COMMUNION: A NEW STRATEGY FOR MEMORY MANAGEMENT IN HIGH-PERFORMANCE COMPUTER SYSTEMS ⋆

Edson Toshimi Midorikawa [1]
Liria Matsumoto Sato [2]
João Antônio Zuffo [1]

[1] Laboratory of Integrated Systems
Department of Electronic Engineering
Polytechnic School, University of São Paulo
São Paulo, SP, Brazil 05508-900
{emidorik,jazuffo}@lsi.usp.br

[2] Department of Computer Engineering and Digital Systems
Polytechnic School, University of São Paulo
São Paulo, SP, Brazil 05508-900
liria@pcs.usp.br

ABSTRACT

Modern computers present a big gap between peak performance and sustained performance. There are many reasons for this situation, but mainly involving an inefficient usage of computational resources. Nowadays the memory system is the most critical component because of its growing inability to keep up with the processor requests. Technological trends have produced a large and growing gap between CPU speeds and DRAM speeds. Much research has focused this memory system problem, including program optimizing techniques, data locality enhancement, hardware and software prefetching, decoupled architectures, multithreading, speculative loads and execution. These techniques have got a relative success, but they focus only one component in the hardware or software systems.

We present here a new strategy for memory management in high-performance computer systems, named COMMUNION. The basic idea behind this strategy is "cooperation". We introduce some interaction possibilities among system programs that are responsible to generate and execute application programs. So, we investigate two specific interactions: between the compiler and the operating system, and among the compiling system components. The experimental results show that it's possible to get improvements of about 10 times in execution time, and about 5 times in memory demand, enhancing the interaction between the compiling system components. In the interaction between compiler and operating system, named Compiler-Aided Page Replacement (CAPR), we achieved a reduction of about 10% in space-time product, with an increase of only 0.5% in the total execution time. All these results show that it's possible to manage main memory with a better efficiency than current systems.

1. Introduction

Studies show that programs usually present an average performance that is only up to 15% of the peak performance in vector supercomputers, like NEC SX-3R and Cray C90. The main reason is the inefficient use of computational resources like vector registers and instructions, vector pipelines and interleaved memory banks.

Modern parallel computers, like NEC SX-4 and Cray T3E have the same problem. But the reasons are completely different. Such class of machines adopts a large number of processors and a complex hierarchy of main memory. These characteristics contribute to the high complexity for programming these machines. In order to get high performance programs, the programmer has to deal with two aspects: efficiently exploiting of parallelism and adequate use of the complex memory hierarchy. The first aspect has been studied for a long time [POLY88]. On the other hand, only recently researchers started to analyze the second aspect.

⋆ This research was supported in part by Finep and RHAE/CNPq.

The impressive progress in arithmetic performance achieved by the latest generation of microprocessors tends to widen the gap between the CPU speed and the memory bandwidth. This issue, critical for monoprocessors, is worse for shared-memory multiprocessors where memory contention can severely degrade main memory performance. To overcome this problem, one of the most widely used techniques consists in designing hierarchically organized memory systems with several levels.

This work presents some aspects that may be considered when developing a program in modern high-performance computers. Here we concentrate on shared memory multiprocessors, although the results can be extended to distributed memory multiprocessors. The structure of this paper is the following. We first present some characteristics of modern high-performance computers in section 2. Then we analyze the memory management techniques in section 3, describing our strategies. The next section presents the traditional phase-based approach for memory management. The section 5 introduces our proposal for future generation memory management. In sections 6 and 7 we present some experimental results in the CAPR strategy and the prototype system in development, and in the locality optimization techniques, respectively. Finally, we present our conclusions in section 8.

2. High-Performance Computing

Shared-memory multiprocessor machines[1] can be classified as having uniform memory access (UMA) or nonuniform memory access (NUMA). In UMA machines, the access time to all memory locations is constant. For NUMA machines, each processor has a preferred section of the address space, which it can access faster than the rest of the address space. Usually, this preferred section represents memory that is physically local to each processor or to a cluster of processors.

A typical NUMA organization is shown in figure 1. The memory modules are associated with processors, which can access their local modules much faster than remote modules. As in the case of UMA machines, processors can have private caches, adding another level to the hierarchy of memory accesses. For example, SGI Power Challenge has a 2-level cache memory for each processor.

Modern high-performance computing systems, like NEC SX-4, have a NUMA organization with a multi-level memory hierarchy implemented by hardware and software. Usually they provide a shared memory programming model for their users. Such model is implemented by some form of distributed shared memory (DSM) system. Nowadays parallel systems with hundreds or thousands of processors are becoming available. For example, the largest model of NEC SX-4 has up to 512 processors.

Such shared memory model can also be used in a NOW (Network Of Workstations) running UNIX-like operating systems by using DSM library packages, like Treadmarks, Munin and Pulsar [MARI96].

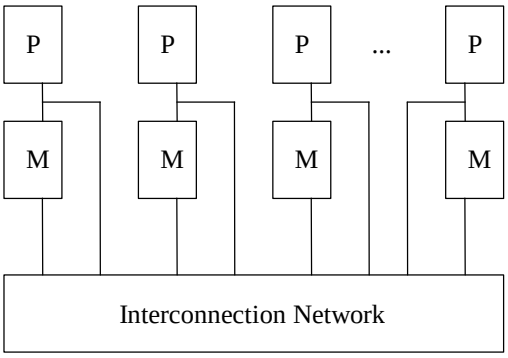

Figure 1 - Typical NUMA machine organization.

3. Virtual Memory and Memory Management Techniques

Today high-performance computers support some form of virtual memory for their users. Although these machines have hundreds of megabytes of main memory, some application programs require bigger memory in order to run. This tendency has been observed since the early sixties. Paraphrasing Parkinson's law, "*Programs expand to fill the memory available to hold then*." [TANE97]

[1] For additional information on high-performance computer systems see [HWAN93].

Memory is an important resource that must be carefully managed. Research in virtual memory has been conducted in monoprocessor systems since the sixties, but only recently researchers had applied studies to parallel computers.

