View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by El Servicio de Difusión de la Creación Intelectual

EVALUATION OF A LOCAL STRATEGY FOR HIGH PERFORMANCE MEMORY MANAGEMENT*

Edson Toshimi Midorikawa, João Antônio Zuffo
{emidorik, jazuffo}@lsi.usp.br
Laboratory of Integrated Systems
Department of Electronic Engineering

Liria Matsumoto Sato
liria@pcs.usp.br
Department of Computer Engineering
and Digital Systems

Polytechnic School of the University of São Paulo
Av. Prof. Luciano Gualberto, travessa 3, no. 158
05508-900 - São Paulo SP Brazil

ABSTRACT

Conventional operating systems, like Silicon Graphics' IRIX and IBM's AIX, adopt a single memory management algorithm. The choice of this algorithm is usually based on its good performance in relation to the set of programs executed in the computer. Some approximation of LRU (least-recently used) is usually adopted. This choice can take to certain situations in that the computer presents a bad performance due to its bad behavior for certain programs.

A possible solution for such cases is to enable each program to have a specific management algorithm (local strategy) that is adapted to its memory access pattern. For example, programs with sequential access pattern, such as SOR, should be managed by the algorithm MRU (most-recently used) because its bad performance when managed by LRU. In this strategy it is very important to decide the memory partitioning strategy among the programs in execution in a multiprogramming environment. Our strategy named *CAPR* (Compiler-Aided Page Replacement) analyze the pattern of memory references from the source program of an application and communicate these characteristics to the operating system that will make the choice of the best management algorithm and memory partitioning strategy.

This paper evaluates the influence of the management algorithms and memory partitioning strategy in the global system performance and in the individual performance of each program. It is also presented a comparison of this local strategy with the classic global strategy and the viability of the strategy is analyzed. The obtained results showed a difference of at least an order of magnitude in the number of page faults among the algorithms LRU and MRU in the global strategy. After that, starting from the analysis of the intrinsic behavior of each application in relation to its memory access pattern and of the number of page faults, an optimization procedure of memory system performance was developed for multiprogramming environments. This procedure allows to decide system performance parameters, such as memory partitioning strategy among the programs and the appropriate management algorithm for each program. The results showed that, with the local management strategy, it was obtained a reduction of at least an order of magnitude in the number of page faults and a reduction in the mean memory usage of about 3 to 4 times in relation to the global strategy. This performance improvement shows the viability of our strategy.

It is also presented some implementation aspects of this strategy in traditional operating systems.

* This paper was supported in part by Finep (Brazilian Funding Agence) under Hypersystems Project.

1. INTRODUCTION

A series of research work has been motivated by the problem of the memory systems in current high-performance computers. The literature presents a very big group of alternatives that was produced as a result of these researches conducted by several groups in universities and research centers around the world. In a general way, most of them deals with the aspect of the memory latency, that is to say, of the long time to access the resident data in the main memory [WULF95][SAUL96].

The proposed methods can be classified in two categories: the first tries to reduce the latency and the second attempts to tolerate the latency. The techniques to reduce the latency of the medium access to the memory operands can be divided in two classes: to reduce the number of faults[1] and to reduce the time spent in the faults. The techniques to improve the hit ratio of cache memories are a classic example of the strategy of reducing the number of faults [CARR94], because the number of requisitions to the main memory is reduced maintaining data closer to the processor. The techniques to reduce the time spent in the faults include faster interconnection networks [HEXS94], multiple levels of cache memories and DRAMs with smaller access times.

As a matter of fact, these solutions have been improving the performance of the modern memory systems. Unfortunately, due to some difficulties, there isn’t a definitive solution for the problem. This paper presents our contribution in the search of this solution. Finally, we argued that the solutions adopted now, in a general way, don't solve the problem of performance of memory systems completely and we present a new strategy for improving the memory performance. Our strategy attempts to change the current approach used in modern operating systems. Some decisions could be took off from the operating system and managed by other system programs.

As the modern systems of parallel computation offer support for temporal and spatial multiprogramming, new management techniques become necessary. This work proposes and evaluates a strategy for this situation. Our conviction is that, while the researches in the several areas of high performance computing continue to be accomplished in an independent way, they won't find an effective and durable solution for the several problems. What intends here is to show a first step in direction of an integrated system, where all system components participate actively in the resource management of the system and in the execution of the programs.

The rest of this paper is organized in the following way. We present a brief review of current research in resource management in high-performance systems in section 2. The framework of our current research is presented in section 3. Section 4 describes the local strategy for memory management studied here. We next present our experimental results in section 5. Relevant issues about the implementation of our strategy are discussed in section 6, and we present our conclusion and future work in section 7.

2. HIGH PERFORMANCE MEMORY MANAGEMENT

Resource management is a great challenge for modern parallel operating systems [VERH98]. As parallel machines are now being used, new strategies should be devised for this class of machines. One reason for this necessity is the current approach adopted in the operating systems: in general, current resource management algorithms are simply a parallel extension of traditional, monoprocessor algorithms.

Ben Verghese points one problem in his thesis [VERG98]: “current operating systems have little support for controlling the allocation of resources to groups of processes, or for providing fairness by any abstraction other than individual processes... The lack of control over resource sharing leads to poor isolation in current shared-memory multiprocessors. As a result

[1] This includes cache misses and page faults.

of unconstrained sharing, the performance seen by an individual user in a multi-user environment is dependent on the load placed on the system by other users. Users have no reasonable guarantee of minimum performance even if they are using much less than the share of the resources they are assigned to. A single user or process can easily load the system unfairly, and tie up a large fraction of the resources. Examples of activities that can lead to unfair load are a user starting many processes, a *process touching a large number of pages resulting in a huge working set*, or a process making a large number of accesses to a disk."

Let's consider the second example presented by Verghese. It happens because current operating systems use a "recent past analysis" approach to decide which pages to keep in main memory. Then, greedy processes tend to be favored in regard of other processes because in the "recent past" they accessed more memory pages than the others.

