proprio [4].
La terza soluzione non richiede la conoscenza di informazioni preliminari, come nei due casi precedenti. La CPU è identificata a priori ed esegue tutte le operazioni di decisione e di trasferimento. Si può quindi verificare il caso migliore, in cui la CPU Master è quella a carico minimo e l'esecuzione dell'algoritmo non influisce in modo rilevante sulle prestazioni. Può però verificarsi anche il caso peggiore, in cui la CPU coincide con quella a carico massimo, con gli svantaggi analoghi a quelli accennati nel primo meccanismo.

La scelta della politica da adottare è quindi legata sia all'architettura degli elaboratori, sia alla complessità delle operazioni che essa richiede.
Infatti, se è importante realizzare un meccanismo affidabile e preciso, occorre evitare che la sua esecuzione incida pesantemente sull'overhead del sistema. Un algoritmo teoricamente ottimale può in pratica causare un overhead superiore ai vantaggi effettivi che esso fornisce.

Caratterizzazione del carico.

Gli algoritmi di load balancing si basano sulla conoscenza del carico di lavoro associato a ciascuna CPU. Per "carico" di lavoro, intendiamo l'insieme di tutte le attività svolte da ogni singola CPU, intesa come sottosistema. Infatti, ciascuna CPU è completamente simmetrica e modulare rispetto alle altre, ha una propria memoria locale, gestisce un proprio insieme di processi e svolge le proprie attività di I/O.
La determinazione del carico deve rilevare le informazioni necessarie a descrivere con la dovuta esattezza le attività di ogni modulo, sia quantitativamente che qualitativamente. Queste informazioni non solo devono fornire una descrizione della situazione attuale, ma devono permettere previsioni sul prossimo futuro, con un buon grado di approssimazione.
Un errore di valutazione che causi il trasferimento di un processo comporta un aumento dell'overhead di sistema, senza apportare alcun miglioramento del carico, ma addirittura rischiando di comprometterne l'equilibrio.
Ci sono alcune attività che il sistema esegue periodicamente e che quindi non influiscono sensibilmente sulla caratterizzazione del carico. Tra le attività "saltuarie", per le quali non è possibile prevedere se e quando saranno nuovamente eseguite, alcune incideranno sensibilmente sull'overhead del processore, mentre altre richiederanno solo una quantità minima delle risorse del sistema.

Nel descrivere il carico di una CPU, occorre quindi disporre di appropriati *indici* che caratterizzino le attività attuali del sistema. Indici particolarmente adatti sono quelli che riescono a fornire una relazione monotona con il tempo di risposta del processso [5].
Il grado di precisione richiesto dipende in gran parte dal tipo di politica adottata. Se si tratta di load balancing dinamico a trasferimento, il grado di precisione è maggiore di quello necessario per il load balancing a piazzamento iniziale, perché trasferire un processo già in esecuzione è un'operazione piuttosto complessa e onerosa.

Trasferimento del processo.

I criteri per decidere del trasferimento di un processo dipendono dalla politica seguita. In generale un processo viene trasferito quando lo squilibrio tra i carichi del sistema supera una soglia prefissata. Torneremo su questi concetti quando presenteremo la parte implementativa dell'algoritmo.
Anche la scelta del processo da trasferire è strettamente dipendente dalla politica adottata.
Nel caso di algoritmi a piazzamento iniziale, viene trasferito il processo che si trova in fase di creazione ("exec"), mentre per gli algoritmi a trasferimento la situazione è più complessa. Infatti, se è vero che un processo può essere trasferito dopo la fase di creazione, è anche vero che non lo si può trasferire in un istante qualsiasi. La scelta del momento più adatto dipende dagli stati in cui può trovarsi un processo e dalla situazione complessiva del carico.
Si possono verificare situazioni in cui il trasferimento di un processo risulta inutile o addirittura dannoso. Questa fase è quindi molto delicata e la scelta del processo, così come il meccanismo di trasferimento, richiedono un'analisi approfondita delle strutture connesse alla gestione di un processo e alla gestione della memoria.

3. SOLUZIONI ADOTTATE

Dynamic Load Balancing.

L'algoritmo descritto in questo articolo appartiene alla categoria degli algoritmi dinamici; si basa quindi sui dati riguardanti lo stato attuale del sistema, rilevati a run-time da contatori inseriti in vari punti del codice del kernel di UNIX.
La politica adottata prevede il trasferimento del processo sia durante la fase di creazione, sia durante la sua esecuzione, seguendo quindi i criteri della politica a piazzamento iniziale e della politica a trasferimento. Il bilanciamento del carico può avvenire, seguendo la prima politica distribuendo i processi appena creati sulle varie CPU. Nel caso si verificassero "gravi" squilibri dopo la loro creazione, vengono trasferiti uno o più processi dalla CPU che risulta più carica a quella meno attiva, utilizzando perciò, l'algoritmo di load balancing a trasferimento.
In questo modo il bilanciamento avviene di norma attraverso l'algoritmo di load balancing a piazzamento iniziale il cui overhead risulta minore di quello a trasferimento. Quest'ultimo viene utilizzato solo nelle situazioni più critiche, quando non sono risolvibili dall'altro. In generale, con il termine "algoritmo", intenderemo l'insieme delle due politiche, gestite seguendo il criterio sopra descritto.

Definizione degli indici.

Finora si è parlato genericamente del carico di una CPU senza specificare come si ottengano le informazioni necessarie per determinarlo.
Una caratterizzazione affidabile richiede, come detto nel paragrafo precedente, l'analisi dei parametri più importanti in modo da fornire un'immagine sufficientemente esatta dello stato della CPU, senza, d'altro canto, risultare ridondante. La ricerca degli indici di carico utilizzati nell'algoritmo si è basata sia sullo studio delle principali attività del sistema, sia su risultati proposti da altri autori, rintracciabili in letteratura [6]. La ricerca svolta nell'ambito della realizzazione dell'algoritmo, ha rilevato i seguenti elementi significativi.
Il primo tra gli indici individuati, analogamente a quanto sostengono Ferrari e Zhou in [6], è quello che misura l'attività della CPU per le operazioni di calcolo interno. Definisce l'utilizzo effettivo della risorsa processore, uno dei principali colli di bottiglia del sistema. Se l'utilizzo risultasse elevato, l'invio di altri processi potrebbe far diminuire le prestazioni del sistema. Inoltre fornisce informazioni sul tipo di carico presente sulla CPU. In base alla quantità di calcolo interno e delle operazioni di I/O richieste, i processi si dividono in due categorie: I/O-bound e CPU-bound. Una CPU con

carico I/O-bound per la maggior parte del tempo è inattiva; se, invece, i processi sono in gran parte CPU-bound, la percentuale di attività sarà elevata e richiederanno un maggiore utilizzo della risorsa.
Il parametro usato per questa misurazione non può essere istantaneo: in questo caso, la sua funzione sarebbe quella di una variabile booleana, che attesta se in un dato istante la CPU è attiva o meno. Potrebbe verificarsi la situazione in cui il processore è inattivo (o attivo) al momento del rilevamento del valore, mentre si comporta in maniera diversa o addirittura opposta se considerato in un intervallo di tempo.
Per la ricerca degli indici di carico, sono neccessari quindi metodi di previsione sul futuro utilizzo delle risorse del sistema. Dato che non è possibile prevedere con precisione l'evolversi della situazione, i metodi esistenti si basano su analisi statistiche. In altre parole, in molte situazioni, occorre l'applicazione del concetto di "storia" degli eventi. Uno dei criteri più diffusi, e verificati sperimentalmente, afferma che se un evento è accaduto in un tempo relativamente recente, con molta probabilità si ripeterà anche in un prossimo futuro (un'applicazione di questo criterio è utilizzata nel concetto dei working set per la gestione a pagine della memoria)[7].
Se una risorsa è stata utilizzata in una determinata percentuale per un certo intervallo di tempo, manterrà con molta probabilità questi valori anche in tempi successivi. Quello che resta da stabilire è l'ampiezza dell'intervallo in cui effettuare i rilevamenti. Un intervallo troppo ampio può far perdere di consistenza la situazione attuale, considerando eventi ormai lontani nel tempo. Un intervallo troppo breve rende instabili i valori ottenuti falsando le reali condizioni del sistema.
L'intervallo, definito attraverso prove sperimentali, per rilevare l'utilizzo della CPU, ha l'ampiezza di circa un secondo. Le condizioni riguardanti l'uso di CPU sono verificate ogni clock tick cioè ogni 20 millisecondi[1]. L'attività di una CPU è normalmente piuttosto instabile e un intervallo di tempo troppo breve può non essere molto significativo. Estendere l'intervallo a valori molto superiori al secondo potrebbe far considerare attività lontane nel tempo, e eventualmente già concluse, come ancora influenzanti il carico della CPU.
Un secondo indice risultato importante ai fini della realizzazione dell'algoritmo di load balancing, considera la lunghezza della coda dei processi in attesa di essere eseguiti detti anche ready_to_run Il suo valore è istantaneo, poiché contiene già in sé un carattere di previsione. Infatti la lunghezza della coda stabilisce tra quanti processi dovrà essere distribuita la risorsa CPU; di conseguenza si può avere una previsione approssimativa dell'attività di questa. In realtà, pur sapendo quanti processi utilizzeranno la CPU nel prossimo futuro, non è comunque possibile fornire, con questo metodo una stima qualitativa. Non è dato sapere, ad esempio, quanto spazio occupi il processo o quanto tempo risiederà nel sistema o che tipo di attività privilegerà. Se, ad esempio, si contassero numerosi processi sulla coda di una CPU, si potrebbe dedurre un carico piuttosto intenso per questo processore nel prossimo futuro. Se, però, tutti questi processi richiedessero un utilizzo minimo della CPU, per poi terminare, il carico in realtà non sarebbe molto elevato. Viceversa, un solo processo in coda che utilizza tutto il suo time-slice in calcolo interno e resta nel sistema per un periodo prolungato, potrebbe risultare oneroso per il carico della CPU. In generale, comunque, questi due esempi sono situazioni limite; di norma la lunghezza della coda dei processi ready_to_run fornisce una previsone, seppure approssimata, sul carico futuro della CPU.
Un terzo indice rileva l'attività di trasferimento di pagine dalla memoria all'area di swap. L'attività di swapping è una delle più onerose del sistema[8]. Il trasferimento di una pagina richiede un' operazione di I/O, che rallenta notevolmente le altre attività del sistema. Si ricorre a questa soluzione solo quando si rileva un congestionamento della memoria. Un valore elevato dell'indice di swap fornisce quindi informazioni sulla quantità di memoria occupata e sull'overhead imposto alla CPU per gestire questa situazione. L'indice utilizzato per rilevare questa attività non deve essere istantaneo. Se così fosse, indicherebbe semplicemente il trasferimento di una pagina in un dato istante, ma non rileverebbe nulla circa i trasferimenti avvenuti anche solo pochi istanti prima; in ogni caso non si potrebbe stabilire se si tratta di un'attività sporadica o di una fase particolarmente critica della vita del sistema. Un'operazione di swap ha la durata di circa 25 millisecondi. Per avere un valore indicativo occorre quindi osservare un intervallo di tempo piuttosto lungo, che nell'algoritmo è stato fissato a quattro secondi.
Le informazioni sull'occupazione di memoria fornite

