RC 15953 (#70916) 7/23/90
Computer Science 7 pages

Research Report

Visualizing Computer Memory Architectures

Bowen Alpern, Larry Carter and Ted Selker

IBM Research Division
T. J. Watson Research Center
Yorktown Heights, NY 10598

Visualizing Computer Memory Architectures

Bowen Alpern, Larry Carter, and Ted Selker

IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598

Abstract — The Memory Hierarchy Framework is a conceptual model together with a visual language for using the model. The model is more faithful to the structure of computers than the Von Neumann and Turing models. It addresses the issues of data movement and exposes and unifies storage mechanisms such as cache, translation lookaside buffers, main memory, and disks. The visual language presents the details of a computer's memory hierarchy in a concise drawing composed of rectangles and connecting segments. Using this framework, we have improved the performance of a matrix multiplication algorithm by more than an order of magnitude. We believe the framework gives insight into computer architecture and performance bottlenecks by making effective use of human visual abilities.

Visualizing Computer Memory Architectures

Bowen Alpern, Larry Carter, and Ted Selker

IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598

Abstract — The Memory Hierarchy Framework is a conceptual model together with a visual language for using the model. The model is more faithful to the structure of computers than the Von Neumann and Turing models. It addresses the issues of data movement and exposes and unifies storage mechanisms such as cache, translation lookaside buffers, main memory, and disks. The visual language presents the details of a computer's memory hierarchy in a concise drawing composed of rectangles and connecting segments. Using this framework, we have improved the performance of a matrix multiplication algorithm by more than an order of magnitude. We believe the framework gives insight into computer architecture and performance bottlenecks by making effective use of human visual abilities.

1 Introduction

This paper is a case study in the application of visual language techniques to a limited problem domain. This domain is performance tuning of programs to make efficient use of a particular computer's memory architecture. We present a detailed conceptual model of computer memory, and a visual language for comprehending the model.

Models allow people to concentrate on relevant features in a problem domain. Some models employ a *visual language*, the systematic use of visual techniques to represent the model [7]. These techniques have been shown to improve problem solving performance in structural domains [5].

A good example is the Turing machine [1] model introduced in the 1930's to understand what functions are computable. A Turing machine is a mathematical construct that manipulates strings of symbols. It is most easily described visually: a long tape is divided into cells, each of which can hold a single symbol. A cart rolls along the tape, reading and possibly modifying the symbol under it. Often, a simplified visual language is used to convey specific Turing machine configurations. The position of the cart is represented by a line under a particular symbol and the program driving the cart is also presented diagrammatically. Invented before computers, the model is still used as an aid for teaching.

Since the Turing machine bears little relationship to the architecture of real computers, it is no surprise that the model is not useful for constructing efficient computer programs. Instead, programmers use the Von Neumann or *RAM* model [1]. A computer is modeled as a memoryless processor and a separate random access memory. The assumption is made that each memory reference takes one unit of time. This assumption permits the running time of a program to be analyzed. The model allows programming without consideration of hardware details. Unfortunately, some of the details ignored by the model have a significant impact on performance. Thus, RAM analysis may be inaccurate.

In trying to analyze and improve some programs, we found the RAM model inadequate. Section 2 describes the Memory Hierarchy Framework, a more accurate computer model together with visual language for presenting it. Sections 3 and 4 show how the framework can be used to illustrate and avoid performance bottlenecks in matrix transposition and multiplication algorithms. Section 5 shows how parallelism is reflected in the framework. The rest of the paper discusses insights gained by using the framework and suggests areas for future research.

2 Memory Hierarchy Framework

This section presents a framework for understanding computers' memories. The framework has two parts: a conceptual model and a visual language that communicates the model and facilitates performance tuning.

We first mention some features of real computers. A computer's memory is organized as a set of increasingly fast *memory units;* for example, disk, main memory, cache and registers. *Blocks* of data are transferred between units along *wires*. Moving a byte automatically moves all the bytes in the same block as well. Different wires and memory units have different speeds and sizes, as well as other peculiarities.

The above features affect performance significantly. Often programs can be made to run many times faster by properly coordinating data movement among the different units [4]. Unfortunately, programming languages and the RAM model hide the irregularities of memory. Few people master the ability to tune programs effectively. Our work is aimed at making this skill less esoteric.

Conceptually, we model the non-uniform nature of memory by a hierarchy of *memory modules*. Each memory module consists of a *box* that can store data and a *bus* that can move

3 Matrix Transposition

Transposing a matrix is a simple problem. However, ignoring the memory hierarchy can significantly degrade the performance. We use our framework to illustrate this.