Accurate control over resource allocation is desirable across a broad spectrum of processes with different behavior. Published work usually concentrates in the processor scheduling area. Extensive research produced many classes of algorithms: priority scheduling, real-time scheduling, fair-share scheduling, proportional-share scheduling, microeconomic resource management and rate-based network flow control [WALD95] [STOI95] [CHER93].

Conventional operating systems employ numerical priorities for scheduling processes [STAL98]. A priority scheduler simply grants a processor to the process with the highest priority. Traditional operating systems, like Unix [VAHA96] and Windows NT [SOLO98], adopt a decay-usage variant strategy, where the priority is dynamically changed with the recent processor usage.

Recently, John Chapin discussed the memory prioritization problem in [CHAP97]. He argues that the way computer systems are used has changed substantially. These changes are sufficient to require reevaluating some fundamental assumptions made in operating systems memory management. One obvious problem motivating this reevaluation is that current systems do a terrible job of maintaining performance for high-priority processes when the system comes under pressure due to the behavior of low-priority processes. There is no memory scheduling policy inside the current operating systems.

Current memory management algorithms are designed to ensure that a process that receives more CPU time will keep a larger working set in memory. These algorithms are less effective today than they were twenty years ago since processors have increased substantially in performance relative to the other system components. Another cause for this problem is that current page replacement algorithms are designed to emphasize overall system throughput rather than process prioritization. This objective was adopted in early timesharing systems [DENN80]. The global clock algorithms used in many Unix variants seek to maximize the probability that the next reference issued by the processor will not cause a page fault. The working set trimmer in Windows NT appears to seek to equalize the page fault rate of different processes [SOLO98].

Edwards and Cao presents an experiment with memory allocation limitation mechanisms [EDWA96]. They propose a user-oriented resource management approach, introducing a set of simple kernel mechanisms to allow users to adjust memory allocation and I/O priorities, and an authorization scheme to prevent users from misusing the kernel mechanisms. Their approach includes a user-level policy daemon, which monitors and dynamically adjusts resources to create a better working environment for the user.

The above examples show the need for new approaches for memory management. Many other aspects must be taken into consideration, such as, new 64-bit microprocessors (huge virtual address space and virtual memory), multithreaded processors, multiple gigabyte main memories, parallel programs using multiple threads, etc. All these topics must strongly affect the way operating systems manage memory.

3. FRAMEWORK

Current systems use a sequential approach of separate phases for memory management of application programs [MIDO97a]. In this traditional approach, each system component is responsible for just one task and executes it as best as possible (at least it tries to). Although a component could possibly help another one, there is no interaction among them. There are many possibilities to explore such interactive approach. We present some examples:

- during the execution of an application program, the *operating system* can collect information about the memory access pattern, memory demand of different processing stages of the application, and the system behavior. Such data could be examined and used by the *compiling system* to generate an improved executable program that is better adjusted to the computer system;
- the execution data collected by the *operating system* can also be used by the *linker* to modify the composing strategy of object modules to create the executable program. Current linkers use simple strategies, like inserting the object modules in the same sequence of input files or ordering them by size (ex. Gnu C);
- based on the memory access pattern, the *programmer* could restructure his source program, organizing its variables in a different way. For example, defining a matrix of structured types instead of a set of simple matrixes;
- the *compiler* has the knowledge of general structure of the program. So, it can insert some special directives to inform the *operating system* about future behavior of the program. Examples of some decision areas that can be aided by this situation are page replacement, dynamic memory management, garbage collection and memory-conscious process scheduling).

Under our proposal, named ***Communion***, the system programs "work together" in order to achieve enhanced performance [MIDO97a]. The figure 1 shows a diagram of the ***Communion*** approach.

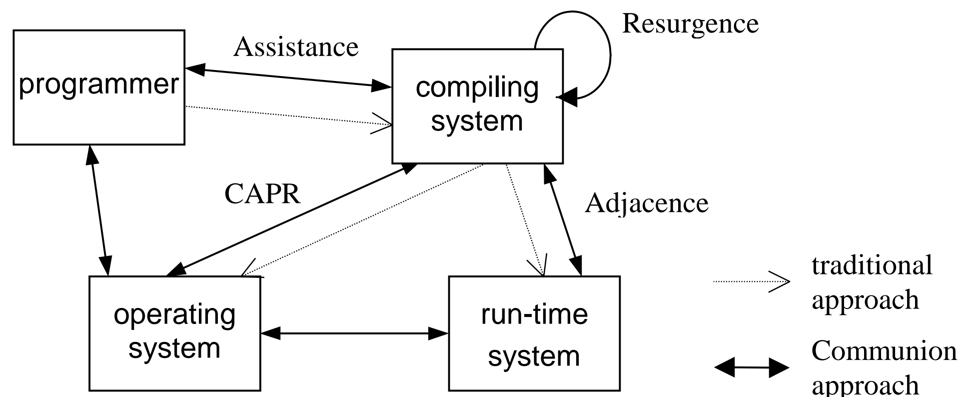

Figure 1 - Interaction among system components.

It can be noted from figure 1 that our approach includes the *programmer*. It is our belief that the most important component of a computer system is the one that design and implements the application. If the program isn't well designed, although the other components do their best, all they can achieve is just "order the messy". In order to achieve such objective, it's necessary to supply the programmer with some program execution behavior information. Traditional systems only provide the total execution time and average memory usage. Modern computer systems require a new approach to program design and tuning, an integrated approach.

In the next sections we detail one specific interaction: between the compiler and the operating system (named ***CAPR***), presenting its application to high performance memory management. The others are detailed elsewhere [MIDO97b] [MIDO98].

4. A LOCAL STRATEGY FOR MEMORY MANAGEMENT

CAPR was the first developed strategy of ***Communion*** [MIDO95]. It is a form of interaction between the compiler and the operating system. In this section we first review the nature of page replacement algorithms to detect their weaknesses, and propose this novel memory management technique.

4.1. Page Replacement Algorithms

In a virtual memory system, one of the most important management policies is the one that controls the choice of pages to be removed from the memory in order to make room for the page to be brought in. This policy is implemented by page replacement algorithms.

