Systems Research in the Age of On-Line Coffee Houses*

Richard Zippel

TR 94-1419
April 1994

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

* This work was supported by the Advanced Research Projects Agency of the Department of Defense under ONR Contract N00014-92-J-1989, by ONR Contract N00014-92-J-1839 and by a grant from the United States-Israel Binational Science Foundation.

Systems Research in the Age of On-line Coffee Houses

Richard Zippel
Dept. of Computer Science
Cornell University, Ithaca, NY 14853

For years, we have spoken of a golden era where high performance "supercomputers" will be available in local department stores and we will be able to communicate with anyone and any organization we want via computer networks. In the past, we have whimsically said that in this era, computing systems will be used in dramatically different fashion than they are today. That era is here now, with 60–80Mips computers available in local computer shops, "on-line" coffee houses springing up in trendy neighborhoods and National Public Radio broadcasting on the Internet for almost a year. And yet, the way we use computers has not changed dramatically.

Clearly, improved speech understanding and generation systems, vision systems and other sophisticated I/O technologies will change our interaction with computers. What will change the content of our interaction with computers? What type of systems research will enable revolutionary changes in the use of computers?

In an attempt to provoke discussion about these topics, a talk was presented several times during the spring semester 1994. A slightly different approach to systems research was presented as well as a few new directions that are being undertaken at Cornell. Hopefully, the ideas and questions raised by this presentation will be of use to others.

This technical report is rather rough and disorganized, consisting as it does of slides from those talks and textual commentary. I felt it more valuable to make the material available in a timely manner than to wait until the details had been worked out, and the prose polished. It is an experiment and I am interested in the reaction any readers have to this form of presentation. You can also view a World Wide Web version of this document in:

http://simlab.cs.cornell.edu/slides/systems/overview.html

I am interested in any comments you might have, both on the content of this report and how it is presented. Please send them to rz@cs.cornell.edu.

This work was supported by the Advanced Research Projects Agency of the Department of Defense under ONR Contract N00014–92–J–1989, by ONR Contract N00014–92–J–1839 and by a grant from the United States-Israel Binational Science Foundation.

SYSTEMS RESEARCH IN THE AGE OF ON-LINE COFFEE HOUSES

RICHARD ZIPPEL

DEPARTMENT OF COMPUTER SCIENCE
CORNELL UNIVERSITY

Twenty or more years ago, researchers fantasized about a future populated by awsomely fast (and cheap) computers. Computers would be well connected, both to each other and to the world. They would relieve us of the drudgery in our lives like cleaning and maintaining our houses. Difficult financial decisions like purchasing cars and homes would be mediated and optimized by computers. They would make vast stores of information available in a fashion that doesn't overwhelm, but encourages further exploration. And electronic models constructed of the works of our great thinkers would allow us to study their thoughts and actions in ways hitherto unimagined. We would, in effect, be able to interact with the great minds of our time and of past times.

We were sure these machines would change our world for the better, and would make the science fiction of our fantasies no fiction. Computers would change our lives. Today computers have reached and even exceeded the performance and price levels required of our earlier fantasies—and the changes they have wrought fall far short of our desires.

This work was supported by the Advanced Reseach Projects Agency of the Department of Defense under ONR Contract N00014–92–J–1989 and by ONR Contract N00014–92–J–1839 and by a grant from the United States-Israel Binational Science Foundation.

US Technology Road Map

Year	1995	1998	2001	2004	2007
Feature Size (microns)	0.35	0.25	0.18	0.12	0.1
Gates/chip	800k	2M	5M	10M	20M
DRAM bits/chip	64M	256M	1G	4G	16G
SRAM bits/chip	16M	64M	256M	1G	4G
Chip size (mm2) Logic/processor	400	600	800	1000	1250
Chip size (mm2) DRAM	200	320	500	700	1000
Interconnect levels (logic)	4–5	5	5–6	6	6–7
Max power (W/die)	15	30	40	40–120	40–200
Number of I/Os	750	1500	2000	3500	5000
Performance (MHz) Off-chip	100	175	250	350	500
Performance (MHz) On-chip	200	350	500	700	1000

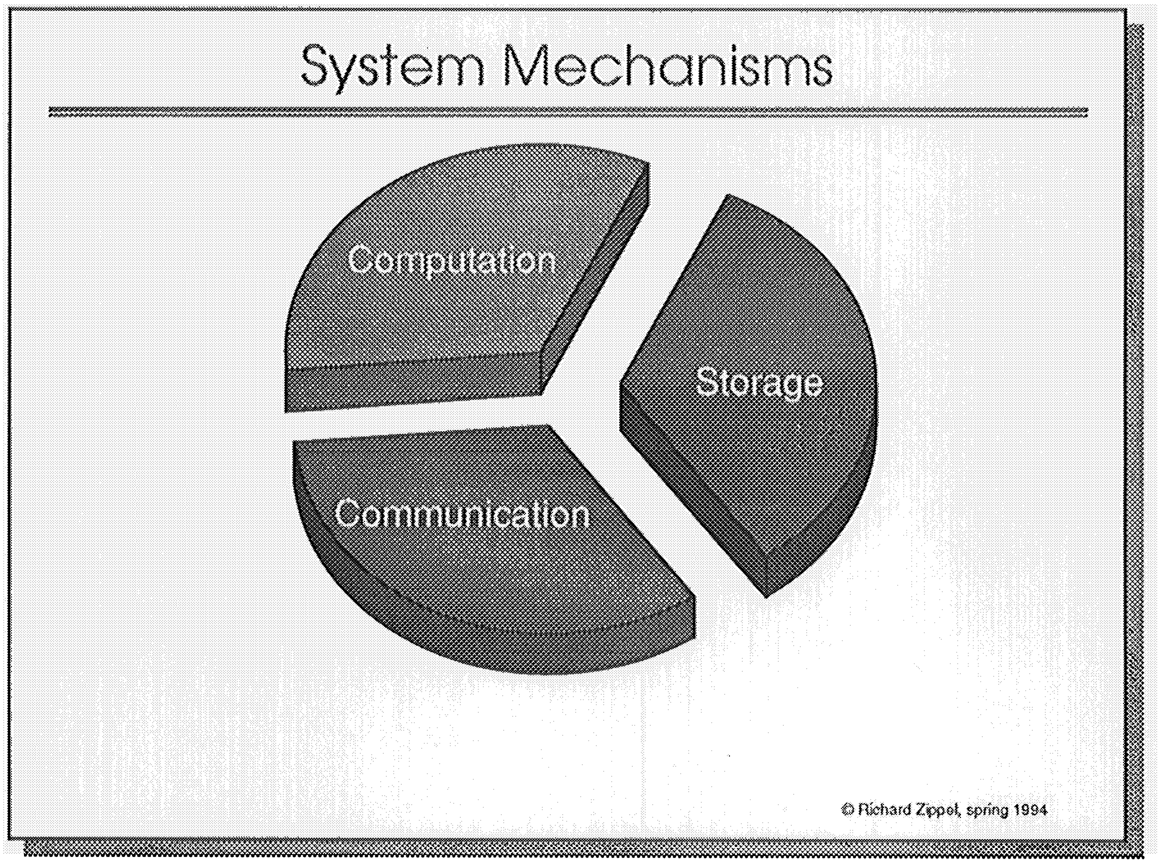
Source: Semiconductor Industry Association