Some characteristics are very important when we want to study the behavior of programs: their size in pages, use of dynamic memory allocation, locality of references, page management algorithms adopted by the operating system, memory access patterns and some hardware parameters.[2]

Modern computers characteristics, such as 64-bit address space, gigabyte main memory and multi-level main memory hierarchies, impose research studies in new strategies to manage memory efficiently. This research has been conducted at Polytechnic School of University of São Paulo.

Our research is divided in two main areas: first, some new algorithms to page replacement are being proposed. The main contribution in this area is the integration of the operating system and the compiler in managing main memory pages. This integration provides better selection of the page being removed. The second area concentrates in techniques to modify the application programs in order to improve their locality of reference. This part involves the implementation of some program transformations[3] into the compiling system, in particular, to the optimization phase. In this way, we can say that our primary goal is to optimize the use of modern computers memory hierarchy so that we can get programs with higher performance.

In the next sections we introduce each of these research areas. First we describe our approach and then we present an evaluation study conducted in order to verify its performance. Although our research is in an initial phase, the results presented here show that it is possible to get higher performance by applying our approach.

4. Phases of Memory Management

The memory management of a computer system is distributed across many different overlapping phases (figure 2). The programmer is responsible for the first step when he organize the application data in program variables (symbolic references). Next there is a translation of the program source files written in a high-level programming language into a machine-specific code. In these object files all symbolic instructions, operands, and addresses are converted into numerical values. This step is conducted by the compiler. After that, the linker combines several object files to make a binary file, basically resolving external references. Later on the executable code is generated by the loader by relocating the memory references. Finally the program dynamic memory requests are served by the operating system and run-time system during its execution. The operating system manages the allocation of the system main memory while the run-time system is responsible for controlling the program resources.

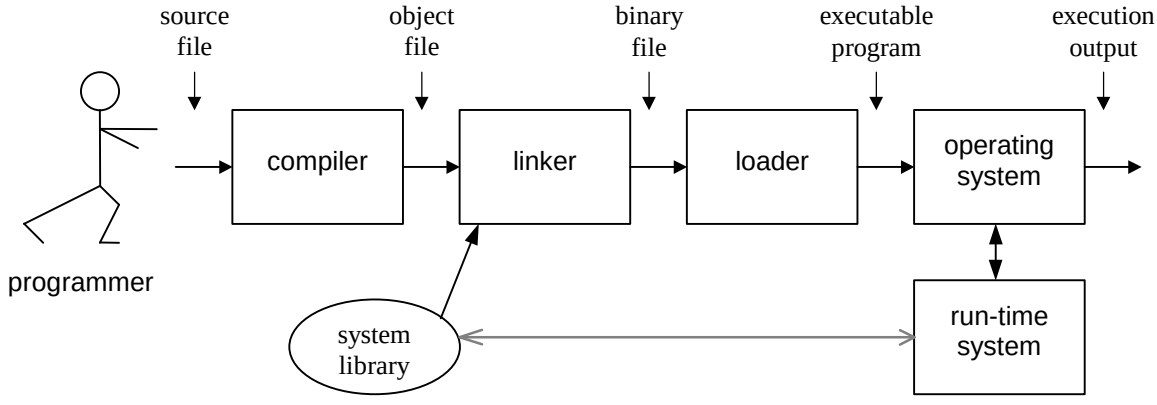

Figure 2 - Phases of memory management.

The current approach of system optimizing techniques is based on the following strategies:

- given a program source file, its memory access pattern is analyzed and, based on the results of this analysis, the variable addresses is reorganized (relocation) and the structure of vectors and arrays is

[2] Parameters like cache line size, page size, data cache size, transfer rate between memory and disk.
[3] These transformations can be divided in two classes: control transformations and data transformations. For further information see [TAKA95].

modified by some data transformations. This strategy is adopted by current restructuring compilers being developed for distributed memory multiprocessors [KREM95];

- given the memory access pattern behavior of a program execution, the system can change some operating system parameters in order to improve performance. Examples of Solaris 2.x control parameters are the number of pages of usable physical memory, maximum number of page out I/O operations per second, number of pages scanned per second by the paging algorithm, and the maximum age of a modified file system page in memory [COCK95].

The first strategy has the advantage of being implemented by automatic restructuring tools, and so, users don't need to have specialized knowledge about memory optimization techniques. The main drawback is that these automatic tools don't know the semantic of the program, and in this way it's not possible for them to suggest the best data organization. That means this strategy only works with the program variables in the way they were defined by the programmer. There is no interaction with the programmer.

The second strategy assumes an unchangeable application program and tries to adjust internal parameters of the operating system to tune the system for its execution. The main drawback of this strategy comes from the fact that the optimum performance is not achievable due to the unchangeability of program memory access pattern. Furthermore, a small change in a global operating system parameter may strongly affect the execution of the other programs in the system.

5. *Communion*: A New Strategy for Memory Management

Current systems use the sequential approach of separate phases for memory management of application programs. In this traditional approach described in section 4, each system component is responsible for just one task and executes it as best as possible. 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).

We are investigating the viability of such interactive approach among the system programs. Under our proposal, named COMMUNION, the system programs "work together" in order to achieve enhanced performance. The figure 3 shows a diagram of the Communion approach.

It can be noted from figure 3 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.

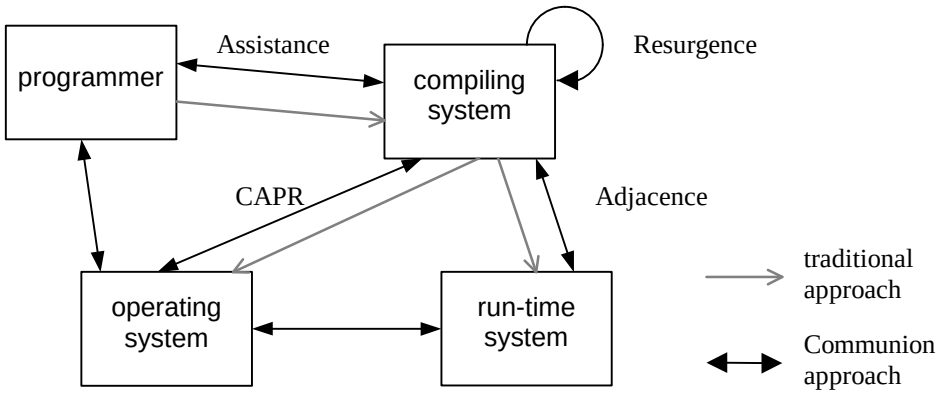

Figure 3 - Interaction among system components.