There are several algorithms proposed in the literature in the last three decades. The criteria for choosing the page and the variability of program resident set size are some characteristics that differentiate these algorithms. Two classes of algorithms are often used in modern systems: fixed partitioning and variable partitioning. The main difference between them is the number of pages in the resident set. If the resident set size is a fixed constant, then it's said that a fixed partitioning approach is being used. Otherwise, we say that the algorithm has a variable partitioning approach. LRU, FIFO, Clock, NRU and LFU are some examples of fixed partitioning algorithms. Examples of the other approach are Working Set and PFF.

We can say that all page replacement algorithms must answer the following question: "*among the pages in main memory, which one is that will be referenced aftermost in future?*". All the algorithms above have one point in common: the choice of the page is based on the knowledge of past behavior of the program. The basic strategy adopted by all these algorithms is "*use the past and/or the present as an indication of the future*." Consider the following examples:

- LRU ("*least recently used*"): make the choice based on an analysis of the recent past, where it is determined by the page that is not referenced for the longest time.
- Clock: discard pages not referenced since last clock scan in a circular list of pages (it is an approximation of LRU).
- WS ("*working set*"): maintain in memory only those pages that were accessed during a time window (in past).

However, studies show that "the past and the present are not good indications of the future." So, it's clear we need to adopt a new strategy for the replacement algorithms. This need is confirmed in [FRANK78] that presents a study of behavior anomalies in some classical replacement algorithms.

4.2. The CAPR Strategy

An alternative is to endow the operating system with some source of information about the future behavior of the programs. Thus, it is possible to get more precise indication of the future and then make a better choice. In order to do this it is necessary to know the programs structure, detect their localities and their transitions. Among the system programs, it is the compiler that has such an information.

Our strategy, named *CAPR* ("*compiler-aided page replacement*"), presents a new memory management technique based on the interaction between the compiler and the operating system. In *CAPR*, both system programs exchange information related to the memory requirements and usage.

After analyzing the source program, the compiler detects possible sources of locality and automatically inserts directives to inform the operating system. Examples of directives are:

- locality change
- change in memory requirement
- change in algorithm specific parameters

- change the management algorithm

On the other hand, after executing the program, the operating system can send all data collected during the execution to the compiler informing the memory access patterns and localities characteristics. Using this information, the compiler can restructure the program code and data in order to improve performance. Examples of information that the operating system can send to the compiler are:

- page-fault rate of selected sections of the program
- average resident set size
- total execution time
- execution space-time product

So, this strategy proposes a custom memory management technique that is the most adequate for that program. The figure 2 illustrates the interaction between the compiler and the operating system.

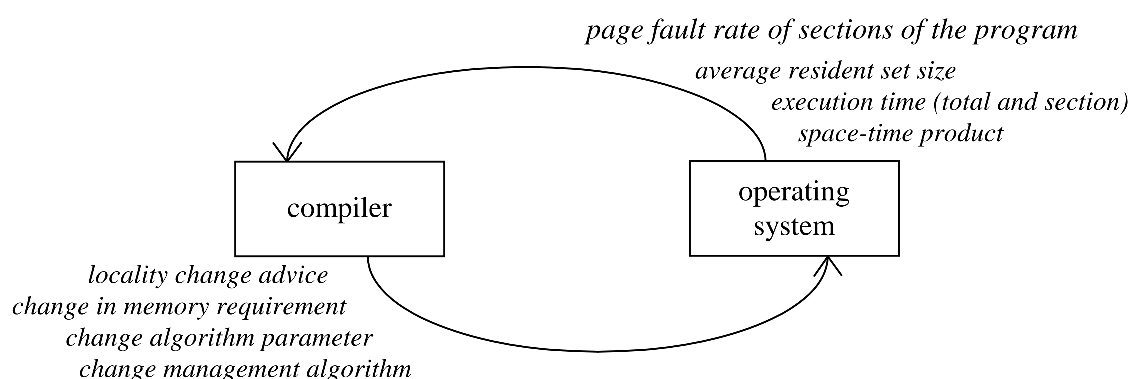

Figure 2 - Interaction between compiler and operating system under CAPR strategy.

The main objectives of the CAPR strategy are:

- to get a better system performance;
- to lower the page fault rate;
- to enhance the memory system usage.

CAPR does not propose new page replacement algorithms as the *Compiler Directed* (CD) approach of Mohammad Malkawi [MALK86]. Our approach consists in augmenting each traditional algorithm with additional functionality, like controlling the resident set size and dynamically modifying some parameters. For example, for the Working Set algorithm, it's possible under *CAPR* approach to define upper and lower limits of the resident set size and to have different virtual time windows in some sections of code.

4.3. A Local Memory Management Strategy

Traditionally, operating systems adopt only one page replacement algorithm for managing all the processes in execution in the system. The choice of this algorithm takes into account many aspects: low implementation and execution overhead, good performance and previsibility of behavior. The adopted algorithm is usually LRU (or some approximation like Clock or NRU) or FIFO.

CAPR allow the possibility of choose a particular page replacement algorithm for each program, dependent on the memory access pattern. The compiler can insert a special directive to the source code of the program informing the better management algorithm for the program access characteristic.

5. STRATEGY EVALUATION

We developed a study where we verified the effects of *CAPR* in a multiprogramming environment, where the main objective was to reduce to memory occupation during the execution of a program. We made an analysis of the viability of this strategy with the study of the influence of the adoption of multiple memory management algorithms in the system, and additionally, a study of the influence of the memory partitioning strategy. After that, we compare the results of our study with the adoption of a global strategy.

The following programs were adopted for the accomplished study:

- *local:* synthetic program that presents a high data access locality. It represents the applications with an appropriate pattern of accesses to the memory system. It presents a total size of 492 pages;
- *seq:* synthetic program with a sequential pattern of memory accesses. It tries to evaluate extreme cases of applications as the SOR. It composes a total size of 54 pages.

Although real applications have not been used in our evaluation, the obtained results can be used to verify the effects of our strategy: adoption of multiple local page replacement algorithms and influence of strategies of memory partitioning among active processes.

