When Virtual Memory Isn't Enough

Computer Science Report No. 136

Clark D. Thomborson
Computer Science Department
University of Auckland
Private Bag 92019, Auckland
New Zealand
cthombor@cs.auckland.ac.nz

November 14, 1996

Abstract

Virtual memory, even on the largest and fastest contemporary computers, is neither large enough nor fast enough for all applications. Some data structures must be held in the file system, and some "performance hints" must be given to the memory-management runtime routines. For these reasons, most large-memory application codes are littered with system-specific names, constants, file-migration policies, pragmas and hints. Such codes are very difficult to develop, maintain, and port.

I propose a new paradigm for the design of large-memory codes, providing many of the performance advantages and few of the drawbacks of system-specific coding techniques. In my paradigm, programmers must organize their data structures into a series of (nesting) data blocks, with blocksizes B_h increasing in a fractal (power-law) progression $B_h = RB_{h-1}^{\delta}$; $B_1 = 1$. Furthermore, the larger blocks must be referenced much less frequently than the smaller blocks. I argue that efficient algorithms for several important problems on workstations and PCs are found at $\delta \approx 3/2$ and $R = 8$. I sketch a model of memory performance that explains why non-hierarchical large-memory codes require system-specific tuning for efficient execution.

1 Introduction

Most algorithmic theorists, computer architects, compiler designers, and performance programmers are aware that memory latency and bandwidth are very important issues. The main difficulty, from a theorist's point of view, is to find an appropriate model for the study of these issues. The model must be analytically tractable and widely applicable.

From the practitioner's point of view, it would be helpful to have a simple model to guide the design and use of existing and future computers. The theorist can guide the practitioner by pointing out the algorithmic implications of design choices, *i.e.* demonstrating an intrinsic latency or bandwidth bottleneck in an architecture, with a proof that no algorithm for some interesting problem can run efficiently on this architecture. The practitioners can guide the theoreticians toward an appropriate model, *i.e.* by pointing out the most common bottlenecks in existing systems. If these bottlenecks can not be observed in a theoretician's model, then clearly this model is inappropriate for studying performance of existing systems.

The four-parameter model for memory performance proposed in this paper is greatly influenced by previous research into hierarchical memory. My model is, for the most part, a reparameterization of a subset of the systems that can be described by the Uniform Memory Hierarchy (UMH) of Alpern, Carter, Feig and Selker [3] and its parallelization[9]. Most importantly, I do not allow latency to be an arbitrary function of the address. Instead, I require latency to be a power function. This allows me to write a compact description of the latency requirements of an algorithm, as a constraint on the exponent α appearing in the latency function of the architecture.

The UMH models discussed by Alpern *et al.* have only singly-exponential growth in their memory capacities, corresponding to the limiting case of the layer capacity parameter $\delta = 1$ in my model. My analysis indicates that $\delta = 1$ is only appropriate for memory architectures which are optimized for large-blocked computations such as matrix-matrix multiplication. Many important computational problems, notably hash-table lookups and data streaming from I/O devices, require an architecture with $\delta > 1$ for efficiency. Note: the capacity C_h of the h-th layer of memory, in my model, is $C_h = RC_{h-1}^{\delta}$ with $C_1 = R$. This recurrence dictates doubly-exponential growth in layer capacity for $\delta > 1$, but only singly-exponential growth for $\delta = 1$. Memory systems with $\delta > 1$ are thus quite "flat," and therefore more suited for hashing, by comparison with those described by $\delta = 1$.

I am aware of simpler but still very interesting models, such as the Hierarchical Memory Model (HMM) of Aggarwal, Alpern, Chandra and Snir [1] or the Block Transfer (BT) model of Aggarwal, Chandra and Snir [2]. Indeed, the HMM model is a limiting case ($\delta = 1$, $R = 2$) of my LMH model; as indicated above, I believe it important to study systems with $\delta > 1$. Also, the scaling factor R should be 4 or 8, not 2 as in the HMM, in the computer systems I have analyzed. Similarly, the BT model is an unrealistic, limiting case, requiring $\gamma = 0$ in my model to give constant bandwidths across the memory hierarchy. Bandwidths in real-world computer systems fall off dramatically as one moves to higher addresses. For example, disk-DRAM bandwidths are a very small fraction of register-cache bandwidths in all contemporary machines. I have found that a power law with exponent $\gamma = 1/3$ is a good fit to the memory layer bandwidths in desktop machines.

In summary, because I believe in the importance of the γ and δ parameters on algorithmic and system design, I am unwilling to work within existing models.

There is a large body of work on two-level memory hierarchies, most notably a classic paper by Floyd [4], and recent publications by Vitter and Shriver [8] and Vengroff [7]. In my view, such models are very useful when tuning codes to specific systems. They are also very helpful in exposing the relationships between latency, bandwidth, and capacity at a known bottleneck in the memory hierarchy, such as the one between DRAM and disk on most contemporary systems. However, in my experience, algorithms designed for two-level hierarchies can perform very poorly when executed under unexpected conditions, for example when given a small allocation of DRAM. By contrast, algorithms designed for a multi-layer hierarchy can be more graceful in their performance degradation on heavily-loaded systems. Indeed, this is one of the major goals of my research: to give a framework for algorithmic design that is relatively insensitive to the exact values of system performance parameters. Or, to state the converse, I seek a small set of system performance parameters such that, given approximate information about their values, I can design efficient algorithms.

The memory model proposed in this paper may be viewed as an extension of Thiebaut's analytic model of cache behavior [5]. Thiebaut's model is designed to predict cache miss rates from synthetic address traces, without the need for simulating the contents of cache. The address traces in Thiebaut's model are modeled by a fractal process. Appropriate parameters for Thiebaut's fractal process are determined by best-fit approximation to a few observations of actual program behavior. The address-generating process in Thiebaut's model has just one fractal dimension, describing both its spatial and temporal locality.

My memory model, in contrast, is given appropriate parameters by observing the latency and bandwidth available at various memory layers in existing systems. The address-generating process in my model has two fractal dimensions, α and γ, allowing temporal locality to be distinguished from spatial locality. (Essentially, the distinction is between latency-limited computations such as hashing, and bandwidth-limited ones such as matrix-matrix multiplication.) I am not aware of any prior work on analytic memory modelling that makes this distinction.

2 Motivation and Background

Over the past five years, I have been involved in the development, maintenance, and performance-enhancement of several large-memory codes. One of my goals in writing this paper is to start the process of translating my "practical" experience into a "theoretical" model of computer performance. Thus I believe it appropriate for me to outline my practical experience to give you, the reader, a sense of the types of bottlenecks and systems I am endeavoring to model in this paper.

Some of my practical work was in the private sector. For example, I worked on the memory-management routines of a PostScript interpreter for a wide-bed color printer manufactured by LaserMaster Inc. of Minnesota, USA. An uncompressed page image could require more than 500 MB, depending upon its dimension and color depth. Our interpreter was running on a PC with only 64 MB of RAM. One of my tasks was to determine whether its memory-management could be modified so that the interpreter would run efficiently on a PC with 32 MB of RAM.