While today's computers are fast and cheap, tomorrow's computers will be even more so. This chart summarizes the semiconductor industry's projection of the types of chips that will be produced over the next ten years. Computers built with these devices will greatly exceed even our wildest dreams.

And yet the world we thought these machines would enable has not arrived. Although the quality of some of our tools have improved, we still use computers in much the same way as twenty years ago. There have been some radical innovations, e.g. spread-sheets, but they are few and far between. We have failed to invent the new modes of computer usage that will change and improve society.

To a large degree we are elaborating on the systems ideas developed in the past, and studying problems that arose when creating systems years ago. Many of the constraints and problems that arose in these earlier days can now be resolved by brute force or can be sidestepped by using newer technologies. For instance, the Global Position System can now be used to guide cars around cities, or ships at sea eliminating the need for some other forms of navigation. In a sense we are guiding the direction of computer science research with our eyes firmly fixed to the rearview mirror.

By rethinking the foundations on which systems research is based, I believe we can free ourselves the pidgeon-holed problems that arose from earlier research, free ourselves to ask new, more revealing questions and free our imagination to use computing and information technology to better our lives and society.

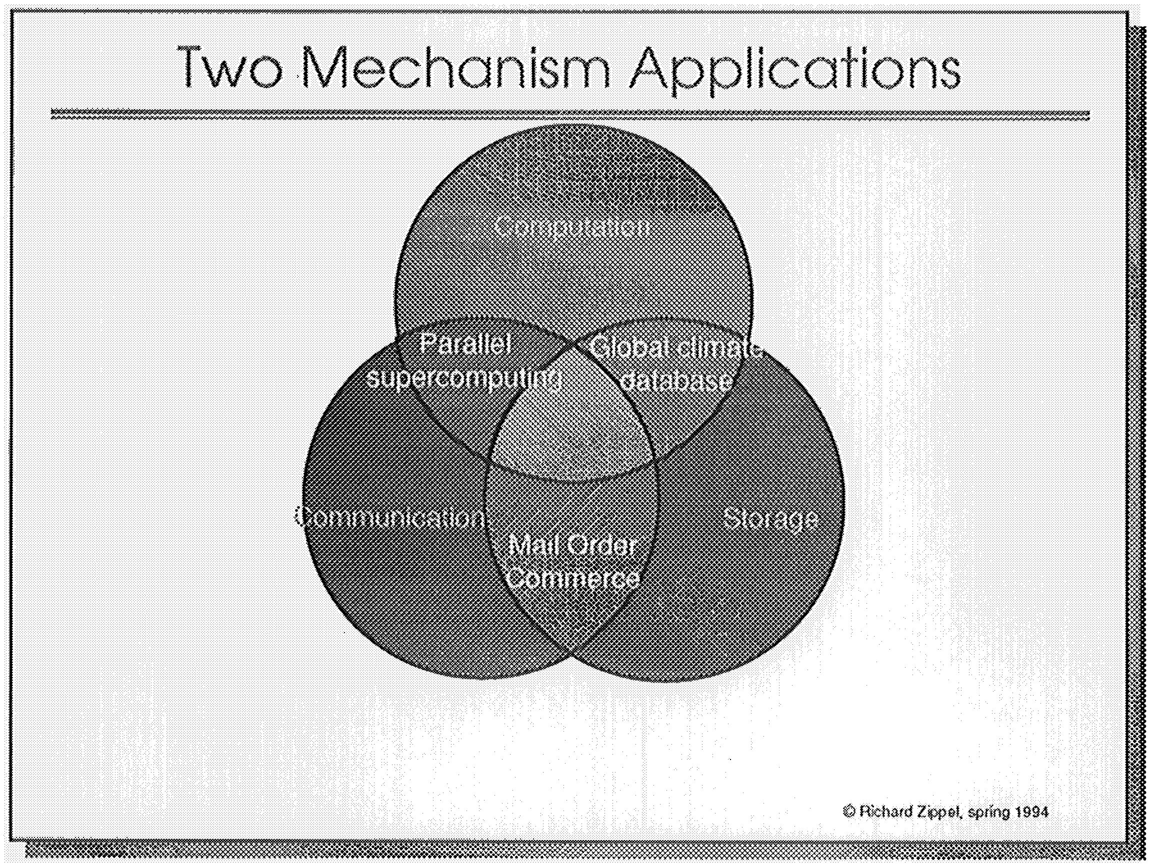


Systems work is traditionally divided into research areas such as operating systems, programming languages, networks, etc. Instead we will look at systems as composed of three different *mechanisms*: communications, computation and storage. Each of these mechanisms impacts decisions made in operating systems, languages etc.

Although no application involves just one of these mechanisms, the earliest applications predominantly tended to use one mechanism. For instance, the early telephone systems essentially only involved communications. In addition to its commercial successes, communications research has produced great advances in switching technologies, data compression and data correction, to name a few areas.

The simplest, and easiest to create database applications are centered primarily on storage issues. They provide efficient ways of storing and accessing information about salaries, population statistics and the books in a library, and particular patient records in a hospital. Finding data records coresponding to combinations of specified properties is difficult and lead a great deal of research in indexing, storage models etc. But by and large these activities are focussed on storage. Relatively small computational and communications was required.

Similarly, many scientific computations rely solely on computation—storage and communications issues are a tiny part of the effort required.