5.1. Evaluation Infrastructure

The benefits of the *CAPR* strategy was evaluated with the implementation of a simulation system of memory systems, named Elephantus. Due to the complexity of incorporating of the *CAPR* algorithms in a real operating system, like Linux or FreeBSD, we decided for the use of simulation techniques.

In *Elephantus*, we adopted the simulation technique driven by direct program execution ("execution-driven simulation") and the simulation technique driven by program traces ("trace-driven simulation").

Elephantus is composed by five main components:

- *on-line simulator:* responsible for the execution-driven simulation. It executes the instructions of the application program and the CAPR directives and makes the collection of the statistical data of the simulation. Based on the Mint [VEEN93] and Augmint [SHAR96] software packages, it's available for simulating machines with MIPS R3000, Sparc and x86 processors;
- *off-line simulator:* responsible for the trace-driven simulation. Starting from a file of program traces, it executes the processing of the date accesses and generates the output files. It implements multiple page replacement algorithms and a multiprogramming simulation environment too;
- *trace file generator:* in the same way that the on-line simulator, it executes the instructions of the application and, like this, generates a file of traces representing the program references to the memory and the specification of the CAPR directives;
- *post-processor:* based on a file of the program execution events (execution traces), it makes an analysis of the data and generates graphs and a statistical summary;
- *graphical interface:* implemented in Tcl/Tk, it allows the configuration of the simulation parameters and the visualization and analysis of the final results (fig.3).

5.2. Evaluation Metrics

In order to evaluate our strategy, we adopted three metrics to compare the alternatives and policies:

- *number of page faults*: the total number of page faults that was generated during the total program execution time;

- *space-time product*: a program's space-time product is the integral of its resident set size over the time it is running or waiting for a missing page to be swapped into main memory [DENN80].
- *average resident set size*: it can be defined as the ratio between the space-time product and the total execution time. It represents the average memory occupancy of the program over its execution.

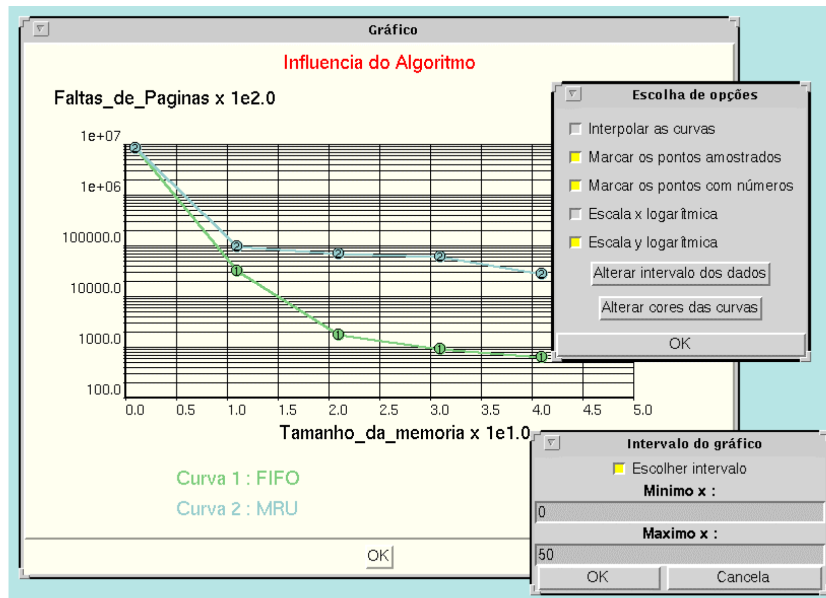

Figure 3 - Elephantus graphical interface.

5.3. Selection of the Local Replacement Algorithm and Memory Partitioning Strategy

It's known that the system performance is dependent of the memory management algorithm. One reason for this behavior is based on the programs memory access pattern: for example, for programs that exhibit a sequential access pattern, the MRU algorithm performs better than LRU or FIFO [GLAS97] [MIDO97b]. Figure 4 presents the page fault behavior for each program under LRU and MRU memory management algorithms.

We decided to verify the influence of the adoption of a local strategy, with the management of each process with an individual algorithm. In accordance with our previous work, we chose the algorithm LRU for the `local` program and MRU for the program `seq`. The available memory was partitioned in a *static* way among the processes, that is, the reserved memory for each process is maintained unchanged during the process execution.

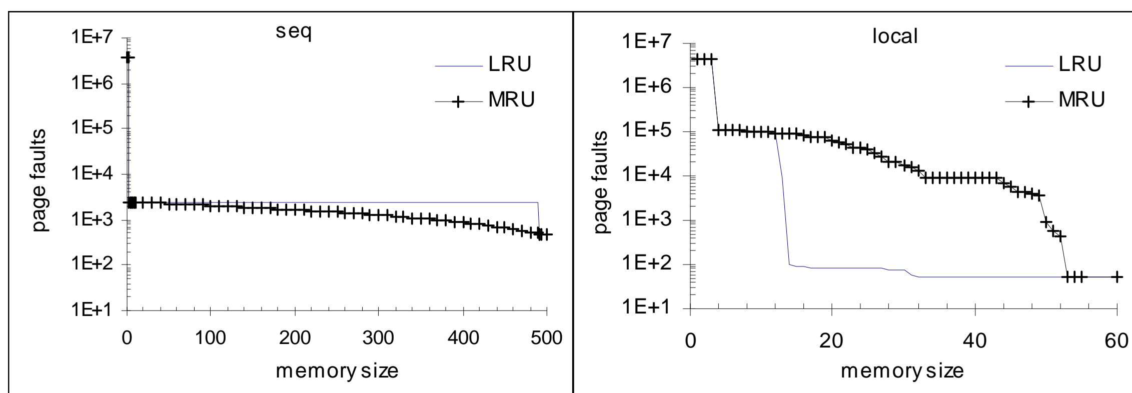

Figure 4 - Individual behavior of each program for LRU and MRU algorithms.

As each process is managed by a different algorithm, the memory pages must be partitioned among them. So, a memory partitioning strategy should be chosen. Due to the fact that a lot of proportions could be applied in the memory partitioning among the processes, we decided to study the effect of these several possibilities. Due to space problems and of visibility of the data in the graphs, we had to restrict the presented cases here. The cases are nominated "x/y" and they represent an allocation in that x% of the memory is reserved the program seq and y%, to the program local. The studied cases include 10/90, 30/70, 50/50, 70/30 and 90/10.