As another example, with Larry Carter and Bowen Alpern of the IBM TJ Watson Research Laboratory, I developed a code for computing rectilinear Steiner minimal trees on $k < 24$ pins using the Dreyfus-Wagner recurrence[6]. Before storage optimization and data compression, our data structures required $2k^2 2^{k-1}$ 16-bit integer words. For $k = 23$, this is 4.5 GB, or slightly more than the 32-bit addressing limit of the workstations (and most supercomputers) available until very recently.

In reflecting on my experience with large-memory codes, I have come to a number of conclusions, listed below.

- No contemporary virtual memory system is large enough to satisfy all programmers. For some applications, we must construct data structures that, even in compressed format, would greatly exceed the size of our paging disk. Indeed, we sometimes use data compression techniques to build data structures that, if decompressed, would overflow all our file systems. Thus even if we are willing to memory-map files into our virtual memory system, using system-specific coding techniques, we may not have enough room in virtual memory for all our data structures.

- Without careful design and implementation, large-memory codes tend to run at the speed of the paging device, *i.e.* one data reference every 10 milliseconds.

- The best way to avoid latency bottlenecks is to access data blockwise. The blocks must be large (64 KB) if the data structure will not be RAM-resident. The blocks must be very large (1 MB) if the data structure is too large to fit on a local disk.

- Some computational problems (*e.g.* hashing) are intrinsically latency-bottlenecked. To obtain good performance on large instances of such problems, you must organize your data by frequency-of-reference. The virtual memory manager will do some, but by no means all, of this organization for you.

- Large-memory codes are rarely CPU-bottlenecked, unless they are very carefully designed.

- Data compression can increase the performance of codes, essentially by trading CPU cycles for capacity in the critical memory resource. The critical memory resource is the layer (usually DRAM in contemporary systems) just inside the latency or bandwidth bottleneck.

- It is a poor idea to add data compression to a code before you know its performance bottleneck on a given system.

- If you know approximate values for just four system- and load-specific parameters for any uniprocessor workstation or PC, you can design efficient large-memory codes for that system.

The purpose of this paper is to explain, and justify, the last of the assertions listed above, that the value of four memory performance parameters are indeed enough to support efficient coding practice on uniprocessor machines. The current state of the art in performance programming is, in my experience, quite far from this. Every code I have inspected, and all that I have written myself, contains dozens of system-specific parameters or coding tricks. All tricks and parameter values must be analyzed, and most seem to require revision, every time the code is ported to a new system, or tuned to a new system configuration.

In future research, I plan to address the multiprocessor case. I believe that a performance model with a small number of parameters (perhaps five) will cover most small-scale multiprocessing systems of commercial importance, as soon as the design of such systems has "converged" sufficiently to support non-system-specific performance programming. Massively-parallel computation may also, someday soon, admit simple, generic performance models.

Computation on uniprocessor desktop systems is at a very interesting juncture. Memory system architecture on these systems has converged to the point that, at least in theory, it is possible to write portable, high-performance, large-memory codes. My aim in this paper is to bring this theory closer to practice.

3 Existing Paradigms for Large-Memory Coding

(Readers familiar with the problems involved in writing large-memory codes for computers with hierarchical memory may wish to skip ahead to the next section.)

I know of two paradigms for developing large-memory codes. The most common paradigm is to make an assumption about the amount of RAM that will be available at runtime, then write explicit file accesses and buffer-management routines that stay within the assumed RAM budget. This paradigm tends to lead to complex codes with deeply-embedded assumptions about RAM size and file system speed.

The second paradigm for large-memory codes is to assume that the virtual memory system is adequate to the task, within certain limits. The trick is to write codes that stay within these limits. A successful second-paradigm code achieves adequate performance without spilling any of its data structures into the file system.

In a simplistic example of second-paradigm coding, a programmer might declare a 100000×100000 array of floating point numbers. A C-language inner loop that forms a dot product on two rows of this array will run with marginally-acceptable efficiency on most workstations or PCs, due to the predictive and blocking properties of most virtual memory systems. In the worst case, both rows are not resident in RAM, but they can be fetched into RAM from the paging disk within a few tens of milliseconds. The computation will thus proceed at approximately the peak bandwidth of the paging disk, typically 1–2 megabytes/second on a desktop computer, or about 10^5 inner-loop operands per second. Because a modern CPU can complete only about 10^8 inner-loop operations per second even if its operands are not delayed, a simple large-memory code based on the second paradigm will run at 0.1% CPU utilization. For many applications, this is more than adequate, but sometimes greater performance is required.

Simple second-paradigm codes have the advantage of portability. Because most virtual-memory systems make similar policy decisions about when to migrate a page from RAM to disk, second-paradigm codes often (but not always) run faster on higher-performance systems. For example, a computer with a 10 MB/second paging device would run the code of the previous paragraph with a 10-fold speedup over a computer with a 1 MB/second paging device.

Second-paradigm code becomes problematic when the data structures become too large. Traditional implementations of virtual memory become expensive, and tend to become much slower, when their capacity is increased. At the present time, I believe the largest commercially-available virtual memory system is limited to 14 GB, in Digital Equipment Corporation's VLM64 technology for their mainframe/database product line. This indeed provides a lot of space for a data structure, however it is not a desktop technology.

If your desktop file system is hierarchical, *i.e.* a 10 GB disk backed by a 100 GB tape, then at least in principle you should be able to write a simple code for a linear-time computation that builds a 50 GB data structure. This 50 GB computation need never even spill to tape if you can devise a compression algorithm that acheives at least 5:1 compression on the disk images of your data structures.

Regrettably, I know of no API that provides a standardized method of calling a special-purpose data compression algorithm whenever a block of data pages in RAM is flushed to disk. Some disk and tape drives provide general-purpose data compression routines, but in my experience these are rarely competitive, on non-textual data, with fairly straightforward special-purpose routines. When writing code in C/C++ under Unix-like operating systems, you could install signal handlers that call data compressors and decompressors during page faults. This is a first-paradigm coding trick: rather non-portable, and certainly non-standard.

For economic and practical reasons, then, you are forced into the first paradigm (of using the file system and possibly special-purpose data compression routines) if you want to write codes accessing data structures larger than your disk.

Another major difficulty with second-paradigm codes arises when they are pushed to their performance limit. The simple dot-product code outlined previously in this section would run r times faster if each row is used r times (on average) in an inner-loop execution before it migrates back out to disk. The easiest way to increase the RAM-use factor r for a second-paradigm code is to run it on a system with enough RAM to hold all the most-commonly-referenced data structures. In our example, the data structure is a $10^5 \times 10^5$ array, or approximately a hundred gigabytes. No contemporary system has this much RAM. In general, large-memory codes must be designed to achieve a high RAM-use factor r when only a miniscule fraction of their data structures are RAM-resident. Sometimes this is trivial to arrange, sometimes it is difficult, and sometimes it is impossible.

I invite you to consider the problem of trying to write an efficient second-paradigm code for the problem of calculating the sum s of a fixed subset of all possible inner-products of rows of a matrix x with 10^5 rows and 10^5 columns. For the moment, let us assume that you have managed to obtain access to a system providing more than 100 GB of virtual address space, so that it is not necessary to use first-paradigm (file-system) coding techniques.