The following FORTRAN code assigns the transpose of matrix A to matrix B.

```
      DO 10 I = 1, N
      DO 10 J = 1, N
10    B(J,I) = A(I,J)
```

Consider any memory module where N is larger than the blocksize and the blockcount. Arrays are stored by column in FORTRAN. When $A(1,1)$ is referenced, a block containing part of the first column of A is transferred into the memory module. Next $A(1,2)$ is referenced, and a second block is brought in. After blockcount steps, the memory is full, and some block (typically the first brought in) must be discarded. This process – bringing in a new block and discarding an old – continues. Unfortunately, by the time the next row of A is begun, the needed data have already been discarded from the memory module! The result is excessive data movement – each assignment takes (at least) the amount of time required to transfer a whole block of data. This situation, where the cost of transferring data swamps the cost of computation, is called *thrashing*.

The framework shows this problem in a natural way. Superimpose a square representing the array on top of the memory module. Notice that the vertical direction of both the array and the memory rectangle represent contiguous storage. If the square fits comfortably inside the rectangle, the entire array will fit into the corresponding memory unit. The module will not be a performance bottleneck. If the array is too tall but not too wide, the module can hold one block's worth of each column. Since each block will be brought in only once, performance will not be degraded by unnecessary data transfers. But one must examine the problem more closely when the array is too wide for the rectangle, or when part of the array falls within a shaded portion.

The unnecessary block transfers of the A matrix would be eliminated by simply interchanging the order of the loops. Unfortunately, the B matrix would now have the identical thrashing problem.

There is a way out [6]. The A and B matrices are partitioned into subarrays small enough that two can fit into the memory module. The program works on one subarray of A at a time, transposing it into the symmetrically located subarray of B. These subarrays must be further partitioned into subsubarrays so they can fit into the next lower memory module in the hierarchy. The entire process is shown in Figure 2. A program that implements the picture for a k-level memory hierarchy would have $2(k-1)$ nested DO-loops.

We use a new scale inside the memory rectangles. As described earlier, the logarithmic scale allows the entire hierarchy to be compared in a single image.

The schematic diagrams inside the rectangles highlight issues in moving data between levels. Specifically the shaded submatrices of a level are the data which is transfered to the next level. It is expanded to the next level but retains its shape.