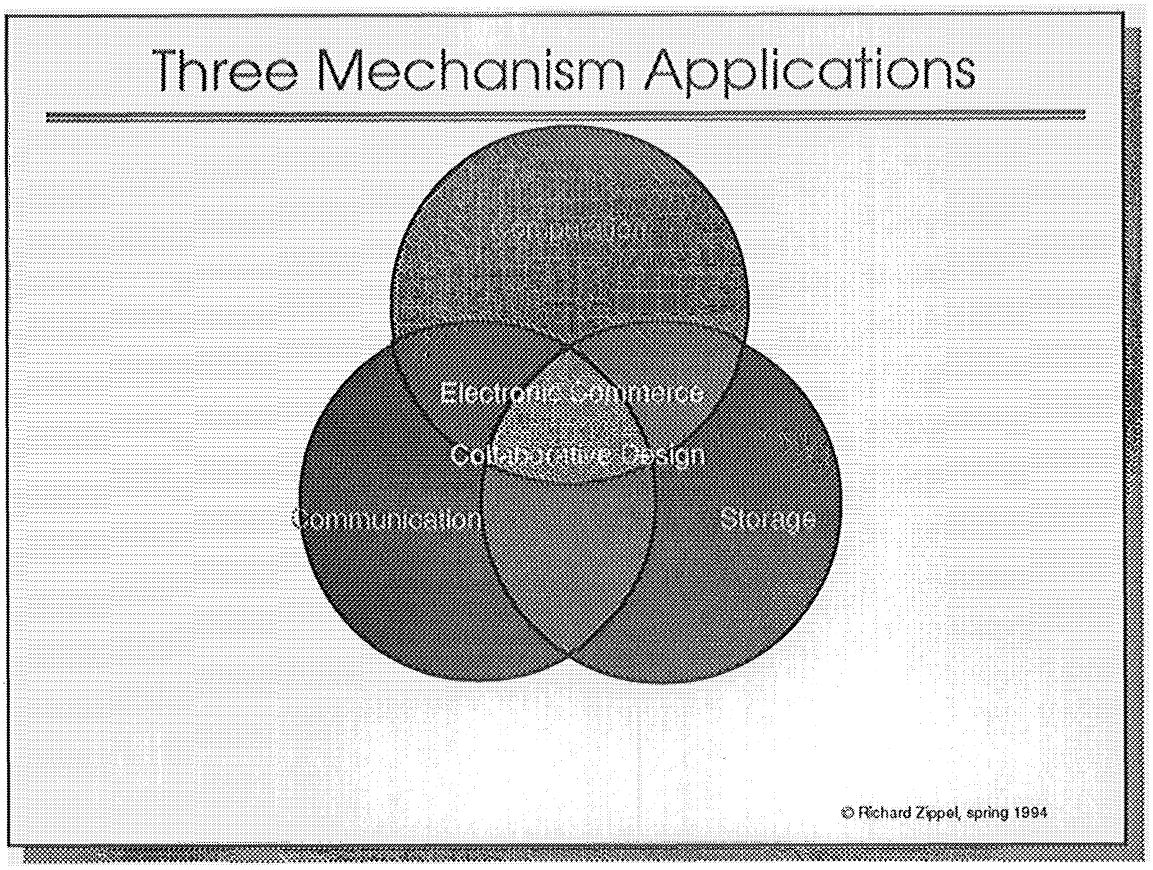


The next generation of applications blends together two different mechanisms. We are just starting to understand the problems that arise with these types of systems, what the proper abstractions are, and how to construt them.

For instance, climate modeling couples gigantic weather databases generated from satellite observation with the computation prowess of super-computers. Without super-computers to process the satellite imagery and perform high level searches and correlations, the satellite data would be of limited usefulness. The raw data provided by the satellites is the starting point for the super-computing computations used to predicate weather patterns. The two components, massive data storage and super-computing together permit accurate weather forecasting and climate modeling.

Combining computing with communications we have parallel computing technology. One of the reasons that parallel super-computers have been so slow in coming is that is so difficult effectively couple communications and computation issues. The first successful attempts at parallel machines used very stylized communications approaches, e.g. vector computing.

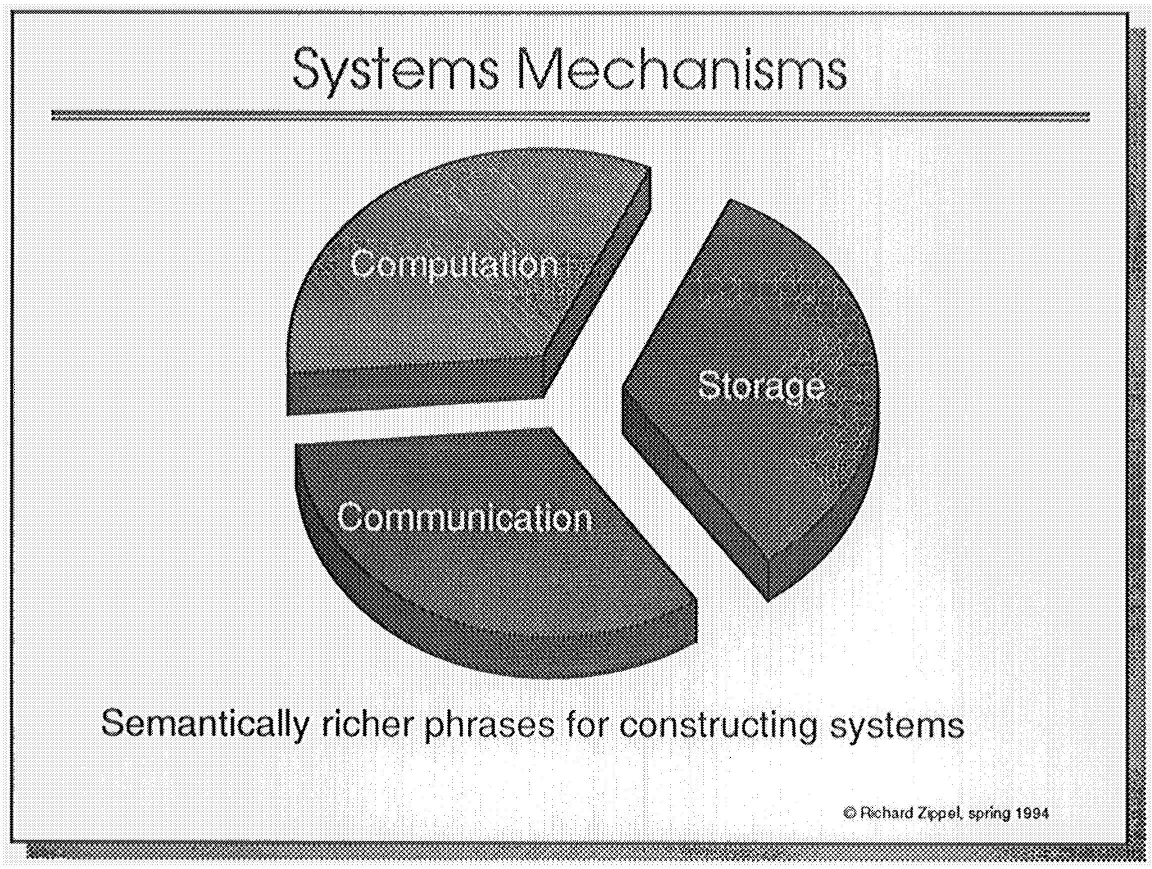
A typical coupling of communications and storage technologies is represented by the huge mail-order businesses that have sprung up in recent years. Without that combination of the telephone communications systems, the electronic distributed electronic databases and Federal Express they could not exist. Similarly the on-demand video systems actively being discussed merge (cable) communications with the huge continuous media databases that serve as movie repositories.