In the next sections we investigate two specific interactions[4]: between the compiler and the operating system (named CAPR), and among the compiling system components (named Resurgence). Our preliminary studies concentrate in these areas.

6. Compiler-Aided Page Replacement

CAPR is the first developed strategy of COMMUNION. 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. Later we present a brief evaluation.

6.1 - 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).
- PFF ("*page-fault frequency*"): identify localities transitions measuring the page fault rate and discard the pages of the old locality based in a time window parameter.
- WS ("*working set*"): maintain in memory only those pages that were accessed during a time window (in past).

[4] Some other interactions are discussed in [MIDO98].

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.

6.2 - 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 such directives are:

- locality change
- change in memory requirement
- change in algorithm specific parameters
- change the management algorithm
- lock / unlock pages in memory

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
- selected program sections 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 4 illustrates the interaction between the compiler and the operating system.

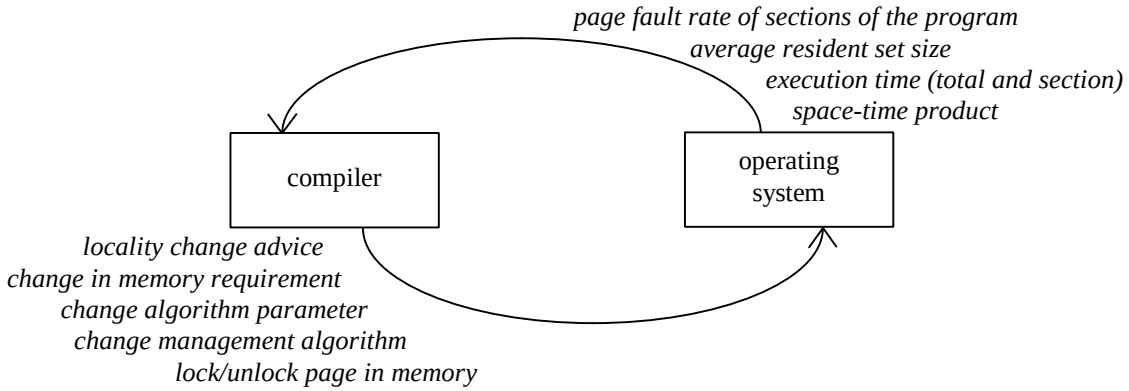

Figure 4 - Interaction between compiler and operating system under CAPR strategy.

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 algorithms with

additional functionalities, 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.

6.3 - Evaluation

In order to evaluate our strategy, we had implemented a memory systems simulator. This simulator uses the execution-driven simulation technique, so it's not necessary to use the traditional execution traces. It interprets the processor instructions and analyzes all memory references. The current version implements LRU, FIFO, LFU, Clock, Second Chance, WS and PFF algorithms. The simulation output is composed by total simulation time, number of memory accesses, number of page faults, and execution space time product. Our evaluation studies were developed in a Silicon Graphics Power Series 4D/480 VGX computer, which has eight processors and 256 Mbytes of memory. The internal structure of the simulator is presented in figure 5.

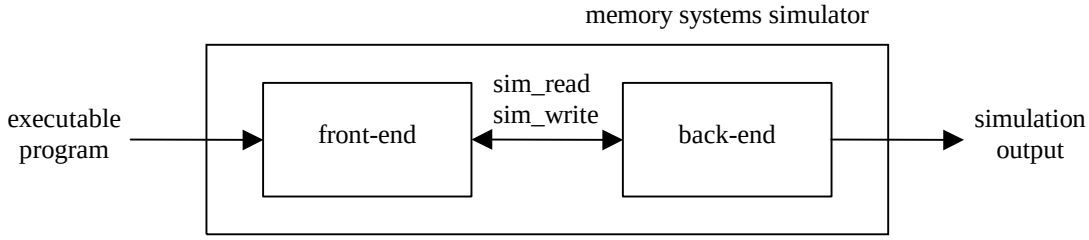

Figure 5 - Memory systems simulator internal structure.

The simulator is based on the MINT package, developed at Rochester University [VEEN94]. MINT is a software package designed to ease the process of constructing event-driven memory hierarchy simulators for multiprocessors. It provides a set of simulated processors that run standard UNIX executable files compiled for a MIPS R3000 based multiprocessor. The input to MINT is a statically-linked Irix executable file and it is not necessary to instrument application code for simulation or to recompile. The internal structure of a simulator is composed by two main parts: a memory reference generator (the "front-end"), and a target system simulator (the "back-end"). MINT communicates memory references to the simulator back-end through special event routines (`sim_read()` and `sim_write()`). The figure 6 shows the graphical interface of the memory systems simulator.

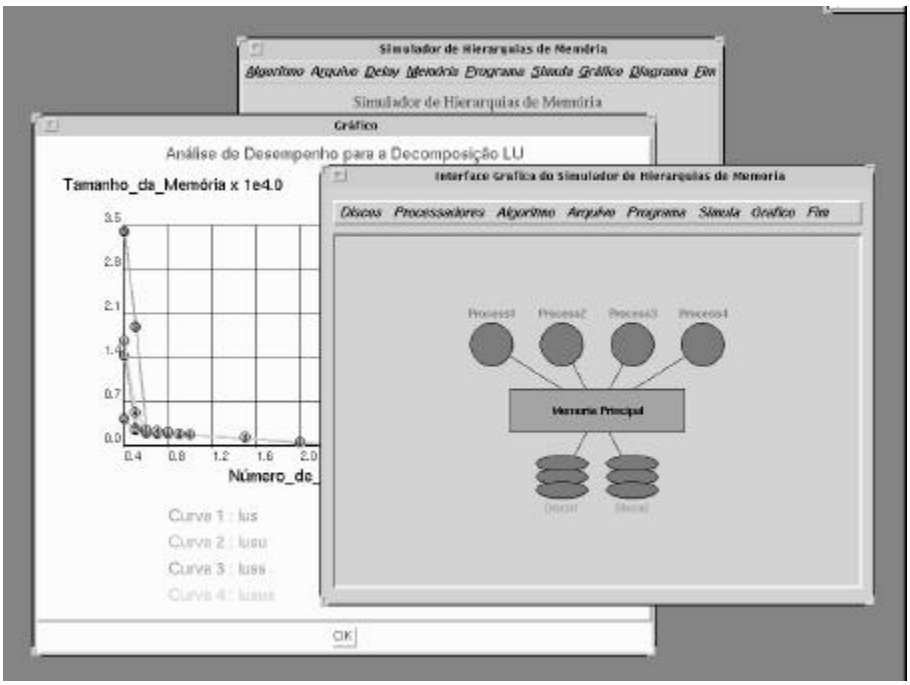

Figure 6 - Graphical interface of the memory system simulator.