[1] incrementando un apposito contatore, nel caso in cui la CPU risulti attiva.

da questo indice non permettono però nessuna possibilità di prevenire situazioni critiche. Si è quindi considerato un quarto indice che permetta di rilevare la quantità di memoria disponibile in modo da attivare un ribilanciamento di processi prima di dover intervenire trasferendo sull'area di swap. Tale indice fornisce la quantità di memoria disponibile in pagine ed è utilizzato dal sistema anche per altri scopi. Questo valore ha maggiore rilevanza se la memoria disponibile raggiunge valori minimi, mentre non è molto significativo se i valori sono elevati.
Un altro fattore che incide sul carico della CPU riguarda la sua attività di trasferimento di processi. Se la CPU sta svolgendo tale attività, il suo carico verrà alterato, talvolta anche sensibilmente; occorre quindi evitare trasferimenti contemporanei su quella CPU. Un indice che rilevi tali attività risulta perciò molto utile.
Un altro elemento considerato nell'algoritmo è il costo di trasferimento.
Se si tratta dell'algoritmo a piazzamento iniziale le operazioni svolte dalla CPU per ricevere un processo non sono molto costose (Fig.1).

La migrazione di un processo nell'algoritmo a trasferimento è più complessa. Sorgono problemi di sincronizzazione; inoltre per mantenere la coerenza dei dati durante le fasi dell'operazione, occorre gestire un meccanismo di comunicazione tra i processori che permetta lo scambio di dati e di informazioni necessari al trasferimento.
Il passo successivo alla ricerca degli indici riguarda la scelta dei pesi da associare ad essi.
Infatti i valori ricavati dagli indici di carico non sempre coincidono con l'importanza che essi rivestono all'interno dell'algoritmo. Analizzando gli indici appena discussi, si possono evidenziare degli ordini di priorità.
L'elemento più rilevante sul carico del sistema è quello associato all'attività di swapping. Si è visto come questa attività risulti tra le più onerose del sistema sia per il costo che comportano le operazioni di I/O ad essa associate, sia per le condizioni di sovraccarico in cui viene a trovarsi la memoria della CPU che svolge questa attività. Il peso associato a questo valore deve essere quindi tra i più elevati.
Il secondo elemento in ordine di importanza è il valore

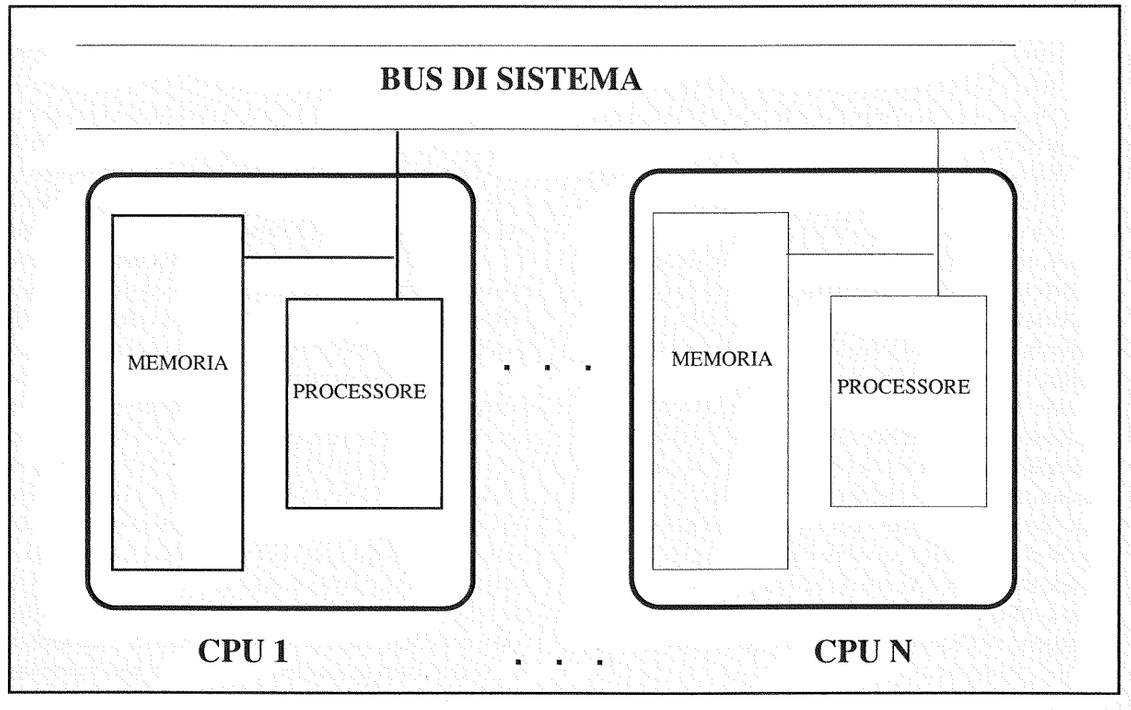


Fig. 1 Architettura del sistema multiprocessor

associato alla lunghezza della coda che offre un buon livello di previsione sul carico futuro della CPU, pur non fornendo informazioni sulla qualità dei processi. Se, ad esempio, si suppone un tipo di carico che sfrutta equamente tutte le risorse offerte dal sistema, il numero dei processi presenti in coda è direttamente proporzionale all'utilizzo delle risorse.
Segue nella lista delle priorità l'indice associato all'uso di CPU. I valori all'interno del range accettabile possono variare considerevolmente, anche se la CPU rappresenta sempre un collo di bottiglia per le prestazioni del sistema.
Il peso relativo alla memoria libera varia in base al numero di pagine disponibili. Se il sistema non è sovraccarico e le memorie di tutte le CPU sono solo parzialmente occupate, questo valore non incide molto sull'attività complessiva. Viceversa, se le condizioni globali sono già critiche e la quantità di memoria disponibile è bassa, ma ancora sopra i livelli di soglia che causerebbero il trasferimento dei processi sull'area di swap, il peso associato all'indice dovrebbe avere un valore superiore rispetto al caso precedente.
L'indice riguardante il costo del trasferimento è stato considerato come un valore costante , per motivi di semplicità, anche se ciò non preclude eventuali affinamenti in versioni future.
Le stesse considerazioni valgono per l'indice riguardante l'attività di trasferimento, che ha un comportamento simile a quello di una variabile semaforica.
La parte dell'algoritmo che riguarda la decisione di trasferimento di un processo considera il carico di una CPU come un singolo valore intero, per permettere un facile confronto. Questo valore è ottenuto come una sintesi dei fattori che contribuiscono a caratterizzare il carico, evidenziati in precedenza. E' calcolato attraverso funzioni lineari che operano su valori interi.
La determinazione del carico massimo e del carico minimo è quindi ottenuta confrontando direttamente i valori di carico. Ad ognuno di essi è associato l'identificatore della CPU e si può stabilire quale sia il processore più attivo e quella meno attivo. In base ai risultati ottenuti decide del trasferimento del processo.

Scelta del processo.

Il processo da trasferire deve essere scelto in modo che il suo trasferimento produca evidenti miglioramenti sul carico del processore su cui risiede. Sono stati analizzati i vari stati in cui può trovarsi un processo e siamo giunti alla conclusione che l'unico in cui ha significato eseguire un trasferimento è quello ready_to_run. Infatti, in questo stato il processo ha tutte le carte in regola per essere eseguito: è solo in attesa di ricevere la risorsa CPU, e il trasferimento in questa fase comporterebbe una probabile riduzione dei tempi di questa attesa.
Tra i processi candidati al trasferimento si possono applicare altri criteri decisionali, utilizzando eventualmente dei pesi così come è stato fatto per la caratterizzazione del carico. L'algoritmo realizzato si limita a considerare il precedente criterio come l'unico per la determinazione del processo da trasferire, anche se non si escludono futuri raffinamenti, orientati principalmente all'analisi del tipo di carico che il processo induce sul sistema.