I believe that the next generation of applications — those that will begin to take advantage of the revolutionary changes in computing, communications and storage hardware and will revolutionize our lives — will be those applications that rely on all three of these mechanisms equally.

The example of such an application that is most immediately at hand is collaborative design, such as when developing a new automobile or aircraft. A number of designers will be involved in the project, scattered throughout the company and its sub-contractors. Cost and performance will be continuously optimized via simulations that predict the behavior of the design before it is fabricated. The results of the simulations, as well as the detailed part descriptions, manufacturing plans and schedules, marketing data and advertising plans are an enormous distributed database. These tools will allow the collaborative design system to identify inconsistencies and inadequencies in the design as they arise, so that they can be dealt with immediately rather than as a "fix-up" after manufacture. We have the standard issues of concurrency control and data storage, since several designers will be working on the design concurrently.

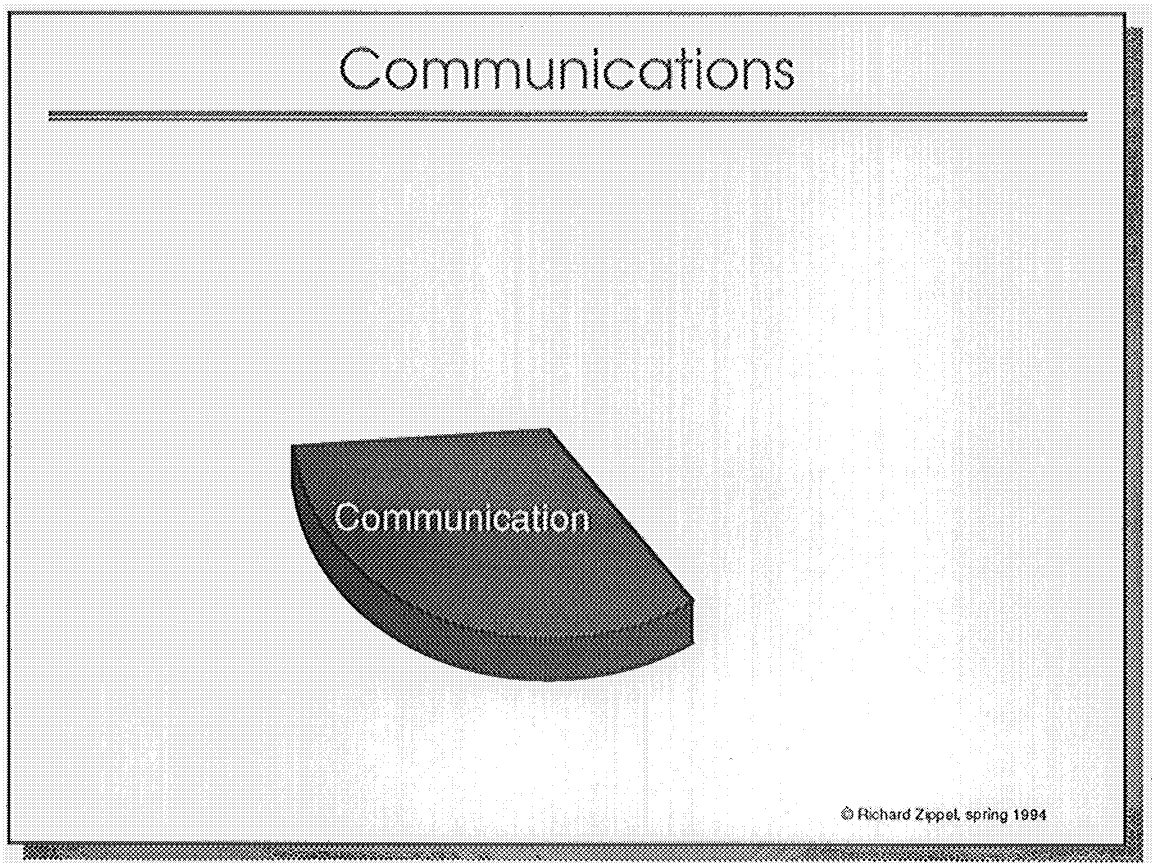
Basic teleconferencing systems have been available for several years. To a great extent these are expanded telephone systems that are able to handle the larger bandwidth requirements of real-time video. Enhanced teleconferencing systems will allow us to use teleconferencing to mediate meetings between several organizations scattered around the world. Image processing and vision tools will allow us to track objects of interest in the video scene, e.g. always having the speaker properly positioned in the image, along the the prop s/he is referring to. The need to record previous meetings, and to call up previous presentations in the middle of a meeting, places strong storage requirements on the system.