In algebraic notation, s is defined by a triple summation,

$$s = \sum_{1 \leq i \leq 10^5} \sum_{j \in f(i)} \sum_{1 \leq k \leq 10^5} x_{ik} x_{jk} \tag{1}$$

Note: this triple summation is computationally similar to the kernel of the Dreyfus-Wagner method for rectilinear Steiner tree minimization, although the basic operation of the Dreyfus-Wagner computation is defined with (ADD, MIN) as ring operations rather than the (MUL, ADD) suggested by the notation of Equation 1. The set-valued function $f(i)$ defines the indices j of the rows that should be combined, by dot-product, with row i.

For concreteness, let us assume that

$$|f(i)| = 500 \qquad \forall i \tag{2}$$

Under this assumption, each row x_{i*} appears as left-hand operand in exactly 500 inner products of the form $\sum_{1 \leq k \leq 10^5} x_{ik} x_{jk}$. The complete computation then requires exactly $10^5 \cdot 500 \cdot 10^5$ multiply-add steps, or 10^{13} floating-point operations, if performed using (MUL, ADD) on reals. We might hope to complete this computation in a couple of weeks (about 10^5 seconds) on a desktop machine capable of storing 100 GB of data, and capable of running at 100 MFlop/sec.

In naive second-paradigm C-code, we would write a triply-nested loop to compute s. Because C stores arrays in row-major format, it is "obvious" that k should be incremented in the innermost loop. If, instead, i were incremented in the innermost loop, our code would make stride-10^5 accesses through our 100 GB array. Such a computation would obtain one inner-loop operand per page fault, thus it would run at about one mul-add step every 20 milliseconds. This is clearly unacceptable: at a rate of 2 Flop/20 milliseconds = 10^2 Flop/sec, our 10^{13} Flop computation would take 10^{11} seconds. Your paging device probably wouldn't survive even 10^5 seconds of continuous operation, and even if it did, I doubt you'd want to wait more than 10^8 seconds (a few years) for any single computation to complete.

After some reflection, then, we might settle on the following design for a second-paradigm code in the C-language. I will call this the "naive" design. This code would have three nested loops. The outermost loop, iterating on i, would scan the matrix x sequentially by rows. The middle loop, iterating on j, would scan the matrix x sparsely by rows, entering the inner loop once for each member of $f(i)$. The inner loop would perform a dot-product on two row-vectors of length 10^5. Unless there is enough RAM to hold the entire matrix x, there is no reuse of the x_{j*} operand: new rows will arrive at the CPU at the bandwidth of the paging disk. The old rows will be paged-out long before they are referenced again as x_{j*} operands. The x_{i*} operand behaves differently. Each row x_{i*} can be stored in about 1 MB of RAM, and each element in that row will be reused about once every 10^5 computational steps. The reuse count for the x_{ik} elements stored in RAM is thus 500, the cardinality of the set $f(i)$. The reuse count for the x_{jk} elements is close to zero, so this is clearly our bottleneck on contemporary systems.

By the arguments above, the inner loop in our naive second-paradigm code will execute at the rate of the x_{jk} fetches from disk: (1 disk operand / mul-add) (2 MB / sec) / (8 B / operand) = 0.25 mul-adds per second on a system with a 2 MB/second paging disk and an 8-byte representation for floating point numbers. This is 0.5 megaflops, or about 0.5% of the 100 MFlop/sec peak performance of our typical contemporary machine. A 0.5%

CPU efficiency is marginally acceptable for some applications, but in this case, our hoped-for ten-day computation would require years to complete.

Some, perhaps most, readers of this paper will know an appropriate second-paradigm optimization for this example. The usual trick is to make an assumption about the capacity of RAM, then make an appropriate "block structure" in our code. For example, it is reasonable to assume that there is room for a few dozen rows of our matrix x in RAM, that is, that our process will have a few tens of megabytes of RAM-resident data structures. We can then efficiently access x in blocks of, say, $c = 10$ rows in our outermost loops. The resulting code has the following algebraic form:

$$s = \sum_{u=1}^{10^5/c} \sum_{v=1}^{10^5/c} \sum_{i=c(u-1)+1}^{cu} \sum_{j \in g(v,i)} \sum_{k=1}^{10^5} x_{ik} x_{jk} \tag{3}$$

where

$$c = 10 \tag{4}$$

and

$$g(v,i) = \{j : c(v-1)+1 \leq j \leq cv)\} \bigcap f(i) \tag{5}$$

To achieve computational efficiency despite low disk bandwidth, we must arrange our data for optimal reuse in the following sense. Each index-set $g(v, i)$ arising in the computation should have either zero or $\max(c, 500) = 10$ members, so that the two innermost summations specify either a no-op or, for some v and i, a computation involving $10k$ mul-adds on 11 distinct k-vectors. Also, these computations on k-vectors should proceed in blocks of 10, that is, we should have $g(v, i) = g(v, i+1)$ for all i not divisible by c. The three innermost summations will then specify either a no-op or computation involving $100k$ mul-adds on just 20 distinct k-vectors in RAM. This would give us $100k$ floating-point operations for every $20k$ words of data fetched from RAM to disk, if none of the k-vectors were reused across invocations of the three innermost loops. In fact, some of these operands will be reused; we will reuse the x_{ik} operands $|f(i)| = 500$ times before they migrate back to disk, as long as there is room for 20 k-vectors in RAM. So the reuse factor is very close to $100k/10k = 10$.

We achieved very close to optimal reuse in our Dreyfus-Wagner computation, for all c simultaneously, by carefully arranging the rows in our array. Note: for arbitrary $f()$ and c, it is difficult and sometimes even impossible to obtain high blocking density through computational rearrangement, so this optimization technique is not always appropriate.

As argued above, our revised code for computing s will reuse each inner-loop operand at least ten times before it migrates out to disk, acheiving a ten-fold speedup over our simple second-paradigm code. This is roughly 5 megaflops, or about 5% of the 100 MFlop/sec peak performance of our desktop machine. Our hoped-for ten-day computation will require months. Possibly it will complete, although I wouldn't recommend trying it. I doubt the paging device on my computer would withstand a few months of continuous use. (Would yours? Would you dare to run this code to completion, in order to find out?)

If we changed the blocking factor in our code to $c = 20$ instead of $c = 10$, we might acheive a twenty-fold speedup over our naive code. We must be careful, however, not to overrun the available RAM. If there is not room in RAM for forty rows of our matrix, our hoped-for twenty-fold speedup will in fact be a two-fold slowdown, because neither of the two row-operands in the inner loop will be RAM-resident at the start of the loop.

The point I'm trying to make is that tuned second-paradigm codes should not be run on a lightly-configured system. Nor should they be run on a robustly-configured system that is under an unexpectedly-high multiprocessing load. Tuned second-paradigm codes tend to perform catastrophically when given insufficient RAM.