4. ASPETTI IMPLEMENTATIVI

L'algoritmo di load balancing è stato implementato seguendo, in parte, due percorsi distinti.
E' stato infatti necessario dividere, a livello implementativo, l'algoritmo a piazzamento iniziale da quello a trasferimento. Le differenze fondamentali si rilevano nel meccanismo di attivazione dell'algoritmo.
Nel caso del piazzamento iniziale, la verifica del bilanciamento del carico è effettuata ogni volta che viene creato un processo. In questo modo, se si evidenzia uno squilibrio, il processo appena creato viene trasferito sulla CPU a carico minimo.
Per l'algoritmo a trasferimento, come è già stato accennato all'inizio, esistono tre meccanismi di attivazione distinti: sender initiated, receiver initiated e centralized controller. Nel nostro caso, l'algoritmo realizzato prevede due procedure di attivazione distinte, basate sui meccanismi di receiver initiated e di centralized controller.
Nella prima procedura il valore di soglia utilizzato coincide con lo stato di CPU_IDLE, cioè con la completa inattività. A differenza di quanto detto per la politica receiver initiated, però, la CPU non si limita a segnalare il suo stato alle altre, ma esegue sia la fase di decisione sia la fase di trasferimento dell'algoritmo. Precisamente, essa valuta i carichi delle altre CPU, determina quella a carico massimo, sceglie un processo appartenente a quest'ultima e ne chiede il trasferimento.
Questa politica evita quei casi in cui una CPU è inattiva mentre le altre sono sovraccariche; si tratta, comun-

```
struct load
{
   - lb_cpuse[]        /* uso di CPU                  */
   - lb_swapuse[]      /* attività di swappping       */
   - lb_rqlen          /* lunghezza run queue         */
   - lb_trans          /* trasferimenti in corso      */

}
```

Fig. 2 Struttura per gli indici di carico

que, di situazioni limite, dalle conseguenze spiacevoli, che si verificano non molto di frequente, anche se non sono rare.
Può invece accadere che il carico sia fortemente sbilanciato pur non essendoci nessuna CPU inattiva; in questo caso è possibile intervenire utilizzando la politica del centralized controller.
La seconda procedura prevede, infatti, l'attivazione dell'algoritmo ad intervalli regolari, da parte di una CPU Master. Questa CPU esegue tutte le fasi dell'algoritmo solo se, durante l'analisi dei carichi, risulta essere la meno attiva.
In tutti gli altri casi si limita ad avvisare la CPU con carico minimo e ad abbandonare la gestione dell'algoritmo a quest'ultima.
L'interazione tra le due politiche evita forti squilibri del carico e, soprattutto, permette di utilizzare, sempre e comunque, tutte le CPU presenti nel sistema.
La parte dell'algoritmo che si occupa della caratterizzazione del carico utilizza due strutture principali contenenti, rispettivamente, le informazioni sugli indici di carico e i valori finali di carico delle varie CPU. Le due strutture sono riportate in Figura 2 e Figura 3. Mentre la seconda struttura contiene dati generali riguardanti il carico delle CPU, la prima , per l'importanza delle informazioni che contiene, merita spiegazioni più dettagliate.
In particolare, le descrizioni riguarderanno gli elementi della struttura che si riferiscono all'uso di CPU (*lb_cpuse*) e all'attività di swapping (*lb_swapuse*) ai quali è stato applicato il concetto di "storia", utilizzando un array.
Per l'indice *lb_cpuse* l'intervallo di tempo è suddiviso in sottointervalli dalla durata di 16 clock tick. Ad ogni interrupt di clock, se la CPU risulta attiva, viene incrementata una variabile di sistema. Ogni 16 interrupt il valore contenuto in questa variabile viene assegnato ad una delle entry della variabile *lb_cpuse* e l'indice dell'arrray viene incrementato in modo tale da poter scandire il vettore circolarmente. L'intervallo di tempo co-

```
struct tot
{

   - tl_sem        /*  variabile semaforica        */
   - tl_tot[]      /*  carico di ciascuna CPU      */
   - tl_min        /*  valore di carico minimo     */
   - tl_max        /*  valore di carico massimo    */
   - tl_cpumin     /*  CPU con carico minimo       */
   - tl_cpumax     /*  CPU con carico massimo      */


}
```

Fig. 3 Struttura per i valori di carico

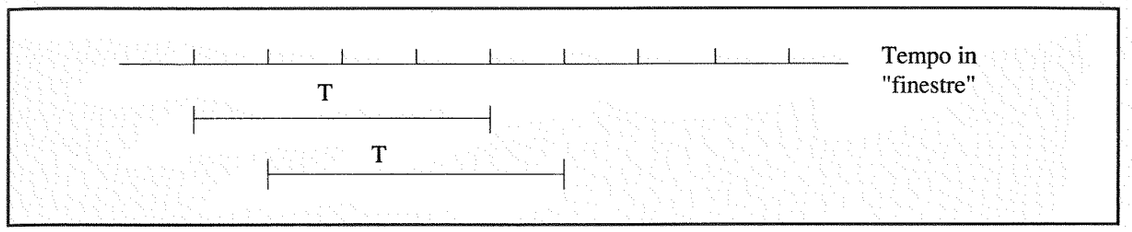


Fig. 4 Slittamento dell'intervallo di tempo

perto globalmente da *lb_cpuse*, come già visto, è fissato ad poco più di un secondo; l'array è formato perciò da quattro elementi ciascuno dei quali descrive una "finestra" di circa un quarto di secondo. In Figura 4 è visualizzato l'intervallo globale (*T*) e l'aggiornamento che avviene ogni 16 clock tickt.

Un analogo criterio è stato utilizzato per il calcolo di lb_swapuse. L'intervallo di tempo in questo caso è fissato in quattro secondi; i sottointervalli hanno invece le stesse dimensionidi *Cb-cpuse*, per non aumentare troppo la granularità degli aggiornamenti; *lb_swapuse* è incrementata ogni volta che è richiesta un'attività di swap. In questo caso l'array sarà composto da 16 "finestre", poiché l'intervallo di tempo è quadruplicato.
La struttura contiene, fra l'altro, informazioni riguardanti il numero dei processi nella run queue (*rqlen*), la segnalazione del trasferimento in corso (*trans*), la quantità della memoria disponibile (*freemem*).
Il calcolo del carico segue criteri analoghi per entrambe le versioni dell'algoritmo, sia a piazzamento iniziale sia a trasferimento.

Come si vede in Figura 5, la CPU che esegue l'algorit-

```
Definizione del carico


/* Medium è la media matematica delle memorie libere
   delle vari CPU */

/* EMPTY e FULL  sono valori costanti */

if (Medium < S)
        memcoeff = EMPTY
else
         memcoeff = FULL

for ( i < N )
{
      while (n < index_cpu)
       {
            cpu = cpu + lb_cpuse[n]
       }
      while (m < index_swap)
       {
          swap = swap + lb_swapuse[m]
       }

      diffmem = Medium - Mem[i]

      if (diff > K     )
           memfactor = diff * memcoeff

      LOAD[i] = cpu*K     + swap*K     + runqueue*K     +


}
```

Fig. 5 Calcolo del carico delle CPU

mo di load balancing calcola prima il valore medio delle pagine di memoria disponibili. Se si trova al disotto di una determinata soglia "*s*" viene aumentato il fattore moltiplicativo associato all'indice della memoria libera (*memcoeff*). I valori possibili sono predefiniti, associati a due costanti *EMPTY* e *FULL*, eventualmente riconfigurabili.
Per ogni CPU, vengono calcolati i valori di *cpu* e *swap* nell'utlimo intervallo di tempo e si sommano, nell'elemento corrispondente dell'array *LOAD* , i vari indici moltiplicati per i loro rispettivi pesi. Notiamo che ogni elemento di *LOAD* contiene il valore rappresentante il carico di una CPU.

L'algoritmo di decisione, riportato in Figura 6, stabilisce l'effettiva necessità del trasferimento di un processo: controlla i carichi delle altre CPU e sottrae un valore che rappresenta il fattore di trasferimento, determina la CPU con il carico più elevato e se la differenza tra la CPU a carico minimo e quella a carico massimo supera un fattore, che rappresenta l'overhead dovuto alle operazioni di migrazione, effettua il trasferimento.
L'algoritmo di decisione del load balanicing a piazzamento iniziale ricerca invece la CPU a carico minimo senza aggiungere ulteriori controlli, come mostrato in Figura 7. Infine, aggiungiamo che per quanto concerne la scelta del processo per il trasferimento, le operazioni principali consistono nella selezione di un processo in stato di ready_to_run dalla tabella globale dei processi.

La comunicazione tra i processori, sia per verificare i rispettivi carichi di lavoro, sia per permettere lo scambio di informazioni attinenti il processo da trasferire, avviene attraverso meccanismi di semaforizzazione, analoghi a quelli descritti da Dijkstra [9].

5. CONCLUSIONI.

L'analisi delle prestazioni è stata effettuata eseguendo dei workload campione mediante il benchmark *sequent*

Decisione del trasferimento

```
max = 0;

for  (j < N)
{
        /* myself è la CPU a carico minimo che esegue
           l'algoritmo */

        if (j = myself)
                continue;
        else
                if (LOAD[j] > max)
                 {
                        max = LOAD[j];
                        CPUMAX = j;
                 }
}

        if (max - LOAD[myself] > overhed)
        /* trasferisci da CPUMAX a myself */
```

Fig . 6 Decisione per l'algoritmo a trasferimento

su macchine in cui era presente l'algoritmo. Il test simula un ambiente multiutente, in cui vengono eseguiti più processi, qualitativamente e quantitativamente diversi tra loro per durata, tipo e dimensioni. I risultati considerano i tempi di risposta per ciascun workload. I valori ottenuti sono stati confrontati con quelli risultati attivando lo stesso carico, su una macchina dello stesso tipo, su cui era presente un altro algoritmo di load balancing che prevedeva solo il piazzamento iniziale. Si sono rilevate delle diminuzioni nei tempi di risposta che vanno dal 20% al 40%; in alcune situazioni, con workload dello stesso tipo, il carico veniva distribuito in maniera completamente equilibrata, con un throughput complessivo vicino alla somma dei throughput delle singole CPU. Si sono anche effettuati dei rilevamenti sul tipo di attività svolta dal sistema utilizzando appositi strumenti di monitoraggio [10]. Da questi esami si evidenzia una distribuzione equilibrata dei processi tra le varie CPU, permettendo così un miglior utilizzo delle risorse del sistema.

6. BIBLIOGRAFIA

[1] Livny M., Melman M., LOAD BALANCING IN HOMOGENEUS DISTRIBUTED SYSTEMS, Proc. ACM Computer Network Performance Symposium, Aprile 1982, pp. 55-57.