One possible heuristic for memory partitioning could be based on the size of the programs. Then, as the relation between their sizes is 54/492, the case `10/90` could be one partitioning proposal. Based on our following simulation results we can verify the validity of this heuristic.

5.4. Analysis of Partitioning Alternatives

The results obtained in our simulation studies are presented in figures 5 to 8. First we will present the global system performance in our analysis presenting the total number of page faults and the space-time product for the execution of both programs.

From figure 5, we can see that for memory sizes larger than 140 pages the best memory partitioning strategy is `90/10`, followed by the other cases, and the worst case is `10/90`. The number of page faults is about 4 times better. But for memory sizes smaller than 140 pages, the case `30/70` presents the smaller number of page faults.

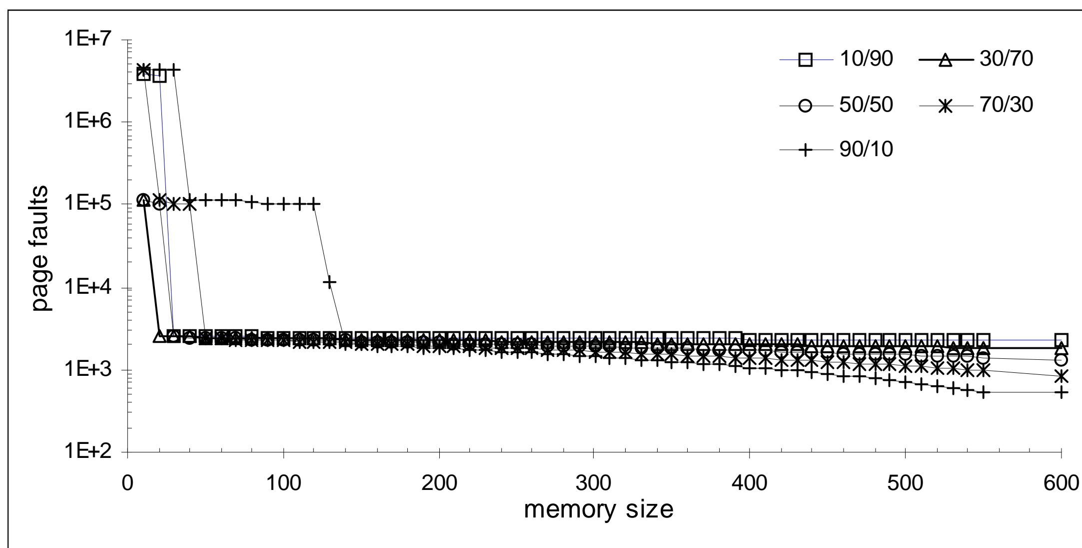

Figure 5 - Total number of page faults.

This result is very surprising, because the case `30/70` presents a much better performance than the case `90/10`: for memory sizes between 20 and 140 pages, the number of page faults for the case 30/70 is close to 2,500, and 100,000 for the case 90/10. This difference represents more than one order of magnitude.

For that cases with small memory sizes, it's possible to verify the high number of page faults for the `90/10` partitioning case. The reason for this is due to the internal behavior of the program `local` (figure 6). This graph shows the individual behavior of the program `local` in the multiprogramming execution.

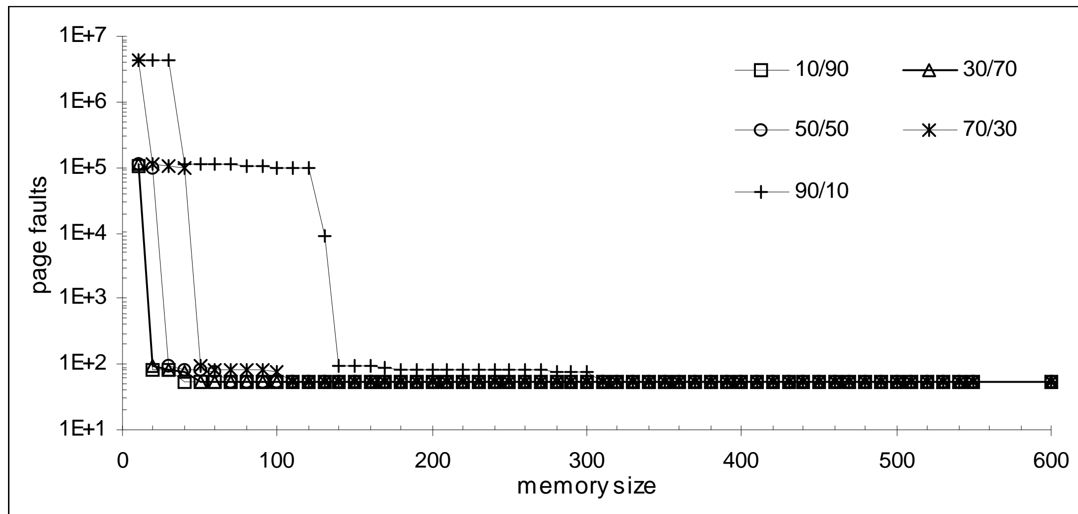

Figure 6 - Number of page faults for program `local`.

It has to be noted that only y% of the total memory size is allocated for the program local. So, in the case 90/10 only 10% of the total memory is reserved to `local` and 90% to `seq`. In this manner, for example, in the case which the memory size is 100 pages, only 10 pages is available to execute the program. This can explain the bad performance of this case in relation to the others.

The figure 7 shows the individual behavior for the program `seq`. All partitioning cases present a similar page fault behavior but the case 10/90 with a peak for low memories. Again, this can be explained by the fact that only 10% of memory are allocated for the program.