A clever second-paradigm programmer would transpose the matrix x before forming the sum s. The computation would decompose into the sum of the results of 10^5 independent computations, one for each column in the original matrix. This optimization is not possible in the Dreyfus-Wagner recurrences, due to a sequencing constraint: in these recurrences, the value of a row x_{i*} is determined by a (simple, but required) computation that is only possible after some of the inner-products on lower-indexed rows have been completed.

Even in cases when transposition is possible, a second-paradigm programmer must proceed cautiously. A transposition operation on a huge matrix must make blockwise accesses to the matrix, if it is to run at optimal speed. This is not terribly difficult to arrange if you know the pagesize, bandwidth, and latency of the paging device, as well as the amount of RAM allocated to your transposition process [8]. Unfortunately, all these quantities, except the pagesize, are runtime variables depending on system load. Even the pagesize might change if you upgrade your computer.

A really clever second-paradigm programmer might write a code that adjusts its blocking factors to the current state of the virtual memory system. Such a programmer might hope for an operating system that would send messages to an application whenever its RAM allocation is changed. I can imagine, although I wouldn't want to implement, a code that responded appropriately to such messages.

Another possibility is to insist on an operating system that will guarantee some amount of RAM, and some performance from the paging device, to each user-level process for its entire lifetime. Such guarantees are not available on most desktop computing systems providing virtual memory, and I am not sure they ever will be. (Note: most versions of DOS, and the MacOS System 7.5 with virtual memory disabled, will guarantee a RAM allocation to a user-level process; the DOS kernel of Windows 3.1, and MacOS 7.5, will allow the foreground process to "hog" most system resources including the paging device; and most operating systems will guarantee RAM allocations to device drivers and other protected-mode processes.

A final possibility is to assume that the problem of tuning, or even writing, second-paradigm codes will be solved by advances in optimizing compilers and/or precompiled libraries. There is certainly some merit to this view: at least in principle, each problem in large-memory, high-performance coding need only be solved once. Each solution, once found, could be enshrined in the library mounted on the systems for which it is appropriate. Clever compilers, linkers, and runtime switches (to adapt to changes in system load) will surely suffice for most people, most of the time. But! Someone has to find the solutions, that is to write the codes, in the first place. This paper is addressed to exactly this problem, that of finding a robust, general method for handling large data structures.

4 A New Paradigm

Nowadays, when I design a large-memory code, my fundamental algorithmic tool is recursion, not iteration. This is not because I am a LISP programmer. I would not write a recursion to solve a small-memory problem in linear algebra. But I do find it an easier task to write a single recursive divide-and-conquer subroutine, than to write a deep loop nest for a clever second-paradigm code. And I am willing to insert a few generally-appropriate constants in my recursive code, instead of trying to write a second-paradigm code that would adapt to various, and variable, runtime environments.

In the previous section, I discussed second-paradigm (virtual-memory, tiled) codes for computing a triple summation s of the form

$$s = \sum_{1 \le i \le n} \sum_{j \in f(i)} \sum_{1 \le k \le m} x_{ik} x_{jk} \tag{6}$$

for the case that $n = m = 10^5$, given some set-valued $f()$ obeying Equation 2.

In my new paradigm, I would declare storage for the matrix x in a pair of recursively-nesting buffers. The smallest pair of buffers in my structure would each hold one of the subarray operands in an inner loop kernel. On a desktop system, a suitable kernel for this problem would be

$$s(u', v', k') = \sum_{u=u'}^{u'+d_u-1} \sum_{v=v'}^{v'+d_v-1} \sum_{i=c(u-1)+1}^{cu} \sum_{j \in g(v,i)} \sum_{k=k'}^{k'+d_k-1} x_{ik} x_{jk} \tag{7}$$

where $g(v, i)$ is defined in Equation 5, and where the tuning parameters c, d_u, d_v, and d_k are given appropriate values, typically small integers. Note that this kernel must be invoked for many different values of u', v', and k', so that all the required inner-products will be taken.

Number of registers R. It is impossible to write a good inner loop without making some assumption about the number R of CPU registers, so this is one of the four parameters in my model of workstation memory. Nominally, $R = 8$.

Let us assume that each element x_{ij} occupies one machine word, and that the data is optimally arranged for dense computation. This gives me enough information to choose appropriate values for the tuning parameters discussed above: $d_u = 1$, $d_v = 1$, $c = 2$, and $d_k = 1$. Each invocation of the resulting kernel would execute $c^2 = 4$ mul-adds on data in five CPU registers, namely four entries from x, and one summation variable s. Note that it is inappropriate to evaluate $g(v, i)$ in fully-optimized code for the CPU kernel: the kernel should not be called at all unless $|g(v, i)| = c$ for all i in the range $c(u' - 1) < i \le c(u' + d_u - 1)$. A more-general "boundary case" code should be called if $0 < |g(v, i)| < c$ for any i; such cases will arise, infrequently, if the data is very-nearly optimally arranged. During debugging, however, I do include an evaluation of $g()$ in the kernel. If $|g(v, i)| < c$, then my kernel code will issue a message to a debugging trace.

Capacity parameter δ. It is impossible to write a memory-efficient code without considering capacity constraints in all layers. In my model, layer h in the memory hierarchy has capacity

$$C_h = RC_{h-1}^{\delta} \tag{8}$$

Layer $h = 1$ is the register layer: $C_1 = R$. Nominally, $\delta = 3/2$.

Note that I have defined $C_0 = 1$ and $C_{-1} = (1/R)^{1/\delta}$. These capacities do not correspond to any layer in memory, but they are useful in theoretical derivations later in this paper.

Layer	Name	Capacity
1	Register	8
2	Primary cache	181
3	Secondary cache	19484
4	RAM	2.2E+07
5	Disk	8.1E+11
6	Tape	5.9E+18

Table 1: Layer capacities for $R = 8$, $\delta = 3/2$.

Table 1 lists the nominal capacities in my memory hierarchy, in units of machine words. The technologically-aware reader will note that my nominal register and cache capacities are conservative estimates of the space that will be available to any running process on any workstation. At the time of this writing (late 1996), however, my nominal RAM capacity is not available to processes running on lightly-configured or heavily-loaded desktop systems. If you want to design code that will run well on such a system, I suggest you design somewhat more conservatively. For example, at $\delta = 1.4$ and $R = 8$, the RAM capacity in my model drops dramatically, to $C_4 = 2.5$ Mwords. (Of course, only highly-local algorithms will run efficiently if given such a small RAM footprint.)

My nominal model is also slightly futuristic with respect to disk capacities. Most desktop systems in late 1996 can provide only 10^9 words of disk space to a running process, not the 0.81×10^{12} words asserted in Table 1. Furthermore, somewhat less than 10^8 words is available in virtual memory on most systems, as typically configured; larger memory footprints would have to be "simulated" with some sort of API that "extends" virtual memory into the disk-based file system and even to a tape library, if present. Such an API is currently under development in my research group.

Readers who are familiar with exact models of cache and RAM sizes will no doubt be dismayed by the inaccuracies in my table. To such readers, I emphasize that the primary purposes of my model are to study the algorithmic implications of contemporary memory designs, to support portable code development, and to suggest a rationale for future designs of hierarchical memory systems. I am not so rash as to expect my simple model to be exact for any existing system.