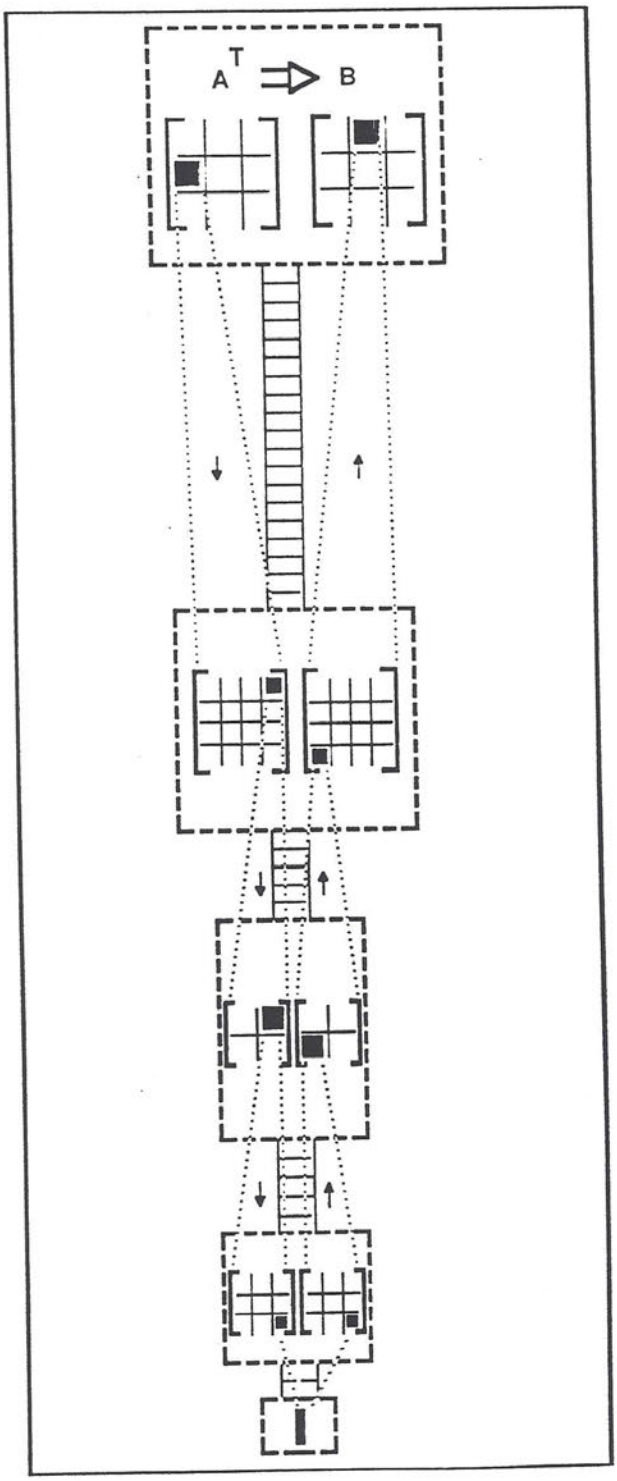

Figure 2. Transposing a Matrix.

5 Parallel Computers

Currently there exists a vast collection of different architectures for machines with more than one processor. Such parallel and distributed machines can be modeled in the Memory Hierarchy Framework. The visual language allows several memory modules to hang below a single rectangle. To avoid overlapping rectangles, busses can have horizontal segments; the transfer time is depicted by just the vertical distance the bus covers. An entire machine is represented as a tree of memory modules as in Figures 4, 5, and 6. The multiple processors are at the leaves of the tree at the bottom of each picture.

The semantic meaning of multiple bus lines hanging below a rectangle is that data can be transferred along all of the busses simultaneously. The busses can carry data from different addresses in the shared memory module.

As with any model, certain hardware restrictions have been ignored. In particular, each machine and operating system has a specific set of rules about data sharing and data protection. A programmer may need to consider these issues carefully for a particular application.

The pictures in Figures 4, 5, and 6 are representative of three very dissimilar multicomputer architectures: the supercomputer, the massively parallel computer, and heterogeneous networks of computers. The different architectures are reflected in how much branching is at different levels. Any particular machine in a given class will also vary from the pictures in specific details, such as the dimensions and number of processors.

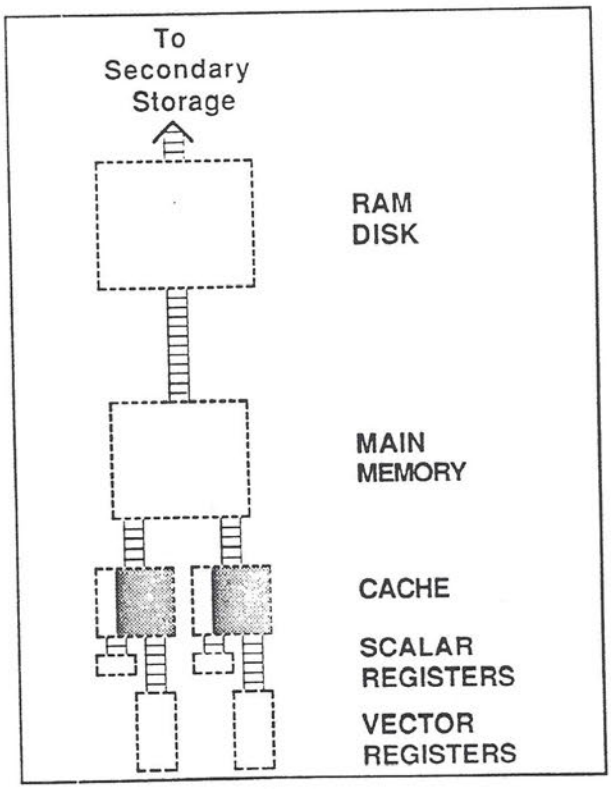

Figure 4. A Supercomputer.

Figure 4 is a typical picture of a *supercomputer*, such as the CDC and CRAY machines and large IBM mainframes. These machines have a relatively small number of processors, and the branching occurs near the leaves. Supercomputers often have vector processors that allow them to "crunch" data extremely efficiently, provided that data are arranged properly in memory. The vector processor shown in Figure 4 is long, thin, and connected by a relatively long bus. This describes a machine that requires vector data to be stored contiguously. Some machines have more flexible vector processing. They would be depicted with shorter and wider vector rectangles.

Some supercomputers gain speed by having several functional units (ALUs) that work on data in a single group of registers. These could be represented by another level of branching below the registers. Again, it is interesting to see that the parallelism of supercomputers is concentrated towards the leaves.