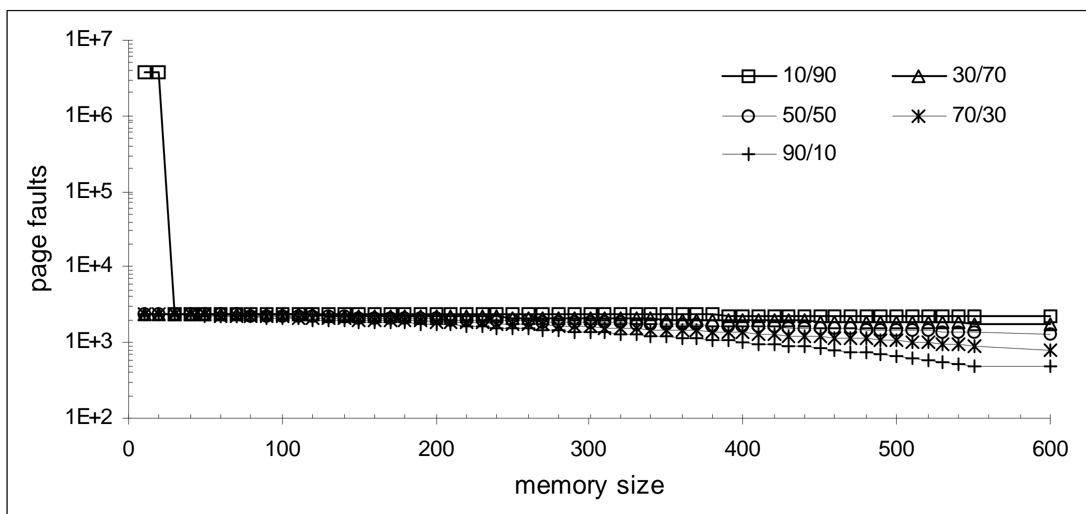

Figure 7 - Number of page faults for `seq` program.

Considering the memory occupancy, figure 8 show the space-time product of all partitioning cases. For memory sizes larger than 60 pages, the case `10/90` presents the lower memory usage. The case `90/10` exhibits a value 5 times larger than case `10/90`.

Unfortunately both cases present a bad behavior for small memory sizes with a peak value close to 10^{11} pages.cycles. Surprisingly, in this small memory interval, the best partitioning strategy in terms of space-time product is again the case `30/70`.

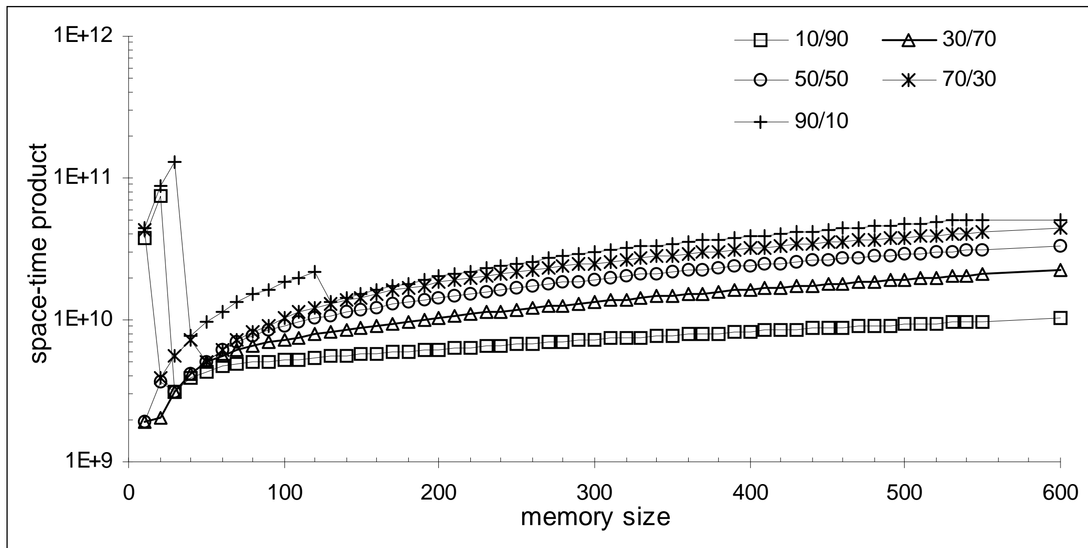

Figure 8 - Space-time product for local strategy.

Based on the previous results, we cannot decide for a single memory partitioning strategy for all possible memory sizes. Depending on the adopted approach, a special case can be selected: if we are interested on minimizing the number of page faults, the case `90/10` should be used for large memories.

5.5. Comparison with a Global Strategy

We compare here the adoption of the local memory management algorithms for each process against the traditional global approach. The question here is: "how better a local strategy could be compared to a global strategy adopted by conventional operating systems?"

The strategies studied are composed by the same local approaches studied in the previous section and global approaches using the LRU and MRU algorithms. The results obtained are presented in figures 9 and 10.

Related to the number of page faults, figure 9 show that the `global LRU` presents a good performance, comparable to the local alternatives. The case `90/10` has the best performance for memory sizes larger than 140 pages, but also presents the worst behavior for smaller memory sizes. For the memory sizes where case `90/10` has a worse performance than `global LRU`, the case `30/70` has a similar performance. Among all alternatives, the `global MRU` is the alternative with the worst performance.

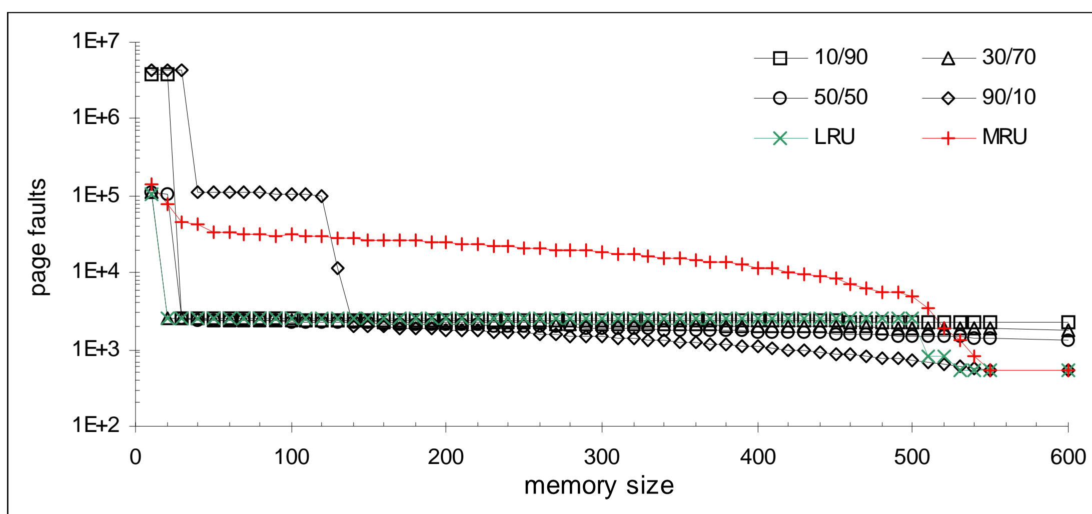

Figure 9 - Comparing local and global strategies: total number of page faults.