Returning to our running example, if I design code to the nominal value $\delta = 1.5$, I would embed my kernel code in a loop nest that implements

Equation 7 for $d_u = d_v = 2$, $c = 4$, $d_k = 8$. Note that I am employing a recursive ("self-similar") coding design here: my kernel implements this same equation as its caller, but for smaller values of the tuning parameters. This is a key technique in third-paradigm coding, one that allows us to design (and analyze) our loop nest once for all layers of the memory hierarchy simultaneously.

Any computation defined by Equation 7 references at most cd_u distinct (and contiguous) rows in its left operand x_{ik}. Similarly, it references at most cd_v distinct, contiguous rows in its right operand x_{jk}. These bounds are tight in the case of data that is optimally arranged for computational density.

The row-lengths in a computation of Equation 7 are given by d_k. The computation thus produces one scalar result from two subarrays of x, one of dimension $cd_u \times d_k$ and the other of dimension $cd_v \times d_k$. The total amount of storage required to hold these operands is $c(d_u + d_v)d_k + 1$. The total number of arithmetic operations, assuming the data is arranged for optimal computational density, is $2c^2 d_u d_v d_k$. The re-use factor F for data held in a memory layer of size at least $1 + c(d_u + d_v)d_k$ during this computation is thus lower-bounded by $F \geq 2c^2 d_u d_v d_k/(c(d_u + d_v)d_k + 1) \approx 2cd_u d_v/(d_u + d_v)$. This lower bound is only tight if the computational operands are *not* re-used across evaluations of Equation 7 for different values of the location parameters u', v', and k'; and it is often easy, with the recursive structure of third-paradigm codes, to arrange for significant re-use. For example, if only v' is changed from one evaluation to the next, then all the x_{ik} operands are re-used.

I had suggested, a few paragraphs before, that I would call the CPU kernel from a loop nest with tuning parameters $d_u = d_v = 2$, $c = 4$, $d_k = 8$. This loop nest would define a cache-resident computation involving $2c^2 d_u d_v d_k = 1024$ arithmetic operations on any system providing at least $1 + c(d_u + d_v)d_k = 129$ words of cache data space to the currently-running process. The re-use factor F on primary-cache operands in our nominal system will thus be at least seven. You may be tempted to tune these parameters carefully, picking somewhat larger values of c, d_u or d_k to obtain maximal re-use. I would not recommend doing this. In my experience, small adjustments in tuning constants for third-paradigm codes rarely make a noticeable difference in runtime. If the constants are slightly too small, then you will have too much control overhead; if they are slightly too large, you won't notice it unless the memory layer you are overflowing is the performance bottleneck.

The next coding task is to design a recursive calling structure, evaluating Equation 7 for, perhaps, $d_u = d_v = 2$, $c = 2^{z+1}$, $d_k = 2^{z+2}$ at level $z \geq 2$ in the recursion. Note that these parameter choices give an eight-way recursion: we halve the span of the i, j, and k variables in the inner loop (Equation 7) with each recursive call. I say "perhaps" I would design the code in this way because, in the general case, computational density (controlled by $g(v, i)$ in our running example) can become very difficult to manage at higher levels of the recursion. For example, when I designed code to evaluate the Dreyfus-Wagner recurrence for rectilinear Steiner tree minimization, it seemed best to increase d_k much more rapidly than the other tuning parameters. By so doing, I obtained near-optimal computational density in higher layers of the recursion (*i.e.* at RAM and disk layers of memory) without running afoul of the sequencing constraints.

In my experience, data rearrangement for computational density is the most important, and by far the most mentally-challenging, activity for a performance programmer. The third-paradigm coding technique, and the supporting API under development in my research group, will allow programmers to concentrate their efforts on this intrinsically-difficult activity, instead of on "tuning" their code for specific computational platforms and loads.

Maximum space S. It is impossible to design an efficient large-memory code without some knowledge of the maximum space S available in your virtual memory system. Nominally, a typical desktop installation provides just $S = 10^7$ words of virtual memory, at the time of this writing. It is generally possible, at modest trouble and expense, to reconfigure a desktop operating system so that $S = 10^8$ or even $S = 10^9$ words of disk storage are allocated to the paging device.

The parameter S defines the boundary between the virtual memory system and the file system. A third-paradigm code must compare the value of S to the ending address p of a segment of a data structure, in order to determine whether this segment is held in virtual memory or in a data file. This comparison, along with a file-based and data-compressed extension to virtual memory, will be encapsulated in the API for third-paradigm coding currently under development in my research group. Once this API is developed, there will be no need for the sizing parameter S to appear in any third-paradigm application code. Note: our API will need a second sizing parameter S_2 to define the boundary between disk and tape storage, unless it is installed on top of a hierarchical storage management system.

As indicated above, in third-paradigm code, data structures are referenced by a scalar pointer p. If $p > S$, the data is stored in a file; if $p \leq S$, the data is stored in a one-dimensional memory array. The test $p \leq S$ appears in the prologue of the recursive procedure used to access the data structure. Also appearing in this prologue is a calculation of the effective address q of a localized copy of the data referenced by p. It is simplest to start by allocating one set of data buffers for each layer z of the recursion, with the buffers for layer z being allocated contiguously, and at higher addresses, to those of layer $z-1$. The data from p should explicitly be copied into q if the pointers p and q straddle a memory layer boundary C_h, that is if $q \leq C_h < p$ for some integer h. If there is no layer crossing, the buffers at q will not be used; therefore they should not be allocated, and the calculation of q for higher-indexed recursive calls should be adjusted accordingly. Note: by Equation 8, a data word at p will cross $O(\log_\delta \log_R p)$ layer boundaries, and thus be explicitly copied $O(\log_\delta \log_R p)$ times, before it is available for use in a CPU kernel.

Let us return to our running example to see how the explicit copy operations will work out in practice. As indicated previously, I would write an eight-way divide-and-conquer routine for this computational problem. The base ($z = 1$) case of the recursion calls my CPU kernel eight times, evaluating the sum $s(u', v', k')$ on various quadrants of two subarrays of x of dimension 8×8. The two subarrays would be referenced by offsets 64 and 128, respectively, into a one-dimensional array. The next ($z = 2$) layer of the recursion would reference two subarrays of dimension 16×16, referenced by offsets 384 and 640, respectively; these offsets are computed from adding subarray sizes to prior offsets, $16^2 + 128$ and $16^2 + 384$. An explicit copy operation would occur whenever the base case is entered because, under the nominal memory capacity parameters $\delta = 3/2$ and $R = 8$, a 640-word data structure would reside in layer 3 of memory (secondary cache) but a 128-word data structure would reside in layer 2 (primary cache).