[2] Eager, D. L., E. D. Lazowska, J. Zahorjan, ADAPTIVE LOAD SHARING IN HOMOGENEUS DISTRIBUTED SYSTEMS IEEE Trans. Soft. Eng., Vol. SE-12, No. 5, Maggio 1986, pp. 662-675.

[3] Ferrari D., A STUDY OF LOAD INDICES FOR LOAD BALANCING SCHEMES, UCB/CSD Report N. 85/262, Computer Science Division, University of California, Berkeley, Ottobre 1985.

[4] Eager, D. L., E. D. Lazowska, J. Zahorjan, *A* COMPARISON OF RECEIVER-INITIATED AND SOURCE-INITIATED DYNAMIC LOAD SHARING, Proc. 1985 ACM SIGMETRICS Conference on measurement and modeling of computer systems"

[5] Zhou S., AN EXPERIMENTAL ASSESSMENT OF RESOURCE QUEUE LENGTHS AS LOAD INDICES, Proc. Winter USENIX Conference, Gennaio 1987, pp. 73-82

[6] Ferrari D., S. Zhou, AN EMPIRICAL INVESTIGATION OF LOAD INDICES FOR LOAD BALANCING APPLICATIONS, Proc. Performance '87, pp. 515-528

[7] Le Van Huu, INTRODUZIONE ALLA STRUTTURA INTERNA DI UNIX, Edizioni Unicopli, Milano, 1988.

[8] Maurice J. Bach, THE DESIGN OF THE UNIX OPERATING SYSTEM, Prentice-Hall, New Jersey, 1986.

[9] James L. Peterson, Abraham Silberschatz, OPERATING SYSTEM CONCEPTS, 2° edizione, Addison-Wesley, Reading Mass., 1986.

[10] Paola Zocco Ramazzo, UNO STRUMENTO DI RILEVAZIONE DATI SULL'UTILIZZO DI RISORSE ORIENTATO AL TUNING DI UN SISTEMA UNIX IN UN AMBIENTE MULTIPROCESSOR, Tesi di laurea in Scienze dell'Informazione, Univ. degli Studi di Milano, 1987.

IL SISTEMA OPERATIVO MACH

G.D. Giuseppe Facchetti
IBM Italia - Segrate

RIASSUNTO

Mach è un kernel di sistema operativo sviluppato presso il Dipartimento di Computer Science della Carnegie-Mellon University di Pittsburgh.
Esso si presenta come una base per la costruzione di sistemi operativi per architetture multiprocessore ed ambienti distribuiti. Questo articolo presenta le caratteristiche principali di Mach, ed indica le ragioni della popolarità di questo sistema, concludendo con alcune considerazioni sulle sue applicazioni future.

ABSTRACT

Mach is an operating system kernel developed at the Computer Science Department of the Carnegie-Mellon University. It provides a new foundation for the development of operating systems in distributed and multiprocessor environments. This paper introduces the main characteristics of Mach, and explains the reasons of its popularity.
A few notes about current and future applications of Mach close the article.

1. UNIX PER ARCHITETTURE MULTIPROCESSORE

Molti aspetti del sistema operativo Unix risentono fortemente dell'ambiente per il quale esso è nato, della macchina su cui è stato originariamente costruito, e degli scopi per i quali i suoi ideatori intendevano utilizzarlo. Di tali aspetti, quello che trattiamo nel presente articolo è la possibilità per Unix di gestire una macchina basata su una architettura a più processori. Il PDP7 ed il PDP11 della Digital, le prime macchine sulle quali il sistema fu implementato, avevano una architettura molto semplice (se considerate con gli occhi di oggi), ed erano macchine monoprocessore.
Unix fu in seguito portato su molte macchine diverse, e questo ne facilitò una notevole evoluzione (non sempre migliorativa, per la verità) e ne favorì la capacità di gestire le architetture hardware piu' diverse. Solo negli ultimi anni, però, è stato affrontato il problema del porting su macchine multiprocessore, e da questo punto di vista Unix ha presentato forti problemi (come, del resto, qualunque altro sistema operativo nato per macchine monoprocessore).

La difficoltà principale consiste nella capacità del sistema di sfruttare il parallelismo intrinseco dell'architettura multiprocessore, eseguendo il codice del kernel su più processori contemporaneamente: in questa situazione, i normali meccanismi di protezione delle strutture dati del kernel (non prelazionabilità del kernel ed inibizione delle interruzioni) non sono più sufficienti. Infatti due processori potrebbero comunque modificare contemporaneamente le stesse strutture dati, con conseguenze disastrose [1]. Alcuni costruttori hanno

deciso di aggirare la difficoltà adottando architetture *Master-Slave*, nelle quali un solo processore (detto *master*) può eseguire il codice del kernel. Questa non è una soluzione valida in generale, infatti per macchine con più di due o tre processori il master diventa il collo di bottiglia del sistema.

Diversi costruttori si sono impegnati nella risoluzione del problema, seguendo due tipi di approccio. Alcuni sono partiti dal kernel di Unix e lo hanno modificato più o meno estensivamente per riuscire a parallelizzarne le operazioni, inserendo meccanismi di mutua esclusione. E' questo il caso di Sequent, Silicon Graphics ed Olivetti.
Altri hanno invece ricostruito da zero il kernel, modellandone la struttura attorno all'architettura multiprocessore cui era destinato, ottenendo così un sistema meglio strutturato, a fronte però di un maggiore sforzo progettuale ed implementativo (è il caso di Data General).

In questa arena di gruppi di progetto software si è fatta luce in questi ultimi anni (ma è quasi il caso di dire mesi) la Carnegie Mellon University di Pittsburgh, dove il Dipartimento di Computer Science ha prodotto, a seguito di una attività pluriennale di ricerca nel campo, il sistema operativo Mach. Mach è un sistema costruito seguendo il secondo tipo di approccio sopra descritto, cioè è stato ottenuto da una riscrittura completa del kernel. E' quindi basato su astrazioni e modelli (orientati soprattutto all'architettura multiprocessore) al di sopra dei quali è stata realizzata la tradizionale interfaccia di sistema di Unix.

I prossimi paragrafi descrivono le caratteristiche di Mach, e la sua diffusione come base di sviluppo per altri sistemi operativi.

2. CONCETTI BASE DI MACH

Prima di entrare nella descrizione dettagliata di Mach, e' importante osservarne una caratteristica fondamentale: Mach è basato su una struttura *Object-Oriented*. Il sistema è visto come un insieme di oggetti, suddivisi in classi, ognuno dei quali ha delle specifiche caratteristiche e funzionalità, ed ognuno dei quali scambia informazioni e messaggi con gli altri oggetti, allo scopo comune di gestire la macchina sottostante [1].

Questa visione è risultata particolarmente utile per costruire Mach come un sistema modulare ed adattabile alle architetture piu' diverse.

2.1 Le Astrazioni Definite in Mach.

Mach è nato come evoluzione di due precedenti sistemi operativi, RIG ed Accent, pure costruiti alla Carnegie Mellon University [4].
Essi già incorporavano alcuni dei concetti che caratterizzano il sistema Mach. Mach si basa su cinque astrazioni fondamentali: *task*, *thread*, *port*, *message* e *memory object*. Su di esse sono basate la gestione della esecuzione dei programmi, la gestione della memoria virtuale, la comunicazione tra processi e la gestione della memoria secondaria.

Un *task* è un "recipiente" che accoglie un certo numero di risorse ad esso allocate. Esso include uno spazio di indirizzamento, e diritti di accesso a risorse del sistema quali processori, file, eccetera. Un *thread* è un oggetto in grado di effettuare del lavoro computazionale utilizzando le risorse del task che lo ospita; l'unico attributo di un thread è il suo "stato di esecuzione", dato per esempio da un *program counter* ed un insieme di registri. Un task può "ospitare" piu' thread, e tutti i thread ospitati da un task ne condividono lo spazio di indirizzi e le risorse disponibili. Un tradizionale processo Unix è rappresentato in Mach da un task contenente esattamente un thread.
La comunicazione tra processi in Mach è definita in termini di *port* e *messaggi*. Un *port* è un canale di comunicazione. Puo' essere considerato come una coda di messaggi protetta dal kernel. Un task può ottenere accesso ad un port ricevendo (dal proprietario del port, che può essere il kernel od un altro task creatore del port in questione) un messaggio contenente il diritto di accesso corrispondente. Un *messaggio* consiste in un descrittore ed una sequenza di oggetti (caratteri, aree di memoria, indirizzi, ...). Ogni oggetto del messaggio è descritto da un tipo, che ne definisce le caratteristiche. Come in ogni ambiente object-oriented, per ogni classe di oggetti sono definite delle operazioni che gli elemen-

[1] Spesso, anche in ambienti esperti, si fa molta confusione tra "sistema object-oriented" e "sistema SCRITTO in linguaggio object-oriented". Si osservi che si tratta di due cose ben diverse, delle quali nessuna implica l'altra. Per esempio Mach è scritto in linguaggio C.

ti della classe sono in grado di espletare. Per esempio, le operazioni definite sui port sono quelle di *invio* e *ricezione* di messaggi.
A parte queste due operazioni, e poche altre operazioni che forniscono la descrizione di task e thread, *tutte* le altre funzionalità di Mach sono espresse in termini di chiamate (di procedura remota) ai port.