In order to confirm the effectiveness of CAPR strategy, we conducted a study of the execution behavior of an example program under LRU and FIFO algorithms. The figure 7 shows the number of page

faults in function of the memory size. This example program can be described as being composed by two processing phases, each of these phases with distinct localities and different sizes. The first phase has 33 pages of size and the second has 21 pages.

For comparing the performance of CAPR with LRU and FIFO, we chose the space-time product metric. We can say that space-time product is a number that represents the memory demand of one program over the entire execution time.

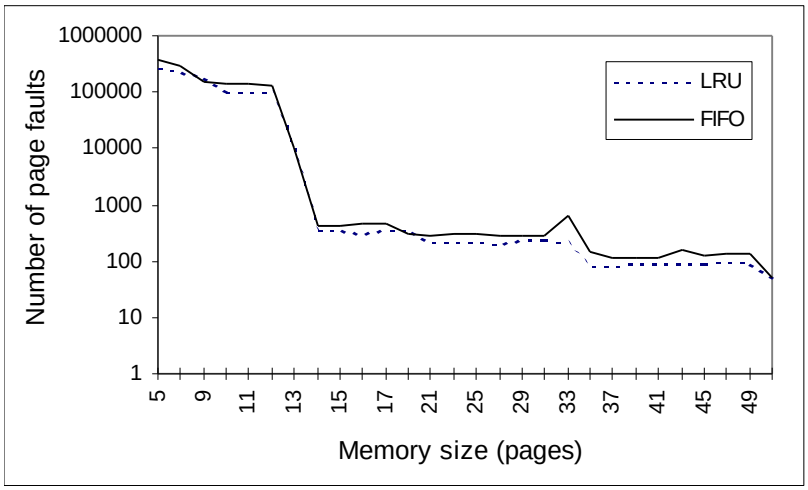

Figure 7 - Number of page-faults in function of memory size for the example program study.

Due to the chosen example program, we decide to use a CAPR directive that informs a change in localities, determining a new memory requirement. We use the "$N_1 \Rightarrow N_2$" notation to represent the CAPR directive used to inform the operating system a change from N_1-page locality to the new N_2-page locality during the execution of the program. The table 1 below shows a comparison between LRU and CAPR/LRU.

Table 1 - Comparison between LRU and CAPR/LRU.

Algorithm	Memory size (pages)	Space-time product (pages x cycles)	Total simulation time (cycles)
LRU	11	2.028.286.920	184.389.720
	12	2.191.388.640	182.615.720
	13	1.211.778.360	93.213.720
	14	1.175.646.080	83.974.720
	15	1.259.560.800	83.970.720
	51	4.266.237.720	83.651.720
CAPR/LRU	$14 \Rightarrow 9$	1.181.216.760	175.539.550
	$14 \Rightarrow 10$	1.102.337.380	91.165.638
	$14 \Rightarrow 11$	1.067.094.768	84.007.638
	$14 \Rightarrow 12$	1.103.156.156	83.984.638

Denning affirms that, in a multitasking environment, the optimum multiprogramming level is associated with running each process at its minimum space-time product [DENN80]. Analyzing the table, we confirm the effectiveness of CAPR. The best resident set size for LRU algorithm is 14 pages. And for CAPR/LRU, the case $14 \Rightarrow 11$ presents the smaller space-time product. Comparing both situations, there was a reduction of 9.2% in space-time product and an increase of only 0.04% in execution time. Considering the best allocation case in terms of execution time (51 pages allocated to the program), with our strategy, there was a reduction of 75% in space-time product, with only an increase of 0.43% in the simulation time.

Now let's consider the data relating to the FIFO algorithm (table 2). The best cases now are allocating 14 pages for the traditional LRU algorithm and the case $14 \Rightarrow 10$ for CAPR/FIFO.

Table 2 - Comparison between FIFO and CAPR/FIFO.

Algorithm	Memory size (pages)	Space-time product (pages x cycles)	Total simulation time (cycles)
FIFO	11	2.438.025.920	221.628.720
	12	2.598.788.640	216.565.720
	13	1.222.048.360	94.003.720
	14	1.176.108.080	84.007.720
	15	1.260.565.800	84.037.720
	51	4.266.237.720	83.651.720
CAPR/FIFO	$14 \Rightarrow 9$	1.048.731.760	90.009.550
	$14 \Rightarrow 10$	1.031.863.380	84.094.638
	$14 \Rightarrow 11$	1.068.019.768	84.075.638

The results obtained with FIFO algorithm are a reduction of 12.3% in space-time product and an increase of only 0.1% in execution time. Comparing with the 51-page case, CAPR/FIFO provided a reduction of 75.8% in space-time product and a small increase of 0.53% in execution time.

The above results verify that CAPR can not only be used to improve performance of one single program but also the performance of the entire system. So we conclude that CAPR can achieve additional performance improvements that can't be obtained by using traditional techniques.

7. Resurgence: Locality of Reference Optimization

The hierarchical organization of main memory of modern high-performance computers can be explored by programs with improved data locality. This can be done by optimizing its temporal locality (a same memory address referenced several times) and its spatial locality (references to consecutive memory addresses) with the application of program transformations.

The technology of program transformations was developed by researchers of automatic parallelizing compilers. The basic concepts are the data dependence and program transformations [POLY88][WOLF96]. Examples of transformations for optimization of locality of reference are scalar replacement, unroll-and-jam, loop interchange, tiling, memory copying, and data alignment.[5]

Usually, the main objective of compiler optimizations is to reduce the program execution time [AHO86]. So traditional compiler always try to get a optimized program with a total execution time that is shorter than the one of the original program. Resurgence proposes changing our attention to the program memory demand. We investigate how to obtain programs with smaller memory demands.