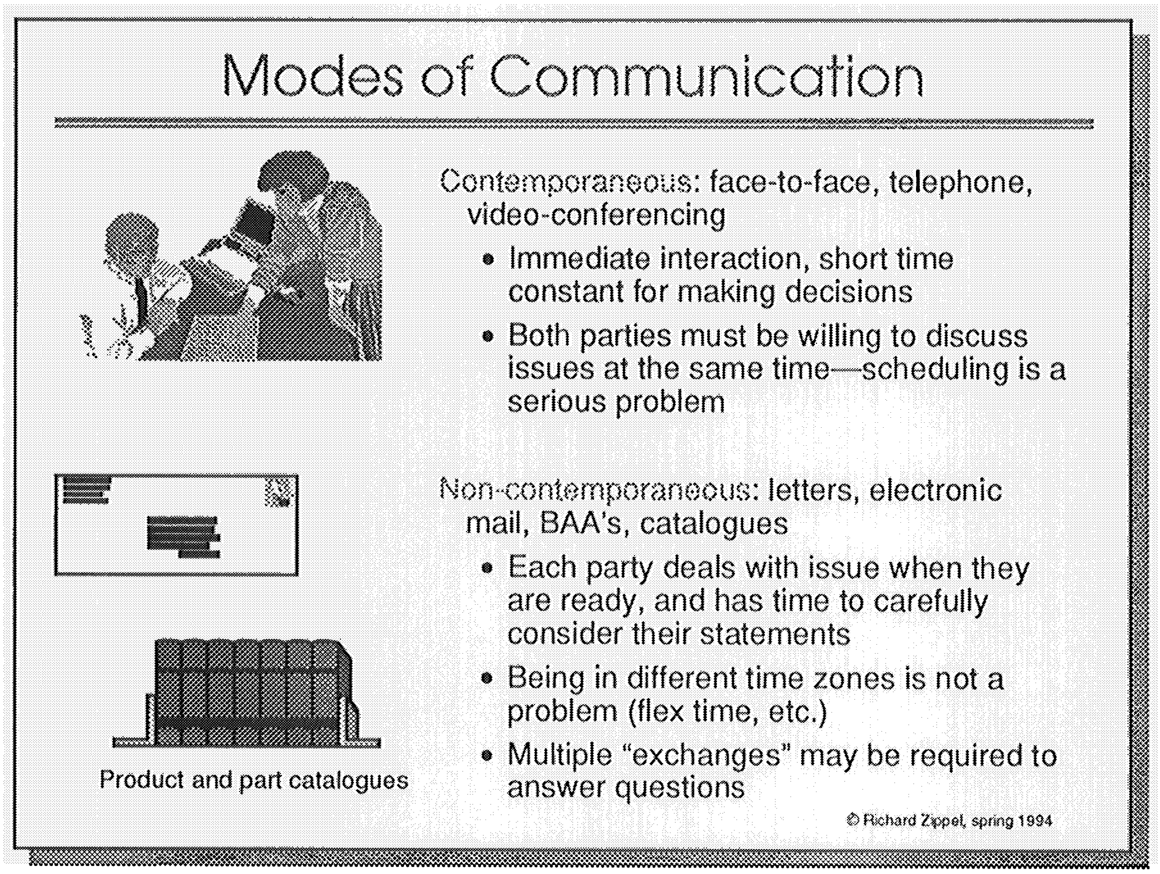
We believe that by looking at each of these mechanisms—computation, communications and storage—with an eye towards modularity and cognizant of the technological changes occurring today, we can develop the revolutionary applications of tomorrow as well as a range of visionary new systems research directions that will enable these applications.

While we have developed impressive technologies for each of these mechanisms, using each mechanism is quite involved. Thus while we have specialists n each area, it is difficult to develop systems that incorporate more than one mechanism.

This difficulty can be resolved by developing *semantically richer phrases* for describing each of these mechanisms. Just as introducing the concept of a procedure call dramatically simplified creating software (and lead to modern programming languages), developing these larger semantic phrases will simplify the creation of the new systems that will revolutionize our world.



We begin this discussion with the mechanism that has been evolving the most rapidly recently, and that is having the most immediate impact on our ideas on how computers are used—communications.

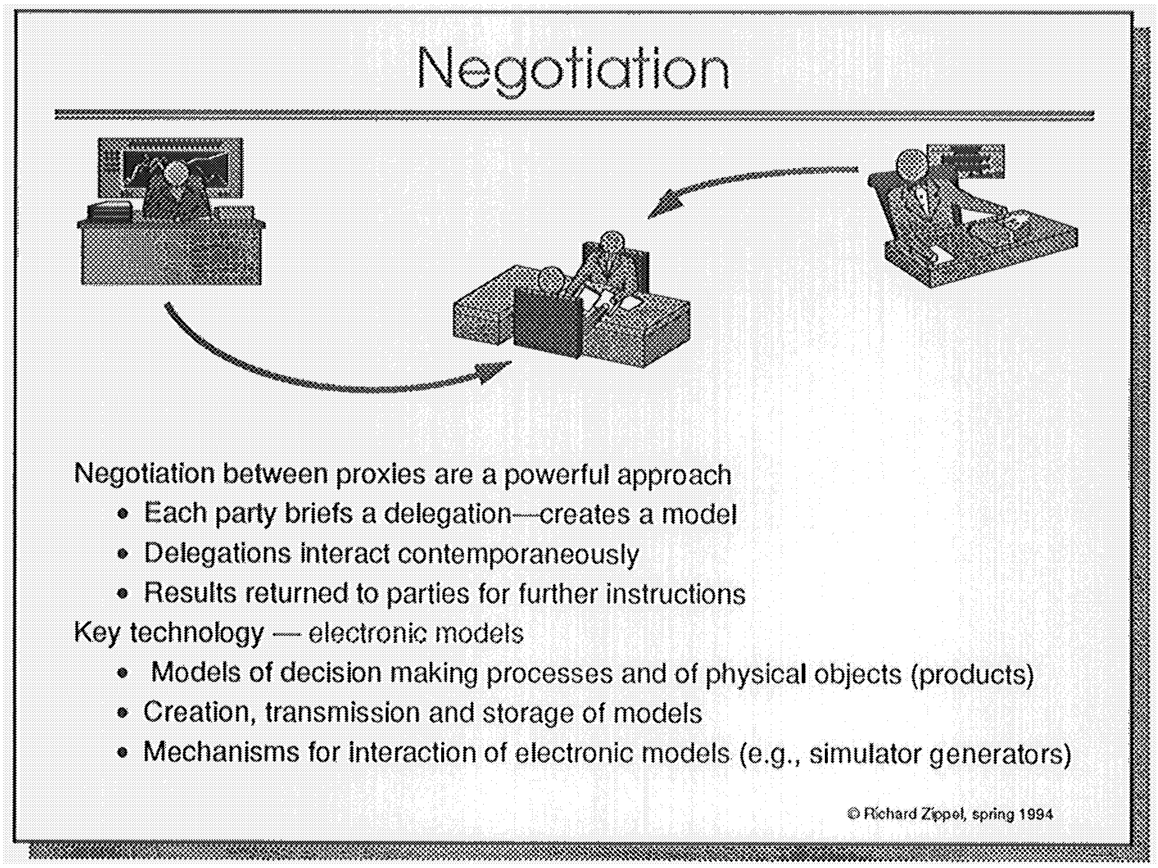


With all the different communications mechanisms developed over the years,only two essentially different paradigms have evolved—contemporaneous and non-contemporaneous communication. Typical *contemporaneous* communications include face to face discussions in the market, telephone conversations and teleconferencing. This mode of communication allows information to be conveyed quickly and is ideal for decision making situations. However, it forces all parties to the discussion to participate at the same time, which can be difficult when the parties have busy schedules and/or are separated geographically. Even with teleconferencing can be serious problem when people are separated by time-zones.