The results above show that a static local strategy cannot achieve a much better performance than a global one in all memory sizes. Some partitioning strategies have better behavior than others for some memory size interval. Concluding, a dynamic partitioning strategy could be considered for analysis.

The next graph (figure 10) shows the average memory occupation of the studied cases: in general, all curves have the same shape. It is possible to confirm that the case `10/90` presented the smaller memory occupancy, about 4 times smaller than the global strategies. Even the case `30/70` is 1,6 times better than `global LRU` and `MRU`. This fact can be explained by the fact that, in global approaches, all memory is used by the programs. In local approaches, as each program receive a quota of memory, some pages can stay free.

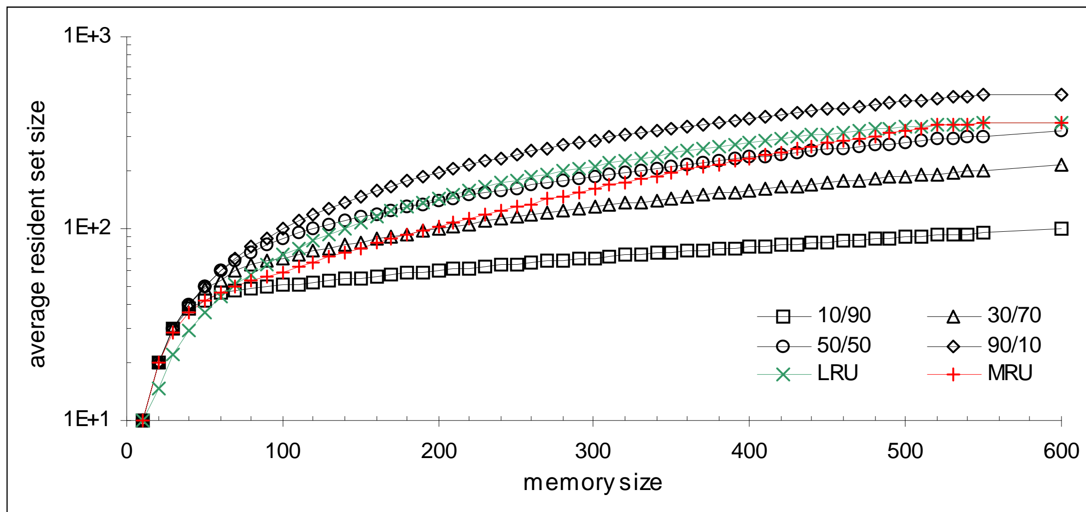

Figure 10 - Comparing local and global strategies: average resident set size.

Based on these results, we can say that for situations where the extreme memory partitioning cases present worst performance than the global strategy, some intermediate cases (`30/70` or `50/50`, for example) present good performance, close to the global LRU and MRU. We conclude that the adoption of a fixed partitioning strategy doesn't take to the best performance in all the situations.

6. IMPLEMENTATION ISSUES

In the previous sections we discussed our strategy for implementing a local policy for memory management. We now describe some implementation issues of this strategy in real operating systems like Linux [BECK98] or Windows NT [SOLO98]. The aspects presented here are not specific to any operating system, and would apply to other operating systems.

Modern operating systems have very few controls for memory allocation. The information provided for each active process include the total size of virtual address space, the current resident set size and a configurable limit to the total virtual memory a process can allocate. Unfortunately these information aren't sufficient to allow the implementation of our strategy.

A local adaptive memory management strategy requires some additional counters of pages: entitled, allowed and used. The kernel must keep track of the total number of pages that is allocated for a process through the *used* counter. The difference between *entitled* and *allowed* counters is that the first counter maintain the initial share of pages assigned to the process and the second counter hold the current limit of the process size. Usually both counters hold the same value. The allowed counter can be changed in case of page migration among processes in a heavy-load moment of the computer system.

These counters can be added on the process control block maintained by the operating system to hold all useful information of active processes. The process control block is named "task structure" in Linux and "executive process block" in Windows NT[2].

With the reference to the paging mechanism, all operating systems adopt a demand paging strategy with a separate page replacement support. When the number of free pages of the system reaches a lower limit, the kernel usually starts a procedure to free some pages of active processes. In order to decide which page to release, both Linux[3] and Windows NT[4] implement a variation of the LRU algorithm known as the clock algorithm.

As the current operating systems support only one page replacement algorithm, a particular problem arises to implement our strategy. How multiple replacement algorithms can be accommodated in the kernel? How the paging procedure can be modified?

The support of multiple algorithms isn't new in the Linux operating system. The current version 2.0 of the kernel provides two alternatives for scheduling a real time process (round robin and FIFO). The chosen algorithm is specified by a field in the task structure.

Our solution to this problem is the following: first, it's necessary to encapsulate each algorithm in a separate kernel object in order to allow an independent memory management of each process. Second, a new field is added to the process control block to hold the replacement algorithm selected to the process. Third, the points in the operating system that is related to the paging procedure must be modified to accommodate a parametrized execution according to the process algorithm. Forth, an independent data structure (queue, stack, etc) is necessary to hold the pages of each process. This data structure is used to implement the selected policy.

Our first prototype is being implemented in a multiprocessor platform based on a dual Pentium II computer with RedHat Linux 5.1 (kernel 2.0.33).