Our research in this area can be divided in two objectives: first, analyze the effect of some transformations on the memory access behavior (locality of reference). Two classes of transformations are being studied, namely control transformations and data transformations. The second objective is the implementation of a prototype system that automatically restructures programs that improve data locality.

7.1 - Transformations for optimizing data locality

One of the problems of programs concerning locality of reference is the memory access pattern. Given a 2-dimensional matrix, for example, the elements can be accessed by line, by column or by diagonal. Depending on the way these elements are organized in memory one access pattern can result in a poor locality.

Consider one program written in C language. The elements of a matrix are organized by line[6]. If one program access this matrix by column, this will result in a bad access pattern. There are two ways to deal with this problem: first, change the way that the program accesses the elements. This can be done by the application of control program transformations. Loop interchange is an example of such a transformation. The basic idea is to change the order of two loops in a loop nest. The other alternative is to reorganize the elements in memory, so that they are stored in the same way they are accessed. This strategy involves the application of data

[5]For further information see [POLY88][WOLF96].
[6]This means that all elements of the same line are organized contiguously in memory.

program transformations. One example of data transformation is data copying. The main objective is to copy non-contiguous data in a contiguous area with the purpose of improve data locality.

7.1.1. Influence on execution time of sequential programs

We have applied two transformations in a LU decomposition program and in a matrix multiplication program in order to study their profitability. The experimental results are shown in table 3 and table 4.

Table 3 - Experimental results with LU decomposition program.

program	original LU	LU with loop interchange	LU with copying
execution time (sec)	198.0	107.3	108.3
speed-up	1	1.85	1.83

It can be observed that both transformations produced good results with speed-ups of 1.85 and 1.83. We conclude that the application of transformations can improve data locality and so improve performance of modern computers. The table 3 shows speed-ups of 2.43 and 2.37 for the matrix multiplication program, indicating the validity of our approach. Similar results were obtained with other programs.

Table 4 - Experimental results with matrix multiplication program.

program	original MM	MM with loop interchange	MM with copying
execution time (sec)	414.4	170.5	174.8
speed-up	1	2.43	2.37

Once we have verified the effect of transformations in the execution time, we conducted an investigation on the memory demand of programs. The purpose was to understand how these transformations affect the resident set size during the execution of programs. The set of transformations is listed in table 5 below.

Table 5 - Set of transformations.

suffix	transformation
li	loop interchange
dt	data transformation
sr	scalar replacement
uj	unroll-and-jam
t	tiling

7.1.2. Influence on execution time of parallel programs

We have also conducted the same study in parallel versions of both programs. Due to space problem we will examine in detail the results for the matrix multiplication program. The initial version of the program (`mm`) adopts the canonical implementation based on the calculation of inner products:

```
for (i=0; i<N; i++)   /* parallelized loop */
    for (j=0; j<N; j++)
        for (k=0; k<N; k++)
            c[i][j] += a[i][k] * b[k][j];
```

Observing the memory access pattern of matrix b, we conclude it's accessed by column. This problem is solved applying loop interchange (`mm-li`). In order to enhance its code we have applied the scalar replacement (`mm-sr`) and unroll-and-jam (`mm-uj`) program transformations. The combined effect of both transformations is verified in `mm-li-uj-sr`. Finally, in order to get better locality, we have applied the tiling transformation (`mm-li-uj-sr-t`)

The execution time of each program version is presented in table 6 and figure 8. We have executed these programs in the Silicon 4D/480 parallel machine.

Table 6 - Execution time of parallel matrix multiplication program.

number of processors	mm	mm-li	mm-li-sr	mm-li-uj	mm-li-uj-sr	mm-li-uj-sr-t
1	480.0	171.6	140.3	133	117.5	79.3
2	245.0	86.2	71.1	68.1	60.3	40.2
3	178.0	58.2	48.3	46.2	41.2	27.7
4	161.9	44.8	37.4	35.4	31.2	20.8
5	154.5	36.3	30.4	29.0	26.1	17.2
6	140.6	31.6	26.0	25.8	23.6	14.4
7	124.5	27.3	22.0	22.1	20.6	12.4
8	111.9	24.1	19.7	19.8	18.1	11.2

Observing the results it's possible to verify the good performance enhancement comparing the original version (`mm`) and the last one (`mm-li-uj-sr-t`). This improvement gets better as the number of processors increases. For one processor, the performance is 6 times better, and it reaches 10 times for the case of 8 processors. This result verifies the syntonization among program transformations, data locality and parallelism.

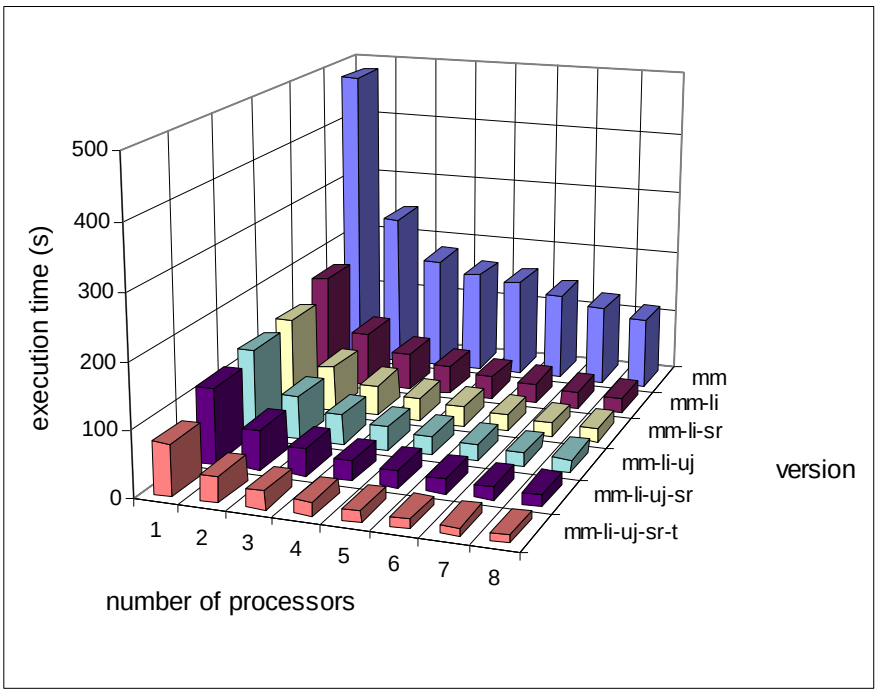

Figure 8 - Execution times for parallel matrix multiplication.