2.2 Architettura di Mach

Vediamo ora come queste astrazioni permettono la strutturazione di un kernel di sistema operativo. Il kernel di Mach è di fatto un task contenente più thread: uno di essi si occupa della schedulazione dei processi, un altro della paginazione, e così via. Questi thread possono essere in esecuzione contemporaneamente su processori diversi.
Il kernel puo creare altri task e thread, ad esempio allo scopo di creare processi utente (questi a loro volta possono creare altri task ed altri thread). L'operazione di creazione di un task o thread restituisce al creatore il diritto di spedire messaggi al port che rappresenta il task o thread creato. Tale port agisce da intermediario per ogni operazione che coinvolge il task o thread stesso (figura 1). I messaggi che vengono inviati ad un port producono cioé l'esecuzione di operazioni sull'oggetto rappresentato da quel port (questo modello non e' dissimile da quello adottato in altri sistemi operativi [5, 6]. L'importanza di questo "lavoro di rappresentanza" svolto dai port sta nella immediata applicabilità di questo modello ad un ambiente distribuito, cioè ad una rete di elaboratori: gli oggetti del sistema possono essere posti ovunque nella rete senza che cambi il meccanismo di programmazione della comunicazione. Per esempio, un thread A può sospendere un thread B mandando un messaggio al port che rappresenta B. Sarà il meccanismo di comunicazione sottostante (che è lo stesso per tutti gli oggetti) a localizzare il port destinatario.
Questo modello conduce immediatamente ad una struttura di sistema di tipo *client-server*, dove cioè alcuni oggetti (detti *server*) forniscono delle funzionalità (gestione di file, gestione della rete, ...), ed altri oggetti (detti *client*) accedono a queste funzionalità spedendo messaggi ai (port dei) server fornitori delle stesse. E tutto questo indipendentemente dal fatto che client e server "girino" sullo stesso processore, oppure su diversi processori di una stessa macchina, oppure ancora su diverse macchine di una rete.

In questo senso (e su questo ritorneremo più avanti), il sistema operativo nella sua globalità ha una interfaccia che è più funzione dei server che supporta, che non del kernel sottostante.

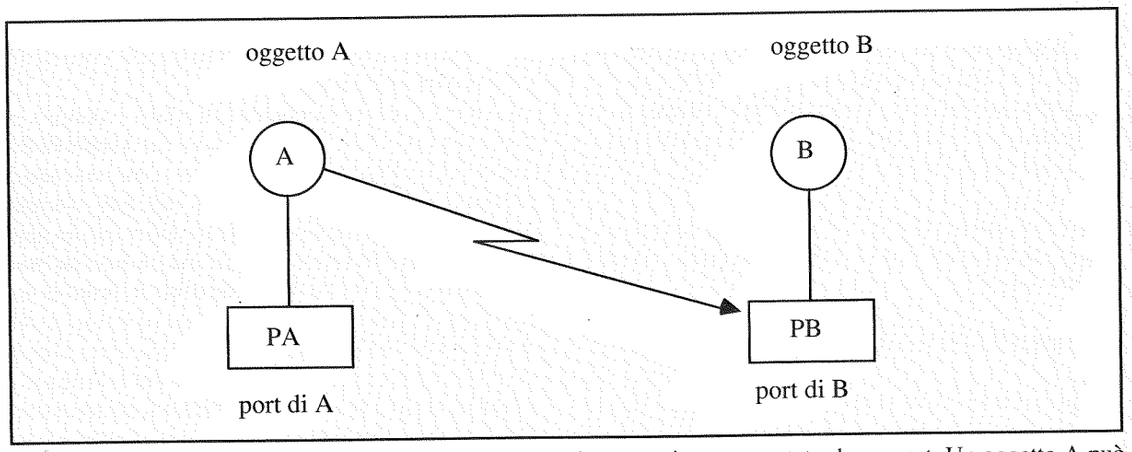


Fig. 1 Comunicazione tra Oggetti in Mach. In Mach ogni oggetto è rappresentato da un port. Un oggetto A può chiedere ad un oggetto B l'esecuzione di una operazione, inviando al port di B un messaggio. L'oggetto A deve avere diritto di accesso al port di B. Il fatto che A e B siano o meno sulla stessa macchina è del tutto trasparente all'interfaccia di comunicazione che essi utilizzano.

[2] Esistono macchine sulle quali la memoria fisica è più ampia della memoria virtuale.

2.3 Gestione della Memoria Virtuale

Uno dei problemi più complessi che i sistemi operativi attuali devono affrontare è quello della gestione della memoria virtuale. Questa è venuta assumendo una importanza sempre maggiore non più tanto e non più solo per permettere ai programmi di utilizzare uno spazio di indirizzamento più ampio della memoria fisica della macchina[2], ma anche e soprattutto per gli strumenti di protezione e comunicazione tra processi che la memoria virtuale permette di implementare. Questo paragrafo descrive i concetti e l'implementazione della memoria virtuale in Mach.

Lo spazio di indirizzamento di un task consiste in una collezione ordinata di *regioni* di memoria. Queste regioni possono essere allocate ovunque all'interno dello spazio di indirizzamento virtuale consentito dalla macchina sottostante. Mach consente a task "parenti tra loro" (cioè aventi un antenato comune) di condividere in lettura e/o scrittura delle aree di memoria. Il meccanismo di condivisione è schematizzato in figura 2, ma non entriamo nei dettagli della sua realizzazione.
Il meccanismo di *copy-on-write* viene utilizzato per migliorare l'efficienza delle operazioni di trasferimento di messaggi.
Una caratteristica fondamentale di Mach è il fatto che anche la gestione della memoria secondaria (e quindi della paginazione) segue lo stesso paradigma client-server descritto nel precedente paragrafo. Memoria centrale e memoria secondaria sono viste non come due sistemi separati, ma come un unico "fornitore" di memoria (*single-level storage*), nel quale la memoria centrale agisce solo come *cache* della secondaria.

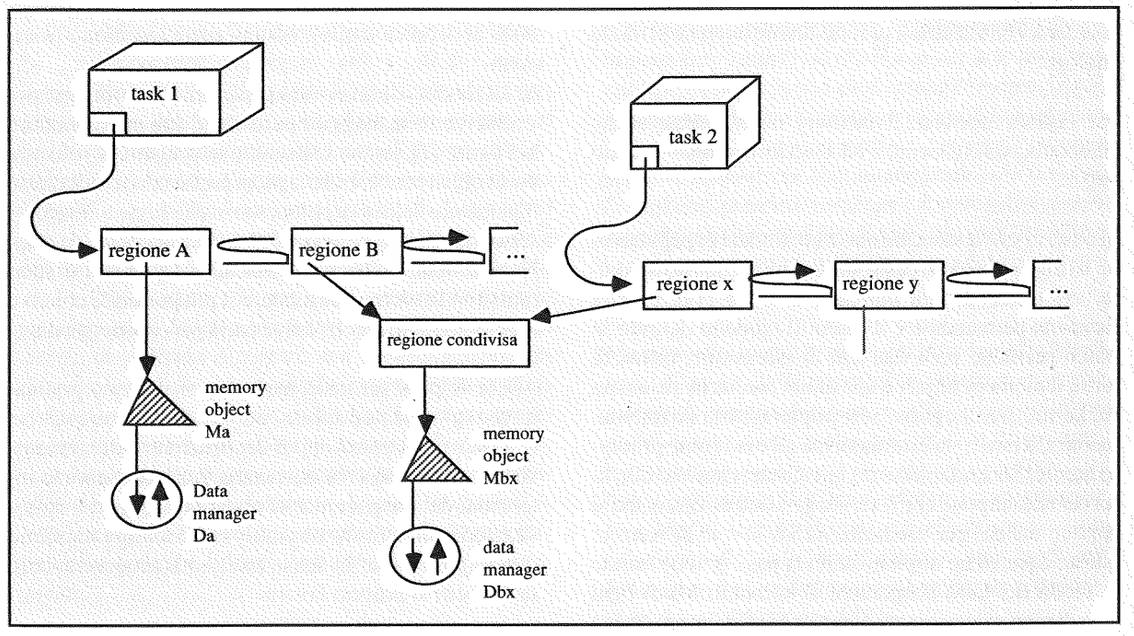


Fig. 2 Spazio di Indirizzamento di un Task in Mach. Ad ogni task è associato uno spazio di indirizzamento costituito da una lista di regioni di memoria. Ogni regione è un intervallo continuo di indirizzi validi. Ad ogni regione sono associati un memory object ed un data manager che gestiscono i trasferimenti di dati tra memoria primaria e memoria secondaria. Due task aventi un antenato comune possono condividere una area di memoria utilizzando un descrittore chiamato "sharing map", che sottende un unico memory object per le due regioni. In questo esempio, la regione B del task 1 e la regione X del task2 condividono l'area di memoria gestita dall'oggetto M.

L'oggetto di sistema che fa da intermediario e da gestore della paginazione in Mach è il *memory object* (figura 2). Un *memory object* (anch'esso ovviamente rappresentato da un port) è un oggetto astratto che rappresenta una collezione di bytes sulla quale è definito un insieme di operazioni quali *read*, *write*, eccetera. Ad un memory object viene associato un *data manager* che è responsabile della inizializzazione di questa collezione di bytes, e dell'eventuale salvataggio dei dati su memoria permanente. Quando si verifica un *page fault*, il kernel effettua una chiamata al memory object che gestisce l'area di memoria nella quale si è avuto il *fault*, chiedendo che la pagina mancante venga portata in memoria centrale. Se tutte le pagine di memoria fisica sono già occupate, è necessario rimpiazzare una pagina in memoria con la pagina richiesta. Come al solito, se la pagina da rimpiazzare è stata modificata, è necessario scaricarla su memoria secondaria: di nuovo, questo verrà ottenuto tramite una chiamata al memory object corrispondente. La deallocazione delle aree di memoria di un task che termina il proprio lavoro avviene in modo analogo.

Per ragioni storiche, il memory object viene anche chiamato *paging object*, ed il data manager è detto *pager*.

Si osservi che il meccanismo tradizionale di paginazione su disco è solo un caso particolare, di questo modello, nel quale il data manager è il *file system*, e la memoria permanente è il disco. Il modello definito in Mach permette però due tipi di estensione: prima di tutto, il memory object è una entità che viene chiamata dal kernel, ma che può essere implementata sia internamente al kernel, sia esternamente ad esso come processo utente. Secondariamente, questo meccanismo client-server non impone che il server gestisca un disco, ma si applica anche, per esempio, ad un *server di rete*: in questo caso viene implementata la paginazione su rete in modo del tutto trasparente al kernel.In Mach ogni area di memoria paginata è gestita da un memory object e da un pager [3], e questa suddivisione del lavoro tra più oggetti separati si adatta ancora una volta perfettamente ad una architettura multiprocessore. Di fatto le strutture di controllo e sincronizzazione di Mach hanno portato alla costruzione di un sistema di memoria virtuale completamente *parallelizzato*.