Until this century *non-contemporaneous* communications (letters) was the main mode of communications between physically separated people. This mode of communications suffices for many forms of collaboration between people in different countries or continents. It encourages people to avoid getting caught up in the heat of the moment, and instead exchange well thought out statements. It is the preferred mode of communication between nations.

In addition, non-contemporaneous communication mechanisms are used when the other parties to the discussion are unknown. The Sears catalogue was a great example of this. The Texas Instruments TTL databook is, perhaps, an even better example since speculative parts were once also described. If enough people actually ordered the part, TI would begin manufacturing it.

Because electronic communications enables people throughout the world to exchange ideas, collaborate and conduct commerce, there will be an increasing need for non-contemporaneous communications mechanisms.

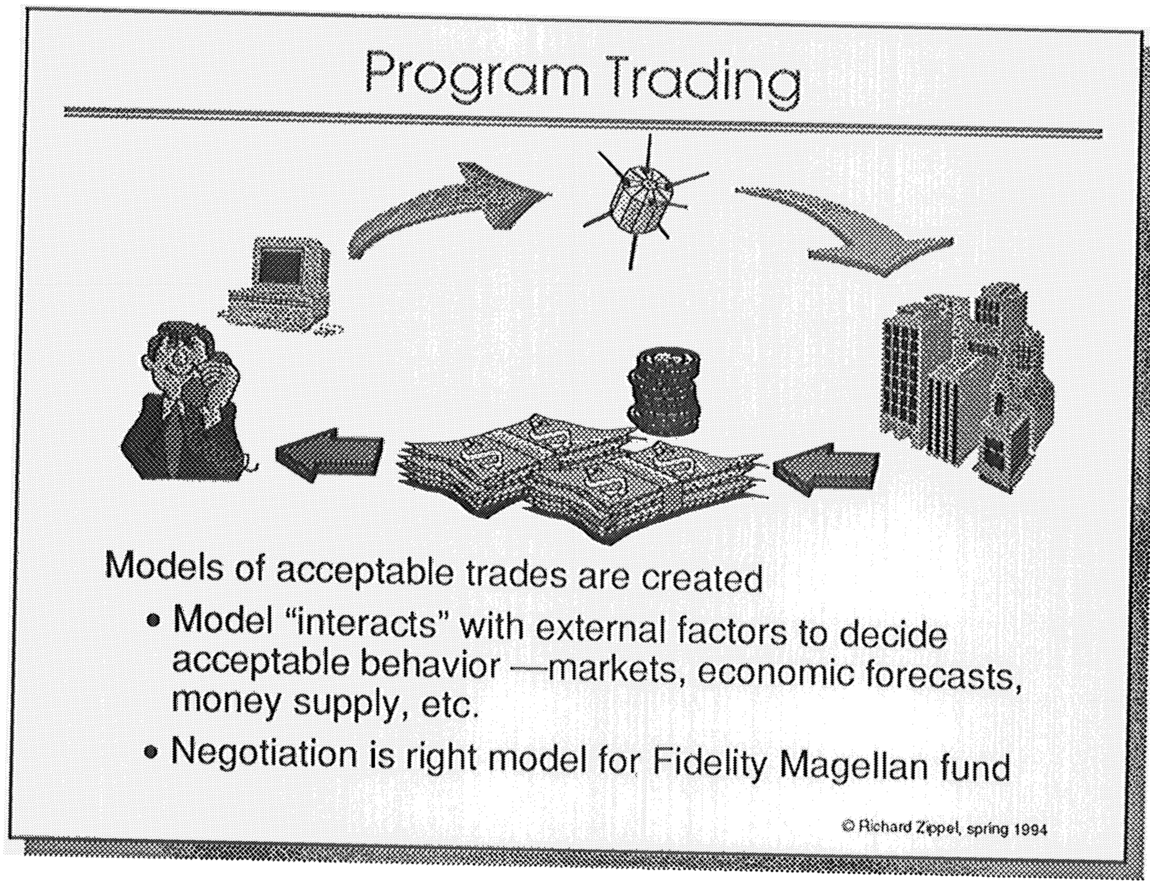


Years ago, when communications were limited, complex business and diplomatic agreements could be reached by laboriously exchanging letters. However, the alternative approach of sending emissaries proved more useful. Although, technology has provided us with telexes, faxes and electronic mail which dramatically improves the time required to transmit information over letters, we have not returned to direct negotiations. Instead the original non-contemporaneous solution is still used—*negotiations between proxy models.*

The presidents of the different organizations do not get together and negotiate. This would not be practical or an efficient use of their *time*. Instead, each party selects a delegation that is briefed on the organization's positions, resources and needs—the delegations are (inexact) models of their respective organizations. The delegations meet and exchange positions and ideas based on these models and try to come to an agreement that meets the goals of their organizations without exceeding the acceptable costs. The actual negotiating positions are, of course, kept secret.

At the end of each negotiating round, the delegations return to their organizations with the proposed (partial) agreement and seek further guidance for refining the agreement. The final decision on the acceptability of the agreement rests in the hands of the organizations' leaders.