Of course, all CPUs are designed to load their caches transparently. Explicit copy operations for loading cache need not appear, and arguably should not appear, in any source code. However, if you write code without explicit copies, you run the risk of encountering associativity conflicts in cache. You also run the risk of "data sloshing" in shared-memory multiprocessing systems, if your code is ever parallelized with insufficient attention to low-level details of cache and data structure. If your data structures are not sufficiently compact, then explicit copies may in any event be required to obtain cache locality. For example, in our running example, the 8×8 array subsections are not compactly allocated in a row-major $n \times n$ array in C or in a column-major $n \times n$ array in Fortran. Finally, if the segments are sufficiently small and adjacent, several data structure segments may be traversed in an unrolled loop with a single pointer and constant offsets, leading to highly efficient kernel codes on some CPUs. For these reasons, among others, I recommend that explicit copies be performed across layer boundaries in my model, except in codes that are highly tuned for specific systems and, perhaps, in codes without data reuse.

It is usually easy to analyze the movement of data, and thus to estimate performance, in a third-paradigm code. For example, let us focus on the two 8×8 subarrays copied into primary cache at level $z = 1$ of the recursion. My recursive control structure reuses each of these 8×8 subarrays twice during the eight recursive calls made from level $z = 2$ of the recursion. At level $z = 3$, each 8×8 subarray is referenced four times during sixty-four evaluations of the inner-loop nest. The secondary cache in our nominal system is large enough to hold two 64×64 subarrays at level $z = 4$; each 8×8 subarray has a reuse factor of $64/8 = 8$ at this level. Roughly, then, $7/8$ of the primary cache misses will be satisfied by our secondary cache.

A similar analysis will reveal very high hit rates in RAM for data being loaded into secondary cache, and extremely high hit rates on disk (instead of tape) whenever data is loaded into RAM. The recursive copying operations of third-paradigm code will expose and exploit whatever temporal and spatial data locality is present in the underlying algorithm.

Bandwidth parameter γ. It is impossible to write an efficient large-memory code without some knowledge of latencies and bandwidths in the memory hierarchy. In my model, a (stride-1) block transfer of w words between layer h and any lower-indexed layer requires time $T(w, h) = L_h + wG_h$, where L_h is the latency and G_h is the "cycle time" between words in a block-transfer, with

$$L_h = (C_{h-1}/R)^{\alpha} \tag{9}$$

$$G_h = (C_{h-1}/R)^{\gamma} \tag{10}$$

and

$$\alpha = 1/\delta + \gamma \tag{11}$$

I require $\gamma \geq 0$. Nominally, $\gamma = 1/3$ and $\alpha = 1$. Note that $\alpha \geq 0$, because of my prior assumption that $\delta \geq 1$.

The reader may wonder why I haven't defined latency and cycle-time in terms of C_h, rather than C_{h-1} as above. My reasoning is that that, in some cases and within limits, it is possible to add capacity to a memory layer without affecting its latency or cycle time. For example, one can add a second disk drive or more DRAM chips to most PCs without affecting latency or cycle time. It seems natural, then, to characterize a memory technology by its "minimum distance" to the CPU. In my parameterization, this minimum distance is equal to the capacity of the preceding layer of memory: the minimum distance of a storage cell in layer h is equal to the "maximum distance" C_{h-1} of any storage cell at layer $h-1$.

There is a "natural blocksize" B_h for transfers into layer h from a lower-indexed layer, defined as follows:

$$B_h = G_h/L_h \tag{12}$$

Note that B_h is the size of a transfer for which the latency L_h to start the transfer is numerically equal to the total cycle time $B_h G_h$ of all words involved in the transfer. This is an important point of design. If all transfers are of blocksize at least B_h, then the computation will not be latency-bottlenecked even if the CPU is stalled for the entire latency period L_h.

In some contexts, it is convenient to model natural blocksizes directly with the rule

$$B_h = (C_{h-1}/R)^{\beta} \tag{13}$$

where

$$\beta = \alpha - \gamma \tag{14}$$

With Equation 11, this implies $\beta = 1/\delta$.

Above, I have defined four power-law exponents α, β, γ and δ. Their values are linked by Equations 11 and 14. Sometimes it is convenient to think of α and γ as the independent variables, describing the latency and bandwidth of a memory hierarchy. In this view, β and δ are dependent variables, describing natural blocksize and capacity.

The relationships between the power-law exponents support the following lemma. The total size C_{h-2} of layer $(h-2)$ is equal to the size of the blocks on layer h:

Lemma 1 *If $R \geq 1$ and $\beta > 0$ then, for all h, $B_h = C_{h-2}$.*

Proof. The capacity of layer $h-1$ is $C_{h-1} = RC_{h-2}^{\delta}$. We have $C_{h-2} = (C_{h-1}/R)^{1/\delta}$, and $C_{h-2}G_h = (C_{h-1}/R)^{1/\delta+\gamma} = (C_{h-1}/R)^{\alpha} = L_h$. Thus $C_{h-2} = L_h/G_h = B_h$. □

I call this the "swapping lemma" because it suggests that desktop systems are designed to load all of secondary cache from a single disk fetch, and all of RAM from a single tape access. That is, the memory layers are as large as possible (maximizing δ), subject to the constraint that the complete context of a running job can be swapped efficiently ($\delta \leq 1/\beta$, or equivalently, $1/\delta \geq \alpha - \gamma$).

Note: it is possible to design systems with $\delta \neq 1/\beta$, that is to say, systems that violate Equation 11. Indeed this may be necessary from time to time, due to technological constraints. My assumption that $\delta = 1/\beta$ is based on observations of existing desktop systems and on algorithmic considerations, one of which is outlined below.

In a workstation with $\beta = 2/3$, if δ were much greater than $1/\beta = 3/2$, or smaller than $4/3$, then the system would not be able to transpose a square matrix in two passes. Here is a sketch of the proof. Apply the lower-bound argument from the two-layer model of Vitter and Shriver [8], for the case of problem size $N = RM^{\delta}$, memory capacity M, parallelism $P = 1$, blocksize $B = (M/R)^{\beta}$, and my model parameters R, β and δ. The number of passes made by any algorithm for transposing a square matrix is at least $\lceil \log_{M/B} \sqrt{N} \rceil$, if $B \leq \sqrt{N}$. (If $B > \sqrt{N}$, then $\lceil \log_{M/B}(N/B) \rceil \geq \lceil \log_{M/B} \sqrt{N} \rceil$ passes are required.) When $1/\beta = 3/2$, the constraint $B \leq \sqrt{N}$ is equivalent to $(M/R)^{4/3} \leq RM^{\delta}$, which is satisfied for $\delta > 4/3$ and $R > 1$, but is not satisfied if $\delta < 4/3$ and R is small.

Because $B \leq \sqrt{N}$, two passes will suffice if $\lceil \log_{M/B} \sqrt{N} \rceil \leq 2$, that is, if $(M/B)^4 \geq N$. Substituting $\beta = 2/3$, we find the constraint $M^{4/3}R^{8/3} \geq RM^{\delta}$. Simplified, this is $R^{5/3} \geq M^{\delta - 4/3}$. For $\delta = 3/2$ and $R = 8$, this reduces to the requirement that $2^5 \geq M^{1/6}$ or $M \leq 2^{30}$. We conclude that a two-pass transpose operation on a gigaword matrix is (barely) feasible on a typically-configured workstation. Larger matrices will require three passes. Alternatively, two-pass transposition is possible for multi-gigaword square matrices on a robustly-configured workstation, that is, one with $R > 8$ or smaller β (and thus larger δ).