Con questo termina la breve descrizione del kernel di Mach, per i cui dettagli più specifici si rimanda alla bibliografia citata. I prossimi paragrafi affrontano gli altri aspetti che hanno reso Mach popolare in ambito accademico ed industriale.

3. SOFTWARE ENGINEERING

Un importante pregio di Mach è la metodologia software con la quale esso è stato costruito. Oltre ad avere definito uno standard di programmazione che massimizza la leggibilità e comprensione del codice (contrariamente a quanto spesso avviene in ambienti di programmazione C ed assembly), i progettisti di Mach hanno strutturato il codice in modo da massimizzarne la portabilità. In particolare, hanno suddiviso l'implementazione in una parte *machine-independent* (più del 90% del codice totale), ed una parte *machine-dependent*.
In tal modo, le numerose operazioni di porting già avvenute verso la maggior parte delle attuali architetture hardware [2], hanno consentito ai progettisti software coinvolti di concentrare la propria attenzione sulla parte dipendente dalla macchina, senza che fosse mai necessario modificare la corposa parte indipendente da essa. Naturalmente nella parte machine-dependent è stato massimo lo sforzo per rendere il più possibile chiaro e leggibile il codice (chi scrive parla per esperienza diretta).
Anche le parti scritte in assembly richiedono praticamente solo una traduzione "uno-a-uno", senza imporre le complesse operazioni di decifrazione e di... chiaroveggenza tipicamente necessarie per indovinare le intenzioni dei programmatori assembly. Questo a dimostrazione del fatto che assembly non è sempre sinonimo di non-portabilita (in senso lato), se il programmatore sa far bene il proprio lavoro.

In altre parole, le persone della Carnegie Mellon hanno dimostrato come ingegneria del software ed efficienza del codice possano non essere in contrapposizione l'una all'altra, semplicemente definendo una minima disciplina comune di programmazione. Recentemente nel gruppo Mach è stata anche costruita una (seppur ristretta) sintassi di programmazione object-oriented sopra il linguaggio C, allo scopo di migliorare ulteriormente la modularità del codice prodotto.

[3] Un pager è un thread all'interno del task che rappresenta il kernel, e diversi memory object possono appoggiarsi sullo stesso pager.

4. EMULAZIONE DI ALTRI SISTEMI OPERATIVI

Anche altre ragioni tecniche hanno contribuito al successo di Mach. In questi ultimi anni, la diffusione di Unix in tutti i campi applicativi ha richiesto una continua estensione delle funzionalità del suo kernel, che è così diventato terra di conquista (e di devastazione) per chiunque avesse un'idea su come "potenziare" lo Unix originario [4]

Mach in questo senso ha voluto riproporre la funzione originaria di "kernel" di sistema operativo, inteso come il fornitore di una ristretta quantità di servizi flessibili e potenti, al di sopra dei quali qualunque ambiente operativo può essere implementato. Ed i progettisti di Mach hanno ricostruito, sopra il loro sistema, l'ambiente Unix BSD4.3, migliorandone le prestazioni globali. Di fatto sono state riscritte le *system call* di Unix in termini di chiamate agli oggetti di Mach, unendo così il parallelismo del sistema sottostante alla potenza (e popolarità) dell'interfaccia di Unix. Il lavoro attualmente in corso alla Carnegie Mellon consiste in una "*kernelizzazione*" (o "<dekernelizzazione>", in quanto per gli americani "to bone a chicken" o "to de-bone a chicken" ha esattamente lo stesso significato) di Mach, cioè nella creazione di una netta separazione tra le system call di Mach e le system call di Unix. Queste ultime dovrebbero diventare una *libreria* eseguita in stato utente, allo scopo di snellire ancor di più la struttura del kernel. Questo lavoro di kernelizzazione prosegue da diverso tempo, e pur non avendo dato particolari problemi implementativi, sembra per ora non soddisfare le esigenze di performance dei suoi progettisti. A breve scadenza dovrebbe comunque essere completata la nuova versione di Mach contenente questi miglioramenti.

Questa ulteriore evoluzione dovrebbe in futuro facilitare l'*emulazione* di altri sistemi operativi nell'ambiente Mach. Basterà infatti sostituire la "libreria" delle system call di Unix con una analoga reimplementazione delle primitive del sistema che si vuole emulare. Anche l'interfaccia di sistema, in altre parole, diventa una specie di "server", che condiziona pesantemente l'aspetto generale del sistema risultante. Nel frattempo, molte società del settore si sono impegnate nell'implementazione di un ambiente Unix System V "appoggiato" sul Mach attuale, ottenendo ottimi risultati.

5. CONCLUSIONI

Con l'eventuale diffusione del sistema Mach, la Carnegie Mellon potrebbe sostituirsi alla University of California di Berkeley come punto di riferimento accademico per l'evoluzione del mondo Unix. Mentre infatti le ultime versioni di Unix BSD hanno proposto praticamente solo dei miglioramenti delle funzionalità già esistenti, con Mach si ha una innovazione radicale nei concetti base del sistema operativo, e soprattutto una evoluzione funzionante ed efficiente verso le architetture multiprocessore. Non a caso, probabilmente, i progettisti di Mach hanno voluto non già proporre un sistema "più potente" o una interfaccia "migliore"[5], ma hanno preferito dimostrare che, mantenendo una interfaccia famigliare, possono aumentare funzionalità e prestazioni del sistema globale. Ed ovviamente la scelta dell'interfaccia è caduta su BSD4.3, la più diffusa a livello accademico.

Ma non solo le università sono interessate a Mach. Il fatto di avere "già' pronto" un sistema in grado di gestire il multiprocessore risulta commercialmente molto interessante anche per le case costruttrici di hardware. Molte società che già offrono Unix (o che hanno intenzione di offrirlo in futuro) si sono interessate a Mach, e molte hanno già effettuato un porting "sperimentale" sulle proprie piattaforme hardware. Il futuro prossimo ci dirà se questo interesse si trasformerà in una scelta commerciale.

A questa popolarità potrebbe contribuire in modo determinante il fatto che le due grandi "correnti di pensiero" che lavorano per la standardizzazione di Unix, cioè

[4] Questo naturalmente, a parere dell'autore, non toglie a Unix nessuno dei grandi meriti che da sempre possiede. Il noto autore D. Casati ebbe modo di osservare che Unix appare ai propri affezionati, più che come un semplice sistema operativo, come una vera e propria religione. L'autore di questo articolo, che di questa setta si sente parte, non può che concordare con la visione del dr. Casati.

[5] Qualunque cosa questi aggettivi possano significare in questo ambito.

Unix International ed Open Software Foundation, si stanno da tempo interessando a Mach come possibile "kernel standard" dello Unix che ognuna di esse propone. L'eventuale adozione di Mach in tal senso renderebbe di fatto tale sistema un *must* per qualunque casa distributrice di Unix, a livello mondiale. Il tutto evitando una gran mole di lavoro (e di litigi) alle compagnie coinvolte. Lo scorso Novembre OSF ha ufficialmente annunciato di voler utilizzare la tecnologia software di Mach.

Concludiamo osservando che, naturalmente, Mach non è l'unico sistema "evoluto" esistente. Diversi altri gruppi universitari ed industriali stanno lavorando in questa direzione. In particolare vanno citati, per la qualità e la lungimiranza del loro design, V-kernel della Stanford University [3], e DASH, un sistema operativo distribuito e multiprocessore che viene attualmente sviluppato alla Berkeley University dal gruppo del professor Domenico Ferrari [7]. V-kernel e DASH contengono idee e concetti in molti aspetti anche migliori di Mach. La vera differenza è che quest'ultimo è già uscito dallo stadio prototipale, e si è già proposto come un sistema utilizzabile e capace di supportare una interfaccia Unix. Di sicuro, comunque, le molte buone idee realizzate in tutti questi sistemi influenzeranno la struttura e l'implementazione dei sistemi operativi futuri. Che questi si chiamino, o meno, Unix.

6. BIBLIOGRAFIA

[1] M.J. Bach, S.J. Buroff MULTIPROCESSOR OPERATING SYSTEMS, AT&T Bell Laboratories Technical Journal Vol. 63, No. 8, October 1984.

[2] M. Young et al, THE DUALITY OF MEMORY AND COMMUNICATION IN THE IMPLEMENTATION OF A MULTIPROCESSOR OPERATING SYSTEM, 11th Symp. on Operating Systems Principles ACM, November 1987.

[3] D.R. Cheriton, THE V DISTRIBUTED OPERATING SYSTEM, Communications of the ACM 31(4), April 1988.

[4] Rashid, R.F., FROM RIG TO ACCENT TO MACH: EVOLUTION OF A NETUORK OPERATING SYSTEM, Proc. of the ACM/IEEE 1986 Fall Joint Computer Conference. ACM, November 1986.

[5] Wulf, W.A. Levin R., Harbison S.P., HYDRA/C.MMP: AN EXPERIMENTAL COMPUTER SYSTEM, Mc-Graw-Hill,1981

[6] Jones A.K., Chansler ,R.J., Drhamn, I.E. Schwans, K, Veglahl, S., STAR OS, A MULTIPROCESSOR OPERATING SYSTEM FOR THE SUPPORT OF TASK FORCES, Proc. of the 7th Symposium on Operating System Principles. ACM, December 1979

[7] Anderson, D.P., Ferrari, D., THE DASH PROJECT: AN OVERVIEV University of California at Barkeley, Techinical Reporter UCB/CSD 88/405, Feebruary 29, 1988.