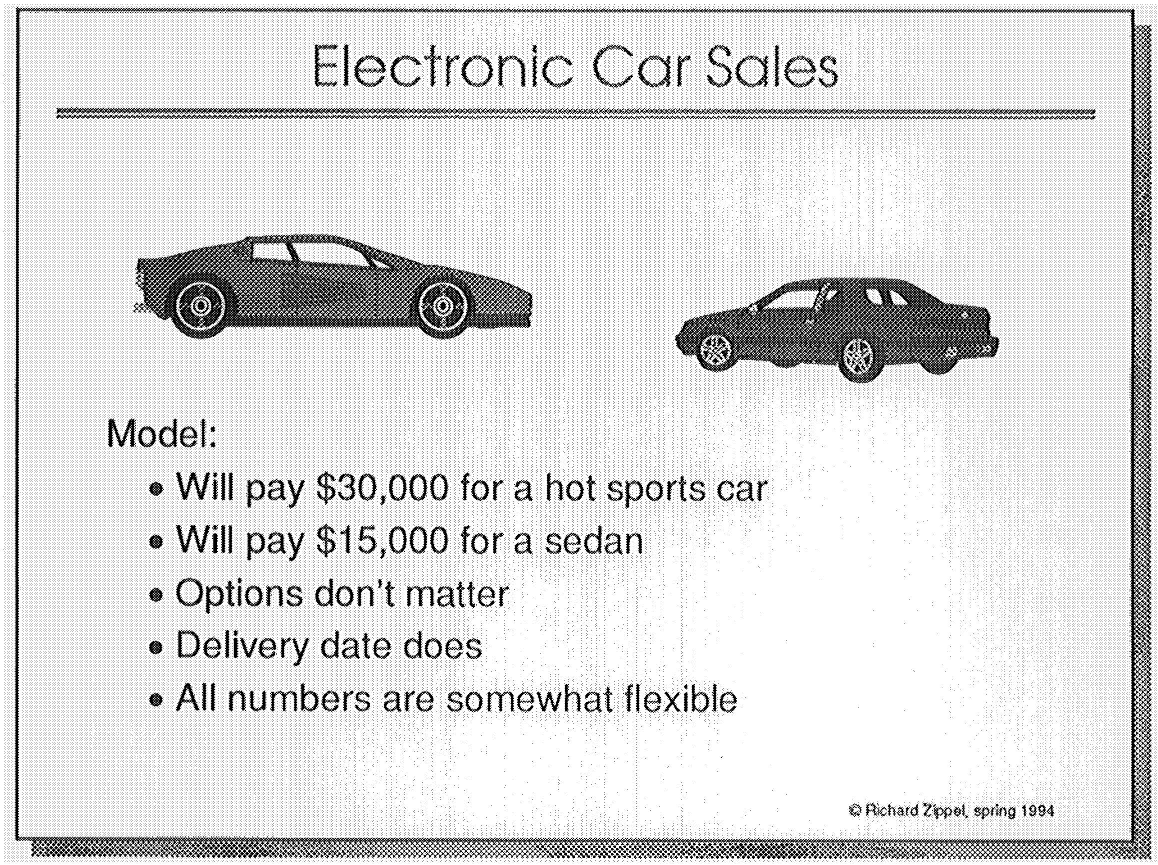
This approach of *negotiating models* continues to be of use, and can be applied to a wide variety of different examples. The key technology required to make this approach work in an electronically connected world is the creation of sealed behavioral models of negotiating positions and physical objects (the products that are being offered for sale) that can be transmitted over the electronic networks. In addition, mechanisms need to be created to allow these models to interact. That is, create simulations, at various levels of detail, from the electronic models.



A prime example of where electronic models have already been used to change the our lives is program trading on the stock markets and commodity markets of the United States. In this case, the fund manager has developed a model of the goals of the fund, the types of trades that s/he is willing to undertake and the conditions under which these trades are acceptable. This model is then continuously played against the market, and other financial data, looking for attractive trading situations.

These electronic models (of brokers) have made markets more efficient and thus the prices of the stocks more accurately assess the value of the underlying company. Ultimately, this means that money will be saved in the economy since overpricing and underpricing of financial instruments will be minimized.

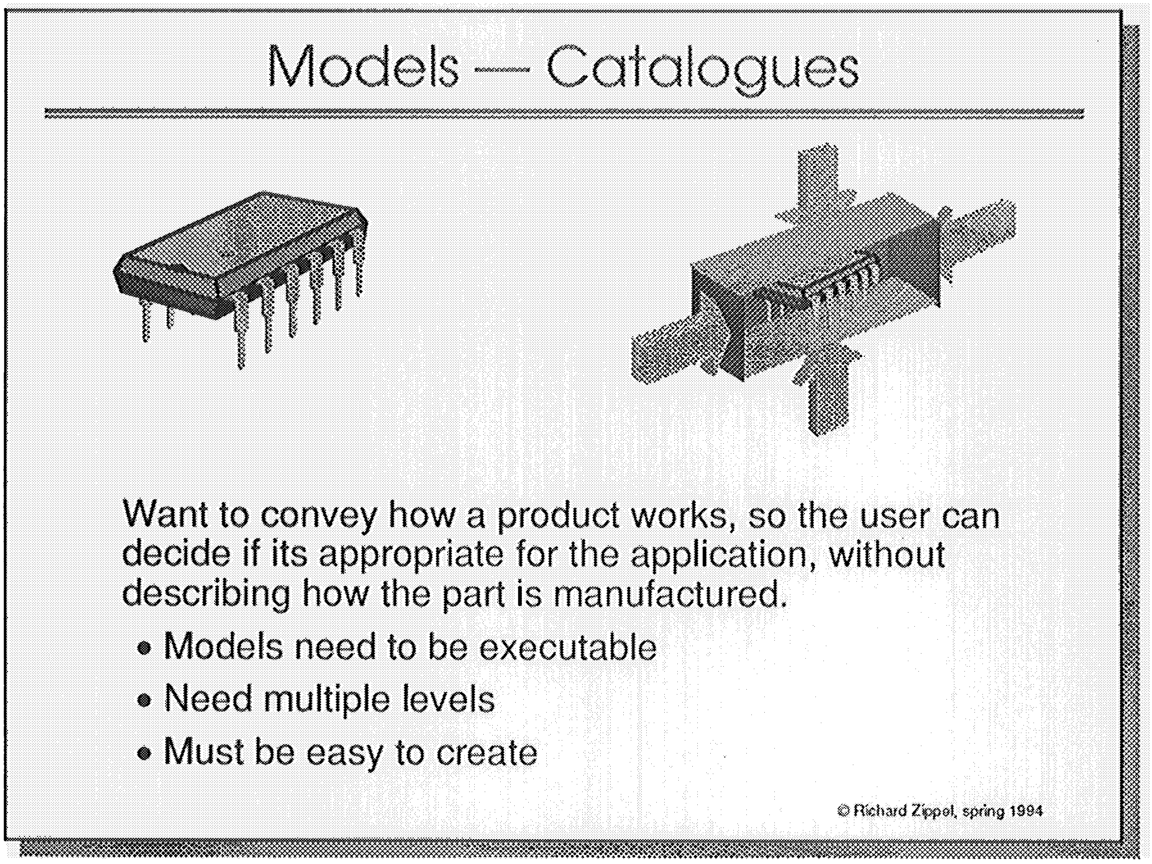
Similar brokering models could be used optimize the prices of all financial transactions performed, ranging from large purchases of energy by hospitals, to food purchases by restaurants and transactions as small as finding a baby-sitter. As commerce is increasingly transacted on the information highway, it will be increasingly possible to make these transactions electronically. And thus similar brokering models will allow business and consumers to optimize their financial transactions.