7. Conclusion and Future Work

This work presented a study of a novel memory management strategy in modern high performance computers. In our proposal, each program can be managed by a different algorithm and the system memory is partitioned among the programs in execution.

The studies and evaluations showed good results and they validated our focus. In a general way, we verified that the directives with the specification of the management algorithm has great influence in the global performance of the system (space-time product) as in the individual performance of the applications (number of page faults).

In spite of the preliminary results that demonstrate a small advantage of the adoption of a local strategy in multiprogramming environments, several subjects stay open and they deserve a deepened study:

- *memory partitioning among the processes in execution*: strategies of dynamic partitioning should be studied for the obtaining of better performance. The study described here adopted a static approach in that each process received a fixed percentage of the available memory;

[2] For detailed description of the process control block of each operating system see [BECK98] and [SOLO98].

[3] In fact, Linux implements different policies for swapping pages out. To shrink the page and buffer cache, the clock algorithm is adopted. However, the swap daemon uses an aging procedure to decide which page to discard from memory.

[4] As a matter of fact, this algorithm is used only in the uniprocessor version of the Windows NT. On multiprocessor x86 systems and on all Alpha systems, Windows NT implements a variation of a local FIFO replacement algorithm.

- *choose of the best algorithm for each process*: tools to aid the system in the choice of the best management algorithm should be developed. Today we adopted the simulation results made under the Elephantus environment;
- *possibility to specify an algorithm for each section of the program*: each program presents several processing phases with different memory access patterns. It is important a study in that the adoption of different management algorithms is analyzed for each processing phase.

Other future works include the extension of the strategy *CAPR* to be applied in another situations, for example, in the scheduling of parallel programs and in the data distribution across the memory modules of a distributed system.

ACKNOWLEDGMENTS

The authors would like to thank all the members of our research groups for their valiant work. We are also thankful to all that revised the preliminary versions of this work, for their critics and suggestions of improvements.

REFERENCES

[BECK98] BECK, M. et all. **Linux kernel internals**. 2nd edition, Addison-Wesley, 1998.

[CARR94] CARR, S.; McKINLEY, K. S.; TSENG, C.-W. **Compiler optimizations for improving data locality**. In: Proceedings of the 6th Int. Conf. on Architectural Support for Programming Languages and Operating Systems, San Jose, CA. p.252-262. October, 1994.

[CHAP97] CHAPIN, J. **A fresh look at memory hierarchy management**. In: Proceedings of the HOTOS VI. 1997.

[CHER93] CHERITON, D. R. and HARTY, K. **A market approach to operating system memory allocation**. Stanford University, Computer Science Department. 1993. URL page: `http://www-dsg.stanford.edu/Publications.html`

[DENN80] DENNING, P. J. Working sets: past and present. **IEEE Transactions on Software Engineering**, SE-6:1, p.64-84, January 1980.

[EDWA96] EDWARDS, J. K. & CAO, P. **User-oriented resource scheduling in UNIX**. Technical Report CS-TR-96-1318, Department of Computer Science, University of Wisconsin, 1996.

[FRAN78] FRANKLIN, M. A. et alii. Anomalies with variable partition paging algorithms. **Communications of the ACM**, 21:3, p.232-236, 1978.

[GLAS97] GLASS, G. W. **Adaptive page replacement**. MSc Thesis. University of Wisconsin-Madison. 1997.

[HEXS94] HEXSEL, R. A. **A quantitative performance evaluation of SCI memory hierarchies**. PhD Thesis. Department of Computer Science, The University of Edinburgh. 1994.

[KRIS95] KRISHNAN, P. **Online prediction algorithms for databases and operating systems**. PhD Thesis. Department of Computer Science, Brown University. 1995.

[MALK86] MALKAWI, M. I. **Compiler directed memory management for numerical programs**. Ph.D. Thesis. Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign. 1986.

[MIDO95] MIDORIKAWA, E. T. & ZUFFO, J. A. **A technique for compiler-aided memory management**. In: Proceedings of the 21st CLEI, Canela, RS, Brazil. August 1995. (in portuguese)

[MIDO97a] MIDORIKAWA, E. T. et all. **Communion: a new strategy for memory management in high-performance computer systems**. In: Proceedings of CACIC'97, La Plata, Argentina. October 1997.

[MIDO97b] MIDORIKAWA, E. T. **A new strategy for memory management to high performance computer systems**. PhD Thesis. Polytechnic School, University of São Paulo, Brazil. 1997. (in portuguese)

[MIDO98] MIDORIKAWA, E. T. **Data locality optimization in real application and strategies for dynamic memory allocation**. In: Proceedings of the 10th Brazilian Symposium on Computer Architecture and High Performance Processing, Búzios, RJ, Brazil. September 1998. (in portuguese)

[SAUL96] SAULSBURY, A. et alii. **Missing the memory wall: the case for processor/memory integration.** In: Proceedings of 23rd International Symposium on Computer Architecture, 1996. May 1996.

[SOLO98] SOLOMON, D. A. **Inside Windows NT**. 2nd edition, Microsoft Press, 1998.

[STAL98] STALLINGS, W. **Operating systems: internals and design principles**. 3rd edition, Prentice-Hall, 1998.

[STOI95] STOICA, I. and ABDEL-WAHAB, H. **A new approach to implement proportional-share resource allocation**. Technical Report 95-05, Old Domonion University, Department of Computer Science. April 1995.

[VAHA96] VAHALA, U. **Unix Internals: the new frontiers**. Prentice-Hall, 1996.

[VERG98] VERGHESE, B. **Resource management issues for shared-memory multiprocessors**. PhD Thesis. Department of Electrical Engineering and Computer Science, Stanford University. 1998.

[WALD95] WALDSPURGER, C. A. **Lottery and stride scheduling: flexible proportional-share resource management**. PhD Thesis. Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology. 1995.

[WOLF96] WOLFE, M. **High performance compilers for parallel computing**. Addison-Wesley, 1996.

[WULF95] WULF, W. A . & McKEE, S. A. *Hitting the memory wall: implications of the obvious*. **Computer Architecture News**, 23:1, p.20-24. March 1995.