RECENSIONI

Joel M. Crichlow,
AN INTRODUCTION TO DISTRIBUTEDAND PARALLEL COMPUTING,
Prentice Hall International, 1988 210 pp.

L'obbiettivo dichiarato del testo è quello di una introduzione all'argomento. Coerentemente l'autore ha scelto una tecnica esplosiva scorrevole basata sul discorso e non sull'uso di strumentazione matematica, nè di argomenti algoritmici, nè di dettagli descrittivi dello hardware.

Il tema del testo è quello dell'esecuzione in parallelo dei programmi; esso viene svolto facendo riferimento sia ai sistemi distribuiti propriamente detti e quindi basati su computer separati che realizzano il parallelismo a livello di processo, sia ai sistemi di elaborazione parallela in cui il parallelismo si svolge a livello di istruzione macchina. Il metodo utilizzato è quello di descrivere queste due classi di sistemi ispezionandone l'architettura hardware e software.

Dunque un testo non di disegno nè di tecnologia, ma di rassegna e di aggiornamento.

Il primo capitolo anticipa la nomenclatura utilizzata nel seguito: per esempio, sono anticipate le diverse architetture multiprocessore, multicomputer, parallele, di rete ecc., le tecniche di comunicazione tra task separate operanti sui sistemi di cui sopra, e quelli di sincronizzazione.

Il secondo capitolo entra nell'organizzazione del calcolatore in senso hardware: analizza le tradizionali soluzioni orientate al multiprocessing: dal meccanismo di context switing alle architetture a pipeline, ai processori vettoriali (quelli che eseguono una stessa istruzione su un vettore di operandi), ai multiprocessori strettamente connessi (quest'ultimi approfonditi, peraltro, nell'aspetto di intercomunicazione).
Per le architetture parallele sono descrittii processori associativi, quelli che permettono di velocizzare la ricerca dei dati in memoria utilizzando tecniche di comparazione diretta del contenuto; quindi seguono gli array processor di tipo SIMD, MIMD e sistologici. L'ultimo paragrafo è per le macchine non Von Newmann, di tipo "data flow" e RISC.
Il capitolo tre dà una breve introduzione alla tecnologia delle comunicazioni, sfiorando la teoria dei segnali, la conversione analogica, la modulazione, i mezzi trasmissivi, ecc., quindi passa all'esame delle architetture delle reti di calcolatori inchiave ISO OSI; si conclude con un breve cenno alle principali reti locali disponibili.

Il capitolo quattro discute dei sistemi operativi: è uno dei capitoli meglio riusciti per la completezza della trattazione, sia pure a livello introduttivo. Esso analizza i sistemi operativi di reti (con un trattamento da citare del problema dell'agent-task) ed i sistemi operativi per architetture distribuite.
Di quest'ultimo argomento sono definite le caratteristiche generali (integrità dei dati, il controllo morbido delle malfunzioni, la privatezza dei dati e la sicurezza degli accessi, le prestazioni), la comunicazione interprocesso con gli schemi remoti procedure call e message passing, e la ripartizione delle risorse. Quindi seguono le descrizioni di esempi molto famosi (Accent, V Kernel, Amoeba, SODS/OS, The Newcaste Connection, Locus) ed un breve cenno ai tre approcci noti per i sistemi operativi delle macchine parallele (master/slave, floating supervisor e distributed control).

Il capitolo quinto è totalmente dedicato alla descrizione del modello cliente-servente di rete nei seguenti casi: file server, name server, printer ed electronic mail server; da notare la trattazione per i file server che comprende l'elenco dei servizi, l'organizzazione del software, i protocolli, la sicurezza degli accessi e l'integrità dei dati. Sempre per questo tema sono riassunte le principali funzioni dei seguenti esempi: Xerox Distributed File System, Campridgfe File Server, e Swallow

Il capitolo sesto è dedicato ai data base distribuiti; anche questo è un capitolo fra i migliori del testo per la sua concisa completezza. E' organizzato in due parti: la prima presenta i concetti generali della struttura di un db (a listeinvertite, gerarchica, reticolare, relazionale), quelli relativi sia al processo di ricerca dati in un db relazionale (con esempi dell'algebra applicata) che al metodo di disegno delle relazioni. La seconda parte è

dedicata al problema della distribuzionedei dati in cui i parametri da considerare riguardano i volumi dei dati e delle attività, il numero degli elaboratori partecipanti, i carichi di comunicazione ed il modello utilizzato (verticale, orizzontale, con replica dei dati, ecc.); quindi affronta le tecniche di interrogazione, con l'analisi delle sottorisposte, l'aggiornamento degli archivi ed i controlli di integrità.
Il tema è trattato classicamente con la soluzione transazionale, esemplificata sia nel modello a dati replicati che con quello a dati interamente distributivi; vi è un accenno alle malfunzioni e ad altre tecniche di recovery. Il capitolo termina con tre esempi: SDD-1, R*, Distributed INGRES.

Il capitolo sette affronta i linguaggi paralleli con il seguente indice: per array processor di tipo von Newman (come il caso ILLIAC IV) è descritto il linguaggio Actus, per architetture distribuite su mini-microelaboratori sono citati i linguaggi Concurrent Pascal, Communicating Sequential Processos (CSP), di Von Newman è citata la programmazione funzionale, il linguaggio Lips e quello data flow CAJOLE.

Carlo Lunghi

W. G. Schneeweiss,
BOOLEAN
FUNCTIONS WITH ENGINEERING APPLICATIONS COMPUTER PROGRAMS,
Spinger, 1989.127 figure, Pagine xii+264.

Alcune applicazioni ingegneristiche delle funzioni booleane sono ben note: per esempio la progettazione dei circuiti digitali si basa sull'algebra di Boole. Nei libri che trattano di tali circuiti c'è solitamente un capitolo sull'algebra e le funzioni booleane, che già risulta abbastanza ostico agli studenti con un'avversione per la matematica.
Questo volume colma in larga misura la lacuna esistente tra il troppo e il troppo poco, presentando in maniera ragionevolmente estesa la teoria delle funzioni booleane ed illustrandone nel contempo le principali applicazioni, mantenendosi a un livello sempre più accessibile. Particolare rilievo viene dato nel testo alla stocastica delle funzioni booleane, cui vengono dedicati due capitoli. Le funzioni stocastiche booleane hanno applicazioni in problemi di affidabilità e sono, in genere, completamente ignorate nei testi sulle reti logiche. Un'utile appendice fornisce i presupposti di teoria delle probabilità delle trasformate di Laplace.
Un altro argomento insolito cui è dedicato un capitolo è costituito dalle "funzioni booleane senza operatori booleani". Le funzioni booleane furono espresse mediante operatori aritmetici già nel volume "The Laws of Thought" di Boole (1854), e questo è spresso un utile artificio per semplificare i calcoli, specialmente nelle considerazioni probabilistiche. Ne derivano tra l'altro l'interessante algoritmo di Enzmann (1973), che non ho mai visto citato in altri testi, forse anche perchè l'articolo originale, in tedesco, è poco noto.
L'autore evita dichiaratamente ogni considerazione sulla complessità degli algoritmi, e rimanda in proposito a due volumi piuttosto generici. Il trattato di Wegener sulla complessità delle funzioni booleane, pur citato come ultimo riferimento, non sembra essere menzionato nel testo, e nel libro "The Complexity of Computer" è ignorato. Peccato, perchè questo è certamente un altro argomento negletto nei testi di impostazione applicativa, mentre una trattazione che tendesse ad accrescere nei lettori una consapevolezza di questi importanti aspetti sarebbe certamente più completa: basti pensare alla rilevanza di una lacuna di questo tipo quando si parla degli algoritmi di minizzazione.
Un po' di spazio per parlare di complessità avrebbe forse potuto essere rubato al lungo capitolo su "algoritmi e programmi", che presenta schemi di programmi per gli algoritmi già descritti nei capitoli precedenti, aggiungendo osservazioni e dettagli per l'implementazione in Pascal.
Molto opportuno, ma breve risulta il capitolo sul calcolo della differenza booleana, con un rapidissimo cenno al problema della diagnosi e localizzazione dei guasti nei circuiti combinatori: anche qui, sarebbe stata utile almeno qualche ulteriore indicazione bibliografica. Da citare infine la presenza di circa 60 esercizi, tutti completamente risolti al termine del volume