The effect of these transformations can also be studied by the speed-up graph (figure 9). Analyzing each curve, it's possible to enhance the memory access behavior. The maximum speed-up for mm version is 4.29 to 8 processors. For the other versions, the average speed-up is 7 for the same number of processors.

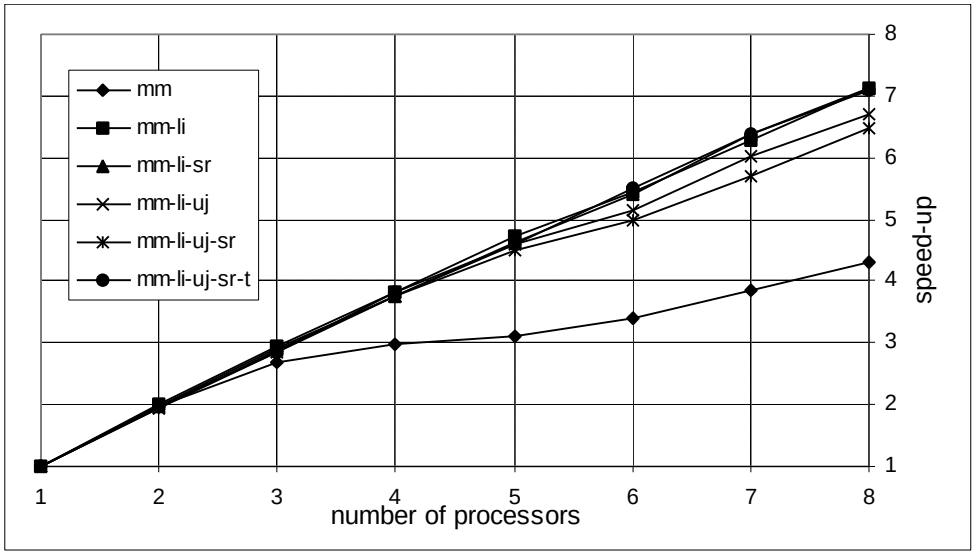

Figure 9 - Speed-up for parallel matrix multiplication.

The same behavior is observable from the efficiency analysis (figure 10). The efficiency of `mm` version diminishes up to 54% to 8 processors. The transformed versions exhibit much better performance (the worst version is `mm-li-uj-sr` that presents an efficiency of 81%).

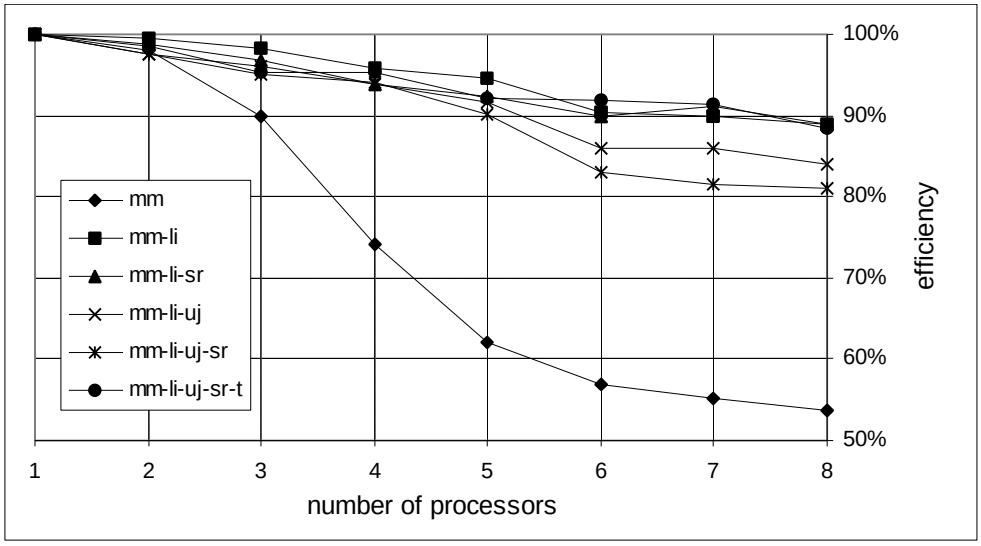

Figure 10 - Efficiency for parallel matrix multiplication.

7.1.3. Influence on memory demand

The first objective of program transformations is the reduction of the execution time. This first result was the only one studied for many years. Recently, some research groups are analyzing the use of such program transformations to enhance the access pattern of complex memory hierarchies by studying their influence on memory demand. We analyzed both programs (`lu` and `mm`) in terms of memory demand during their execution period.

In order to evaluate the memory demand of a program, we have used the process accounting subsystem of UNIX systems (`pacct`). This subsystem collects some information during the program execution that can be accessed later. One of these informations is the average resident set size (in Kbytes/minute).

The experimental studies were conducted in three different platforms presented in table 7. Each machine is different in some aspects, for example, processor speed, bus width, main memory organization, cache memory associativity, internal memory management algorithms.

Table 7 - Description of utilized machines.

Machine	Processor	Main Memory	Cache Memory	Operating System
SGI Power 4D/480 VGX	MIPS R3000A (8)	256 Mbytes	64 Kbytes Icache 64 Kbytes Dcache 1 Mbyte Dcache	IRIX 5.3
Bull ISM Server 490	Power PC 601	128 Mbytes	32 Kbytes I&D Cache	AIX 3.2
Sun SPARCStation 20 MP 712	SuperSPARC II (2)	128 Mbytes	20 Kbytes Icache 16 Kbytes Dcache 1 Mbyte Dcache	SunOS 5.5 (Solaris 2.5)

The figure 11 shows the results obtained for the LU decomposition program. In general, both transformations used in this study resulted in an enhanced version of the original program. The improvement varies from 1.2 for the scalar replacement (`lu-sr`) in SGI up to 1.97 for combined transformation (`lu-sr-uj`) in Bull machine. Then we conclude that it's possible to get a new version of the LU decomposition program that requires only half of memory during its execution.