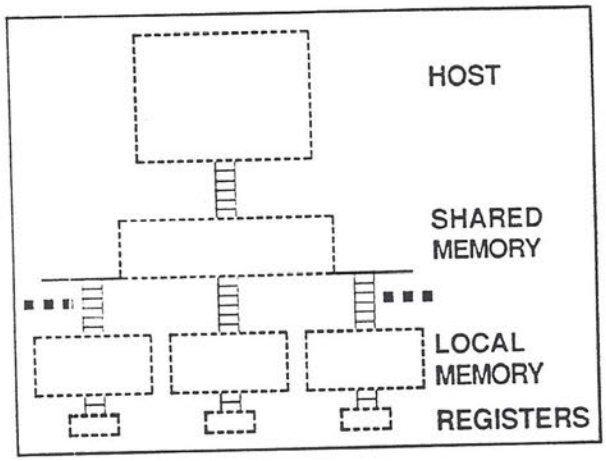

Figure 5. A Massively Parallel Architecture.

Another class of machine, the *massively parallel computer*, has a very high branching factor near the middle, but little at the top or bottom (see Figure 5). The particular machine of the picture has 1024 processors, each with a quarter megabyte of memory, connected by a fast switching network. The machine has a communication latency of 32 cycles and allows data to be transferred at a rate of 1 byte per cycle. The picture approximates this by showing that 32 byte blocks can be transferred in 64 cycles. The picture is accurate within a factor of 2 for any sized message.

Another class of multicomputer is the loosely coupled network. Pictures of these machines are characteristically bushy near the root of the tree, as in Figure 6. A set of workstations of different capabilities can share the same disk memory (fileservers). Files are moved through the network. The top rectangle represents the totality of data stored in the network. Although these data are distributed physically, the system behaves as if there is a memory unit containing a copy of all the data.

The width of the unshaded sub-rectangle represents associativity. A branch of the tree represents a bus. Its length represents transfer time. In order to comprehend widely disparate quantities visually in a confined space, sizes are chosen to be proportional to the logarithms of quantities rather than to the quantities themselves.

The visual language and the model were developed in tandem. Indeed, the language facilitated the development of the model. It also facilitates thinking about the model. In addition, it makes identification of potential performance bottlenecks easy. Techniques described in Sections 3 and 4 were used to obtain a 30-fold speed-up in an implementation of the matrix multiplication routine of LAPACK. We believe the framework will be useful in speeding up the implementation of other computationally intensive applications. We hope that the visual language will prove a convenient medium for communicating our model to others.

This paper is a case study of a successful application of visual language techniques in a limited domain. We hope that it can serve as a demonstration to encourage others to build languages to aid problem analysis. Only a tiny collection of visual techniques is used in our framework. Barely scratching the surface of the visual language approach we have found this approach powerful. In order to unleash the full power of visual language, a large collection of techniques must be systematically catalogued, and experience must be developed as to the appropriate uses of each. To this end, further case studies are called for.

References

[1] Aho, A. V., J. E. Hopcroft, and J. D. Ullman. "The Design and Analysis of Computer Algorithms." Addison-Wesley, 1974.

[2] Alpern, B., L. Carter, and E. Feig. "Uniform Memory Hierarchies." to appear *Proc. 31st IEEE Foundations of Comp. Sci.* October, 1990.

[3] Demmel, J., J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, and D. Sorensen. "LAPACK Working Note #1: Prospectus for the Development of a Linear Algebra Library for High-Performance Computers." Argonne National Laboratory, ANL-MCS-TM-97, September, 1987.

[4] Gallivan, K., W. Jalby, U. Meier, and A. H. Sameh. "Impact of Hierarchical Memory Systems on Linear Algebra Algorithm Design," Int. J Supercomputer Appl., Vol. 2, No. 1, Spring 1988, pp. 12-48.

[5] Monika, G., and G. Rohr. "Which Task In Which Representation on What Kind of Interface." *Proc. Interact '87*, pp. 513-520, September, 1987.

[6] Rutledge, J.D., and H. Rubinstein. "Matrix Algebra Programs for the UNIVAC." Presented at the *Wayne Conference on Automatic Computing Machinery and Applications*, March, 1951.

[7] Selker, T. and L. Koved. "Elements of Visual Language." *IEEE Workshop On Visual Languages*, October, 1988.

Copies may be requested from:

IBM Thomas J. Watson Research Center
Distribution Services F-11 Stormytown
Post Office Box 218
Yorktown Heights, New York 10598