Another implication of $\delta = 1/\beta$ is that the natural blocksizes B_h in my model grow with the same parameter δ as do the layer capacities C_h: $B_h = RB_{h-1}^{\delta}$ and $C_h = RC_{h-1}^{\delta}$. Only the starting points differ: $C_1 = R$ and $B_1 = 1$. This relationship between blocksize and layer capacity is reminiscent of the fixed "aspect ratio" of the HMM and UMH models [3], yielding similar computational properties.

Table 2 lists the latencies and bandwidths of the layers in a memory hierarchy with nominal parameter values. The latencies are scaled to the natural timebase of a desktop system, the frequency of the CPU clock, nominally 200 MHz. Bandwidths are defined as $1/G_h$, scaled to the CPU clock and to an assumed wordlength of 8 bytes. Note that the definition of L_h and G_h guarantees that, for any positive value of R, δ, and γ, transfers into primary cache from registers, or from registers to primary cache, occur with one clock of latency and at cycle time of one clock per word. That is, the

Layer	Name	Latency	Natural Blocksize	Bandwidth
1	Register			
2	Primary cache	5 nsec	1 word	1.6 GB/sec
3	Secondary cache	113 nsec	8 words	570 MB/sec
4	RAM	12 usec	181 words	120 MB/sec
5	Disk	14 msec	20K words	11 MB/sec
6	Tape	500 sec	22M words	0.3 MB/sec

Table 2: Latency, blocksize, and bandwidth of transfers between layer h and lower-indexed layers, for $\gamma = 1/3$, $\delta = 3/2$, $R = 8$, on a 200 MHz computer with 8-byte machine words.

natural blocksize for these transfers is $L_2/G_2 = 1$ word in almost all desktop systems. Only the most aggressively designed CPUs can transfer two words to register per clock, and almost all desktop systems can achieve one word per clock on some CPU kernels. Also, vectorized CPUs are rarely found in desktop systems at the time of this writing, so each transfer from primary cache to register consists, almost universally, of a single word. (*N.B.:* The Pentium Pro provides vectorized operations on subword values, typically used to represent pixels or voxels, if these are packed in a single word in a CPU register.)

The technologically-aware reader will note that the bandwidths in Table 2 are fairly accurate in all cases except the disk, but some latencies may seem too high. For example, it is not always necessary to transfer 181 words from RAM to secondary cache to avoid a latency bottleneck. However, please note from Table 1 that my RAM layer contains tens of millions of words. A reference to an arbitrary page in RAM is liable to cause a TLB fault, that is, to encounter a few microseconds of latency while some virtual memory translation tables are loaded. So, whenever possible, I try to reference blocks of a few hundred words in each RAM reference, so that the latency of the TLB fault is small in comparison to the total transfer time.

A latency of 500 seconds for tape-to-disk transfers is very generous, allowing time for manual loading from tape libraries if desired.

A disk bandwidth of 11 MB/sec may seem somewhat high, but is often acheivable on low-cost systems if we use data compression. The bandwidth figures refer to uncompressed data; with 5.5:1 compression, even a 2 MB/sec IDE disk drive can indeed deliver 11 MB/sec of uncompressed data to RAM on a sustained basis. If compression is infeasible or inappropriate, several SCSI disk drives must be used in tandem to sustain 11 MB/sec of disk bandwidth in a desktop computer.

Of course, data compression and decompression routines require CPU time, so the use of compression on disk transfers is not appropriate unless the disk bandwidth is indeed a performance bottleneck that, if eased, offers a payback on coding effort and future software maintenance cost. This reflects another design goal of my API for third-paradigm coding: choices in data compression subroutines, and in their associated data structures, should be orthogonal to the main structure of the underlying code. An orthogonal design makes it easy to enable or disable data compression, and even to apply it to differing layers in the memory hierarchy, when appropriate.

5 Algorithmic Claims

Some years ago, I wrote a third-paradigm code for matrix-matrix multiplication: it ran at near-peak speed on a Sun-4 workstation. It also behaved very well under heavy system load, for example, when three instances of my code were running concurrently on matrices too large to fit in RAM, the system was still CPU-bottlenecked.

In unpublished work, I have analyzed the performance of third-paradigm code designs for the following problems: matrix transposition, matrix-matrix multiplication, and finding duplicates. I have included some notes on these analyses below.

Matrix-matrix multiplication is an interesting case. A CPU-bottlenecked computation is possible when $\gamma \leq 1/3$. Otherwise the computation, even if optimally blocked, is bottlenecked on memory latency. Curiously, as noted in the previous section, the convergent desktop system has $\gamma = 1/3$, that is, it provides just enough bandwidth to support efficient matrix-matrix multiplication. It would seem that the desktop computer is designed to run large matrix-matrix multiplication problems, or at least to perform well on LINPACK benchmarks!

Matrix transposition is another interesting case. As observed in the previous section, it is a very tight fit to the $\delta = 3/2$ observed (see Table 2) on a common desktop system, if (as assumed in my model) $\beta = 1/\delta$. A straightforward two-pass algorithm suffices in this case, even when the matrix is square and fills half of some layer in memory. (A more complicated "in-place" algorithm would be appropriate if the matrix is slightly larger, that is, if the matrix dimension n were in the range $\sqrt{C_h}/2 < n \leq \sqrt{C_h}$ for some h, and if you were confident that this range of n were important enough to deserve special consideration for some particular system and system load.)

In the first pass of the transposition of an $m \times n$ matrix, with $m \geq n$, data is read in blocks of size C_{h-2} and scattered into $y = C_h/C_{h-2} - 1$ buffers. Here h is the smallest integer such that $2mn \leq C_h$. Each of the y buffers will receive the data for a $C_{h-2}y/n \times n/y$ submatrix. These submatrices should be transposed with a recursive call to the transposition routine; note that this recursive call will require data movement only on layers $0, 1, \cdots, h-1$ of the memory hierarchy. In the second pass, transposed data from the y buffers on layer h is reassembled into an $n \times m$ matrix.

It seems that a third pass is required for the transposition of large matrices on systems with $\delta = 1.4$, if $\beta = 1/\delta$. The number of passes rises to a small constant greater than three for some $\delta < 1.4$; but for $\delta = 1$ two passes suffice. I intend to make a fuller study of the interaction between δ and the maximum number of passes required for matrix transposition in the very near future.

The duplicate-finding problem I consider in this report is a very simple one, designed to illustrate the analysis of a latency-limited computation. The input to a duplicate-finding routine is a series v_i of key values; the output is a series y_i of booleans. The output y_i should be true iff there exists $j < i$ such that $v_j = v_i$, that is, $y_i = 0$ unless the key v_i has occurred previously in the input. In the on-line version of this problem, analyzed here, the output y_i must be produced before the next input v_{i+1} is read. I assume there are at most $m \leq C_h/2$ distinct keys in the complete input sequence v, for some layer h in the memory hierarchy.