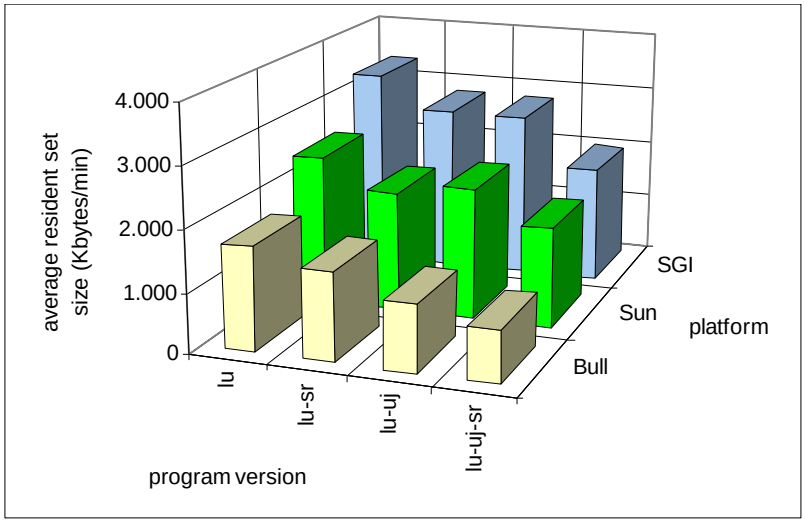

Figure 11 - Effect in memory demand for LU decomposition program.

The same study was conducted for a matrix multiplication program, and the results are presented in figure 12. Here we got a better improvement: the combined application of transformations resulted in a memory demand that is 4 (Bull and Sun) to 5 (SGI) times smaller than the original program.

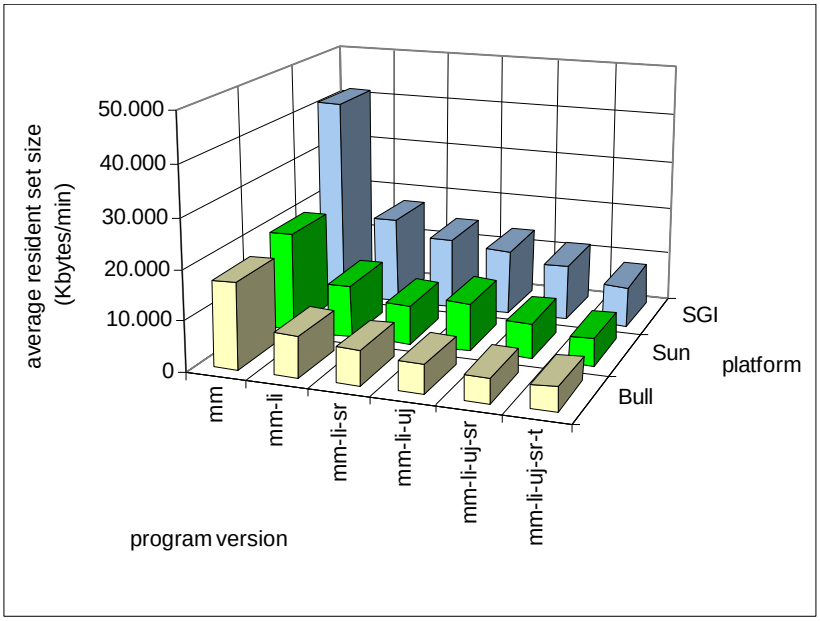

Figure 12 - Effect in memory demand for matrix multiplication program.

All these results show that program transformation can be used not only to reduce the execution time of a program, but also to diminish its memory demand. So it is possible to enhance system performance by allowing the increase of the multiprogramming level.

This study shows that one reason for the effectiveness of program transformation related to diminishing programs execution times is associated to their effect in memory demand.

7.2 - Prototype system

Considering the good results of previous work, we decided to implement a system that automatically analyzes a program and applies some transformation in order to improve locality of reference. Our system has one additional goal that is to parallelize the program whenever it is possible.

The set of transformations implemented comprises the so-called *unimodular transformations* [BAN88]. The unimodular transformations are a unified theory of a class of loop transformations. It is based on the theory of unimodular matrixes, so that the analysis of its application can be easily studied.

In order to decide which transformation we apply, it was chosen a metric for locality of reference, the reference window [EISE90]. In a general way, the concept of reference window is related to a set of matrix elements referenced by two statements that carry a data dependence. We used the size of reference windows to have an indication of data locality (cost).

We have conducted a study with a parallel program that multiply two matrixes with the following dimensions, 1024×128 and 128×256. This program was executed in a parallel computer with eight processors, a Silicon Graphics Power Series 4D/480 VGX.

Applying this program in our prototype, we got the following output (table 8).

Table 8 - Output of our prototype system.

program version	ijk	ikj	jik	jki	kij	kji
cost	33,155	33,153	263,425	263,429	132,225	132,233

Executing each version of the program in a real machine, we got the execution time in function of the number of processors. The figure 13 and table 9 below resume the results.

Table 9 - Execution time of each version of the matrix multiplication program.

number of processors / program version	Execution time (sec)							
	1	2	3	4	5	6	7	8
ijk	65.53	33.32	22.29	16.72	13.57	11.40	9.86	8.62
ikj	36.42	18.21	12.19	9.27	7.52	6.39	5.50	4.82
jik	71.45	36.32	24.36	18.42	14.86	13.30	11.68	9.55
jki	151.85	90.93	67.07	60.50	55.38	55.67	52.18	46.62
kij	148.66	78.15	56.17	43.44	38.81	32.84	29.21	24.73
kji	258.54	140.80	105.25	87.04	80.01	72.07	66.41	60.36

The results show that our prototype system chose the correct version as the one with better locality, the `ikj` version. This proves that our approach is valid, and it is possible to automatically optimize the memory access pattern and integrate optimization of locality of reference and parallelism.