Mauro Torelli

Le Van Huu ,
INTRODUZIONE ALLA STRUTTURA INTERNA DI UNIX,
Edizioni 1989

Anche in Italia il sistema operativo UNIX sta sempre

più uscendo dall'ambito accademico, e si sta diffondendo negli ambienti industriali e commerciali più diversi, e non solo come ambiente di sviluppo software.
Le proiezioni future riguardanti la diffusione di UNIX nel mondo, e anche in Italia, indicato un tasso di crescita notevole. A livello mondiale si é passati dai 10.000 sistemi istallati del 1983 al milione del 1988 ed il trend di crescita per il 1992 sembra essere ancora superiore (anche se fonti diverse forniscono diverse percentuali. Per quanto riguarda l'Italia, da circa 14.000 sistemi del 1987si dovrebbe passare a più di 60.000 nel 1992.
Questo indica come il sistema operativo UNIX stia diventando sempre più un oggetto che interessa non solo gli studenti di facoltà di scienze dell'informazione, ma tutti gli operatori del settore.
In questo senso il libro "Introduzione alla struttura interna di UNIX" si presenta, come il suo stesso autore indica, sia come testo universitario, sia come testo di divulgazione e consultazione per progettisti software e programmatori che operino nell'ambito del software di base, o che ad esso si interfaccino da livelli più applicativi. Esistono pochi testi realmente validi che descrivano i sistemi operativi, ed ancora meno ne esistono che descrivano UNIX in modo organico e dettagliato. Questo è dovuto anche al fatto che la prima implementazione, a opera di Thompson, Ritchie ed altri dei Bell Laboraziones AT&T, è diventata in questi anni un nucleo attorno al quale sono state di continuo aggiunte" funzionalità" di tutti i tipi, spesso con disuniformità di interfacce, e dimenticando la filosofia delle "poche idee ma buone" propria dei realizzatori originali di UNIX. Associazioni internazionali come UNIX International e Open Software Fondation, guidati da forti obiettivi commerciali, stanno cercando di mettere ordine tra le varie versioni di UNIX esistenti.
Il testo di Le Van si concentra quindi sulla struttura interna di UNIX e , pur non entrando sempre in dettagli che a volte sarebbe stato opportuno precisare, ne descrive tutti gli aspetti in modo chiaro e sistematico.
La trattazione è divisa in sei capitoli . Il primo dà una descrizione generale dell'architettura di UNIX: partendo da alcuni brevi cenni storici, si passa alla descrizione dei concetti di base di UNIX, e poi alla definizione dei "livellidi funzionamento". Il secondo capitolo si concentra sulla descrizione del file System, elencando i tipi di file di UNIX prevede, la loro organizzazione logica, (relativa a come il programmatore "vede" i file), e la loro organizzazione fisica (relativa ai dispositivi periferici che UNIX sottende al concetto di "file").Questo capitolo include anche informazioni interessanti inerenti le operazioni di "bootstrapping" di una macchina gestita da UNIX. Le primitive di Imput/Output fornite da UNIX sono considerate nel capitolo tre, dove le principali system call per la gestione dei file sono elencate e descritte in dettaglio.
Il quarto capitolo si occupa della gestione dei processi, dall'insieme dei possibili stati in cui un processo può trovarsi, alla gestione di interruzioni, ai segnali che i processi possono scambiare tra loro e con il sistema operativo. Questo capitolo descrive anche il meccanismo di memoria virtuale, che forse avrebbe meritato uno spazio più esteso, date le importanti applicazioni che essa ha con gli altri aspetti del sistema.
Il capitolo cinque descrive le primitive che UNIX mette a disposizione di un processo, all'esecuzione di un programma, alla gestione dei segnali che i processi possono inviare, alla terminazione di un processo. Un'area particolarmente interessante è quella che descrive la comunicazione tra processi, tramite i meccanismi di pipe, di scambio di messaggi e di condivisione della memoria.
Infine il sesto capitolo dà una introduzione alla Bourne shell (uno degli interpreti di comandi che UNIX mette a disposizione), fornendo anche interessanti dettagli relativi alla implementazione della shell stessa.
Un'ottima bibliografia conclude il testo di Le Van .
Di esso osserviamo unico refuso, che ci auguriamo solo tipografico: nei cenni storici viene citato un certo BEN Thompson, che con UNIX non ci risulta abbia mai avuto a che fare...

Giuseppe Facchetti

Gordon Clarke,
EXPERT SYSTEMS IN THE CITY,

IBC Financial Books, London, 1989

I Sistemi Esperti stanno vivendo, nella seconda metà degli anni ottanta, un periodo molto delicato di confronto con il mercato produttivo. E' noto che l'Intelligenza Artificiale, ambito scientifico e tecnologico all'interno del quale ha preso avvio l'impostazione metodologica con cui vengono oggi realizzati i Sistemi Esperti, ha vissuto alterni momenti di popolarità nel corso della sua storia ultratrentennale, ma proprio negli anni ottanta ha dovuto condurre la propria battaglia più aspra con le

aspettative applicative generate da lustri di cospicui investimenti ed "entusiasmanti" risultati teorici.

Proprio ai Sistemi Esperti, oltre che ad alcune discipline specifiche della robotica, é stato affidato il compito di fornire una prima risposta applicativa , ma soprattutto commerciale, a tali aspettative. E le risposte più consolidate, sicuramente le più confortanti, provengono oggi da uno degli ambiti applicativi trascurati da chi per anni ha progettato Sistemi Esperti sperimentali al fine di evidenziarne le potenzialità: sono infatti le problematiche finanziarie che oggi detengono il primato di successi applicativi per i Sistemi Esperti, sia per il numero di sistemi realizzati, e quindi di problemi diversi affrontati, sia per diffusione, vantando ormai migliaia di installazioni operative.
Questo recente ed aggiornato libro di Gordon Clarke, autore noto del mondo anglosassone per la sua profonda esperienza ed attenzione ai problemi di automazione bancaria, intende fugare qualsiasi dubbio residuo circa l'opportunità offerta dai Sistemi Esperti per la soluzione di una classe di problemi che spesso costituiscono il terreno di competizione tra le diverse istituzioni finanziarie. Il libro rappresenta la logica e storica evoluzione di *Expert Systems - A Businessman's Guide*, scritto nel 1986 a quattro mani dallo stesso Clarke in collaborazione con Andrew Morris in concomitanza con l'avvio in Gran Bretagna di un significativo numero di esperienze nel settore e con la pubblicazione dei primi risultati del progetto di ricerca Alvey, finanziato dal Governo britannico.

Come esplicitamente dichiarato dall'autore nella prefazione di questo libro, *Expert System in the City* è destinato in particolare a tre categorie di professionisti:
-i " business managers", ovvero i potenziali utenti di queste applicazioni, che vengono portati per mano attraverso una serie di esperienze concrete, verso la dimostrazione del fatto che esistono reali opportunità commerciali alle quali solo questa tecnologia può fornire subito risposte esaurienti;
- gli esperti di tecnologie dell'informazione, ai quali potrebbe essere affidata la gestione dei progetti di Sistemi Esperti in grandi strutture di servizi finanziari (Banche, Assicurazioni, ecc.), e che troveranno in questo testo illuminanti passi relativi soprattutto alla prudenza e all'attenzione che occorre riporre nelle fasi più critiche del progetto per evitare di allungare la lista già cospicua di applicazioni inutili o abbandonate, realizzando invece strumenti che si ripaghino rapidamente diventando indispensabili supporti quotidiani al "business";
- i fornitori di soluzioni basate sui Sistemi Esperti che, in un ventaglio di applicazioni realizzate e di sperimentazioni concluse, troveranno utili indicazioni sugli strumenti di sviluppo, le strategie di marketing, di formazione e di informazione, nonchè le "nicchie" più produttive per i Sistemi Esperti applicati alle problematiche finanziarie.

Il libro è diviso in tre sezioni, la prima delle quali presenta un'introduzione all'argomento attraverso la storia, le prospettive e le esperienze più significative sviluppate nelle diverse realtà geografiche e tecnologiche del panorama internazionale. Buona parte delle considerazioni contenute in questa sezione derivano da una lettura ragionata dei dati pubblicati nel recente rapporto della OVUM Ltd. "Expert Systems in Banking and Securities", che rappresenta la fotografia più completa e aggiornata dello stato dell'arte per questo argomento.

La sezione II del libro esplora un approccio strategico nella valutazione delle concrete opportunità offerte dall'applicazione dei Sistemi Esperti allo specifico ambito dei problemi connessi all'attività di chi fornisce servizi finanziari. In questo contestol'autore sottolinea che per ottenere un Sistema Esperto di successo non bisogna mai dimenticare due fattori determinanti ad un problema *reale*, la cui soluzione sia effettivamente necessaria per elevare il livello di qualità della struttura; troppo spesso infatti una eccessiva prudenza consiglia di affrontare tematiche marginali, che fatalmente conducono a risultati parziali che non giustificano gli investimenti profusi. Il secondo fattore di successo è proprio il corretto dimensionamento tra l'investimento in risorse e risultati attesi, individuando con precisione gli ambiti in cui i Sistemi Esperti rappresentano una soluzione più conveniente rispetto ai sistemi software convenzionali, e scegliendo il momento in cui passare dal progetto all'installazione operativa, dato che i Sistemi Esperti si presentano facilmente ad infinite evoluzioni che rischiano di infrangere la soglia di convenienza.

La terza sezione è destinata ad individuare le aree di applicazioni più favorevoli, attraverso la disanima di numerosi esempi concreti che rappresentano sicuramente un campione significativo. Ragionando ad alto livello, Clarke indica due classi di Sistemi Esperti realizzabili in questo contesto: la prima, intorno alla quale si è sviluppata ad oggi la maggiore esperienza, è rappresentata dalle basi di conoscenza destinate a fornire un supporto decisionale nelle attività che richiedono un livello di "expertise" elevato; la seconda classe, per la quale l'autore prevede un crescente interesse nel prossimo futuro, contiene quelle applicazioni che, opportunamente integrate, forniscono un "plus intelligente"al tradizionale data processing. La "chicca" di questa sezione è però sicuramente rappresentata da una sorta di ricetta per il sistema esperto di successo: un condensato di buon senso e ottimi consigli, frutto di un attento lavoro di sintesi su tutto quanto precedentemente presentato.

A completamento della trattazione, dopo aver sviluppato alcune ragionevoli ipotesi sulle prospettive di consolidamento dei Sistemi Esperti nelle attività finanziarie, il testo fornisce una serie di utili appendici, con un completo glossario dei termini e una aggiornata ed esauriente bibliografia.

Nel complesso *Expert Systems in the City* fornisce pertanto un insieme di utili indicazioni non solo a chi opera nel settore finanziario, ma anche a tutti coloro she intendono comprendere le caratteristiche di questa tecnologia e individuarne le potenzialità in differenti ambiti applicativi confrontando esperienze che nascono da esigenze molto diverse tra loro pur riferendosi tutte al mondo finanziario. Forse l'unico limite di questo testo deriva dall'origine britannica dell'autore, che lo porta, soprattutto nelle parti analitiche della trattazione, a concentrarsi maggiormente sulle esperienze sviluppate in Gran Bretagna e sulle esigenze di quel mercato, riducendo quindi un poco "l'università" delle considerazioni proprste.

Alessandro Malacar

EDITO DA CALEIDOGRAF S.R.L.
VIA L. DA VINCI N.4/6 TEL. 039 - 587541
22058 OSNAGO (CO)

Spedizione in abbonamento postale, Gruppo IV, 1° semestre.
Pubblicità Inferiore al 70%
N. '36, 46 - Dicembre 1989
TAXE PERCUE