A simple third-paradigm code for the duplicate-finding problem would insert the first $C_2/2$ unique values into a hash table of size C_2, the next $C_3/2$ unique values into a hash table of size C_3, etc. The largest table should have room for the remaining $m - \sum_{j<h} C_j/2$ elements at a load factor of 1/2. With nominal memory performance parameters $\delta = 1/\beta = 3/2$, $\gamma = 1/3$, and $R = 8$, the latency L_j of a transfer from layer $j+1$ to layer j is linearly related to its capacity: $L_j = (C_j/R)^{\beta+\gamma} \leq C_j/4$. On average, then, a well-designed hash probe algorithm would take at most $2\sum_{k\leq j} L_j + O(1) = O(C_j)$ time for a probe into our hash tables of size C_1, C_2, ..., C_j. If the probe is successful, we are done processing an input v_i. If the probe is unsuccessful, and the table of size C_j is not full, we must insert a new entry at time cost L_j. If the table of size C_j is full, then we must probe the table of size C_{j+1}.

Let us assume that the input values v_i are Zipfian random variates, and that enough distinct inputs have already been processed that our table of size C_j is full. Then the probability $P(C_j)$ of an unsuccessful search into our hash table of size C_j can be bounded as follows: $P(C_j) \leq \ln C_j / \ln C_h$. (*N.B.* I will prove this assertion in a later version of this report.) This is a convenient, albeit loose, upper bound on the probability that our algorithm will probe our hash table of size C_{j+1}, for $j < h$, while searching for a match to a Zipf-law input v_i, for any i.

Our largest hash table will never overflow. We will search this table with probability $P(m) \approx 1 - \ln C_{h-1}/\ln C_h = 1 - \beta \ln(C_h/R)/\ln C_h \approx 1 - \beta = 1/3$. That is, for Zipf-law inputs, our hierarchical coding strategy has achieved no appreciable speedup over a naive code that maintains a single hash table capable of holding m elements. Both codes require $O(m)$ time to process each input, making $O(1)$ probes into the h-th layer of memory.

I conjecture that my $O(m)$-time duplicate-finding code is optimal for computers with $\alpha = 1$, within a logarithmic factor for Zipf-law inputs. The x-th most probable Zipfian input value occurs with frequency $1/(x \ln m) + O(1/x)$. Let us call this input value k_x, bearing in mind that the relative frequency x is generally not deducible from the value k_x. An arguably-optimal but generally-infeasible code would keep track of occurrences by using a different boolean flag for each input value. The most-probable inputs should have the lowest addresses; indeed, in an optimal code, the flag b_x for the value k_x will be stored at memory location x and thus will be accessible in $O(x^\alpha)$ time. If, upon the event that the i-th input y_i has the value k_x, the relative frequency x is somehow deduced and then b_x is accessed in just $O(x^\alpha)$ time, then the average time per input would be $\sum_{1\leq x\leq m}(1/(x \ln m)) + O(1/x))O(x^\alpha) = O(m/\ln m)$. It is an open problem whether an $\Omega(m)$ lower bound can be proven, or for that matter, to prove an $\Omega(m/\ln m)$ bound for all possible codes. The lower-bound argument sketched above is only applicable to codes using a boolean flag to keep track of input values in a naive way.

The analytically-inclined reader will have noticed that our hierarchical, third-paradigm hashing code will run at comparable speed to the naive code for any input distribution that is less "spatially local" than the Zipfian distribution assumed in our analysis above. If the inputs have more than Zipfian locality, then our hierarchical code will run significantly faster. Also, Zipfian locality is sufficient to give speedups to third-paradigm codes over naive codes, on computers with latency parameter $\alpha < 1$.

It is an open problem to analyze the runtime of an offline version of my simple duplicate-finding problem. For example, we might allow output y_i to be produced at any time before input v_{i+m} is read. We may also require that the outputs be produced sequentially: output y_i must be produced before output y_{i+1}. I conjecture that hierarchical third-paradigm algorithms for offline duplicate-finding will show significant speedups over naive codes, even on Zipf-law inputs.

It would be interesting to analyze and develop third-paradigm codes for more realistic problems involving hashing. In most real-world applications, key distributions are not stationary, and keys must be deleted.

6 Conclusions and Future Work

I have sketched a new paradigm for large-memory programming, to sidestep the capacity and performance constraints of virtual memory, and to avoid writing complicated codes with many tuning parameters. My coding scheme is based on a four-parameter model of memory performance, so, at least in principle, all tuning parameters in such codes could be expressed as functions of constants in the problem statement and my four memory-performance parameters. In practice, performance programmers will not be so disciplined in their use of constants, and some system-specific optimizations are surely appropriate. But a simple model of performance, and a standardized coding method, would surely be helpful to performance programmers as well as to compiler writers.

In future work, I intend to develop an Application Programmer's Interface, or API, to simplify the task of developing third-paradigm codes. In particular, it would be very helpful to have a suite of buffer-management routines as well as some standard, recursively-defined data structures such as two-dimensional arrays that can be efficiently accessed by quadrant.

I also intend to extend my four-parameter model to some limited forms of parallelism in memory and processing. The approach I have in mind at present is to build a performance model for the data-parallel style of programming. It may suffice to have a single parameter P for available parallelism at all layers, from register to tape, although I strongly suspect that most supercomputers are not configured so robustly. It is probably more accurate to assume that the available parallelism falls off as one goes deeper into the memory hierarchy. For example, a 1024-processor CM-5 or T3E may be configured with only ten or twenty disk channels.

References

[1] A. Aggarwal, B. Alpern, A.K. Chandra, and M. Snir. A model for

hierarchical memory. In *Proc. of the 19th Annual ACM Symp. on Theory of Computing*, pages 305–314, May 1987.

[2] A. Aggarwal, A.K. Chandra, and M. Snir. Hierarchical memory with block transfer. In *Proc. of the 28th Annual IEEE Symp. on Foundations of Computer Science*, pages 204–216, October 1987.

[3] B. Alpern, L. Carter, E. Feig, and T. Selker. The uniform memory hierarchy model of computation. *Algorithmica*, 12:72–109, 1994.

[4] R.W. Floyd. Permuting information in idealized two-level storage. In R. Miller and J. Thatcher, editors, *Complexity of Computer Calculations*, pages 105–109. Plenum, New York, 1972.

[5] D. Thiebaut. On the fractal dimension of computer programs and its application to the prediction of the cache miss ratio. *IEEE Trans. Computers*, 38(7):1012–1026, 1989.

[6] Clark Thomborson, Bowen Alpern, and Larry Carter. Rectilinear Steiner tree minimization on a workstation. In N. Dean and G.E. Shannon, editors, *Computational Support for Discrete Mathematics*, volume 15 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 119–136. American Mathematical Society, 1994.

[7] Darren Vengroff. A transparent parallel I/O environment. In *Proc. DAGS Symposium*, 1994. `ftp://cs.duke.edu:pub/dev/papers/tpie-dags94.ps.Z`.

[8] J.S. Vitter and E.A.M. Shriver. Algorithms for parallel memory, I: Two-level memories. *Algorithmica*, 12:110–147, 1994.

[9] J.S. Vitter and E.A.M. Shriver. Algorithms for parallel memory, II: Hierarchical multilevel memories. *Algorithmica*, 12:148–169, 1994.