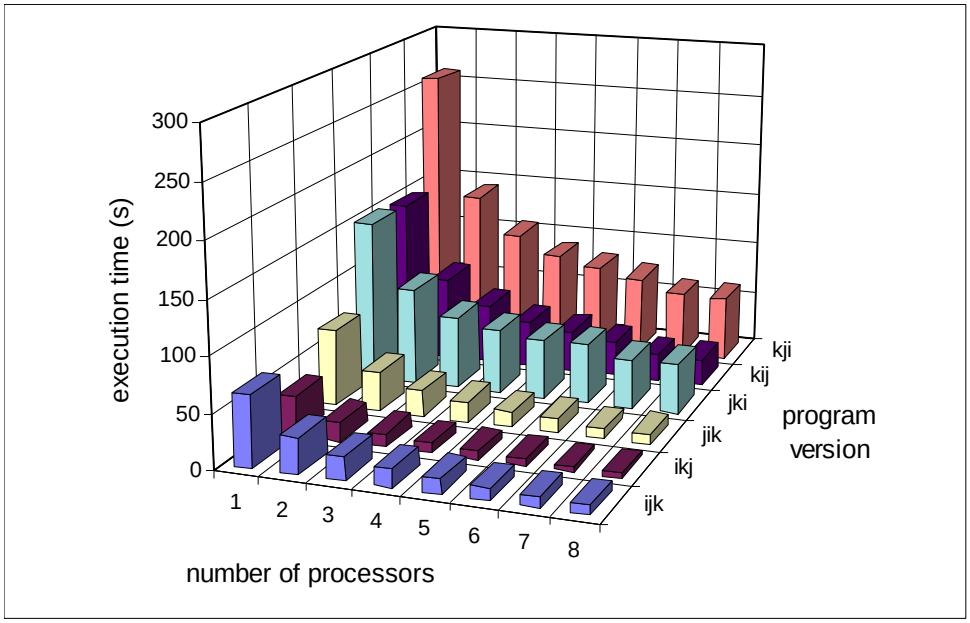

Figure 13 - Execution time for matrix multiplication program versions.

8. Conclusion and Future Work

This paper presented our research in methods to obtain high-performance programs, by presenting COMMUNION. Communion is a new memory management strategy that emphasizes cooperation of system programs. Unfortunately, this area is restricted to specialists in parallel processing and memory management by now because it requires some specific knowledge.

The evaluation studies had shown that our approach is valid. We got very good results with a performance gain in both techniques presented. The CAPR is a novel page management technique that is based on the integration of the operating system and the compiler. This integration can be applied in other situations, like for scheduling parallel programs or distributing data across memory modules. The application of program transformations in order to improve data locality has proved to be very effective. The speed-ups confirm our expectations. And the prototype system that implements this approach has shown to be very efficient to improve program performance.

As future work, we are planning to complete the implementation of the automatic optimizing system in order to provide such a tool for non-specialist users. In this way, more people can make use of these modern machines. On the other hand, we want to continue to research new techniques to improve program performance: in CAPR, we plan to test new directives and in Resurgence, new optimizations will be tested.

This research is important because of its primary objective, which is to allow the use of modern parallel machines for all users that are not specialists in parallel processing. We aim that all scientific community can make use of these new generation computers.

Acknowledgments

The authors wish to thank all members of our research group for their valuable work. We are grateful to the people that have revised the earlier drafts of this paper.

References

[AHO86] AHO, A.V.; SETHI, R.; ULLMAN, J.D. **Compilers - principles, techniques and tools**. Addison-Wesley, 1986.

[BANE94] BANERJEE, U. **Loop parallelization**. Kluwer Academic Publishers, 1994.

[COCK95] COCKCROFT, A. **Sun performance and tuning: sparc & solaris**. Prentice-Hall, 1995.

[EISE90] EISENBEIS, C. et alii. **A strategy for array management in local memory**. Rapports de Recherche n^{o} 1262. INRIA, France. 1990.

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

[HWAN93] K. Hwang. **Advanced computer architecture: parallelism, scalability, programmability**. McGraw-Hill, 1993.

[KREM95] KREMER, U. **Automatic data layout for distributed memory machines**. Ph.D. Thesis. Computer Science Department, Rice 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.

[MARI96] MARINO, M. D. **Pulsar: um sistema de memória virtual compartilhada e distribuída, baseado em consistência relaxada**. Master Thesis, Polytechnic School, University of São Paulo, 1996.

[MIDO94] MIDORIKAWA, E. T. **Análise da otimização de acessos à memória**. In: Proceedings of 6th Brazilian Symposium on Computer Architecture - High Performance Computing, p.37-52, Caxambu, MG, Brazil, August 1994.

[MIDO95] MIDORIKAWA, E. T. et alii. **Um sistema integrado para otimização automática de paralelismo e localidade de dados**. In: Proceedings of 7th Brazilian Symposium on Computer Architecture - High Performance Computing, p.523-537, Canela, RS, Brazil, July 1995.

[MIDO97] MIDORIKAWA, E. T. **Uma nova estratégia para a gerência de memória para sistemas de computação de alto desempenho**. Ph.D. Thesis. University of São Paulo, São Paulo, Brazil 1997.

[MIDO98] MIDORIKAWA, E. T. **Otimização de localidade de dados em aplicações reais e estratégias de alocação dinâmica de memória**. In: Proceedings of 10th Brazilian Symposium on Computer Architecture - High Performance Computing, Búzios, RJ, Brazil, October, 1998.

[POLY88] C. D. Polychronopoulos. **Parallel programming and compilers**. Kluwer Academic Publishers, 1988.

[TAKA95] S. Takahashi et alii. **Análise do padrão de acessos e otimização de localidade em sistemas de computação de alto desempenho**. In: Proceedings of PANEL'95 (CTIC'95), v.2, p.1479, Canela, RS, Brazil, July 1995.

[TANE97] TANENBAUM, A. S. & WOODHULL, A. S. **Operating systems: design and implementation**. 2nd. Ed., Prentice-Hall , 1997.

[VEEN94] VEENSTRA, J. E. & FOWLER, R. J. **MINT tutorial and user manual**. Technical report 452, Computer Science Department, University of Rochester, 1994.

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