Negotiating programs behave rather differently from conventional programs. To be precise, consider the situation of needing to purchase a car. In the world of electronic commerce we expect the buyer to develop a neogiating agent using special tools. This neogtiating agent would be made available on the network and anyone with a car to sell would interact with the negotiating agent to see how close of a match there is. After a period of time the buyer would look at the best offers his or her agent has found and decide whether to pursue them in person, or to refine the negotiating agent and continue.

The negotiating position modeled by the negotiating agent involves a number of different aspects. For instance, the buyer might be willing to pay a moderate amount of money for a hot sports car, but is expecting to purchase a sedan at a cheaper level. Options, delivery dates and other aspects of the car will influence how interested the buyer is in the vehicle and how much he or she is willing to pay.

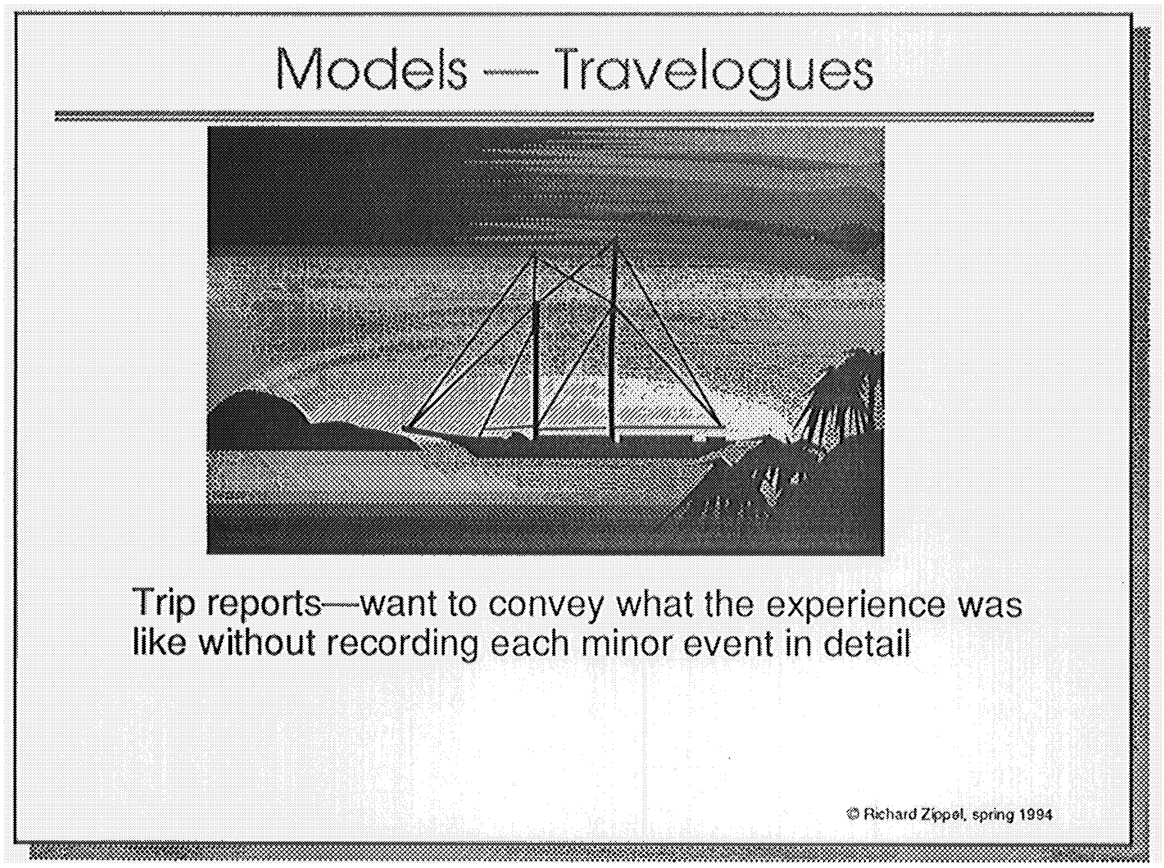
A program that captures all these different trade-offs is difficult to express in a procedural language. What is the proper language for expressing negotiating agents? Also, the negotiating agent needs to protect the details of the negotiating position it represents from the sellers. How is this best done? Can cryptographic protocols or randomization be used?



The metaphor of electronic models also works well in other domains. In electronic commerce, vendors describe and publicize their products so that potential customers can choose to purchase them. The customers will only purchase products when they are convinced the products will satisfy their needs, so the vendors are encouraged to reveal as much information abou the products as possible. On the other hand, the vendors want to reveal as little about the product as possible, since these commitments preclude future modifications and cost savings. Furthermore, the vendors want to avoid reveal any data that would allow a competitor to duplicate the product. Thus both the vendors and customers want to deal with electronic models of products of products.

Today, most of these models are provided by catalogues. The catalogue provides the customer with some information about the product, but not that much. The customer would prefer a model that could be carefully examined in the context where it will be used. Today this is why mail order firms are tend to be rather flexible with returned merchandise—it allows the customer to examine the product at home, where it will be used. It would be better if a model of set of towels could be examined—their color compared with that of the bathroom where they will be used, their feel and absorbency examined. This approach was tried in the late 1980's by Buick, who made available electronic models of a new car which would allow you to “experience” how it handled.

In order for these models to interact with the customer's environment it must be possible to create executable versions of these models. And it must be easy to create these models. One area where we believe we have a handle on this problem is models of physical systems.



Models are a generic description mechanism. In effect, they are a limited form of telepresence. In the past, this was accomplished by literature, paintings, photographs and movies. Perhaps in the future we will be able to use models.

In giving a trip report, there are two things one wants to do. First, on needs to pass on the detailed information that has been gathered. This is probably the most important aspect of a trip report is usually dealt with well via a memo. By archiving this memo in some repository where others can access it, the benefits of the trip can be more widely distributed in the organization

And second, it is often important to convey the vague impressions that arose throughout the trip. How did the sunset feel when you arrived in Hawaii? What sort of person was the host to negotiate with? What sort of humor works best? What was the morale like? These sorts of issues are more difficult to capture but they characterize the role of a modelled trip report. They aremore like an impressionist painting than a photograph.

If this type of information can be captured well, then trip reports of vacations, or travelogues of good writers might be just as good as being there.

Research Questions

Model creation

- Language for describing decision making processes
- Modularity in model creation

Model interaction/negotiation

- Protocols for interaction
- Protocols that ensure negotiating positions are not compromised
- Mechanisms for causing models to interact efficiently
    - Model interaction may involve more than two agents

Model storage

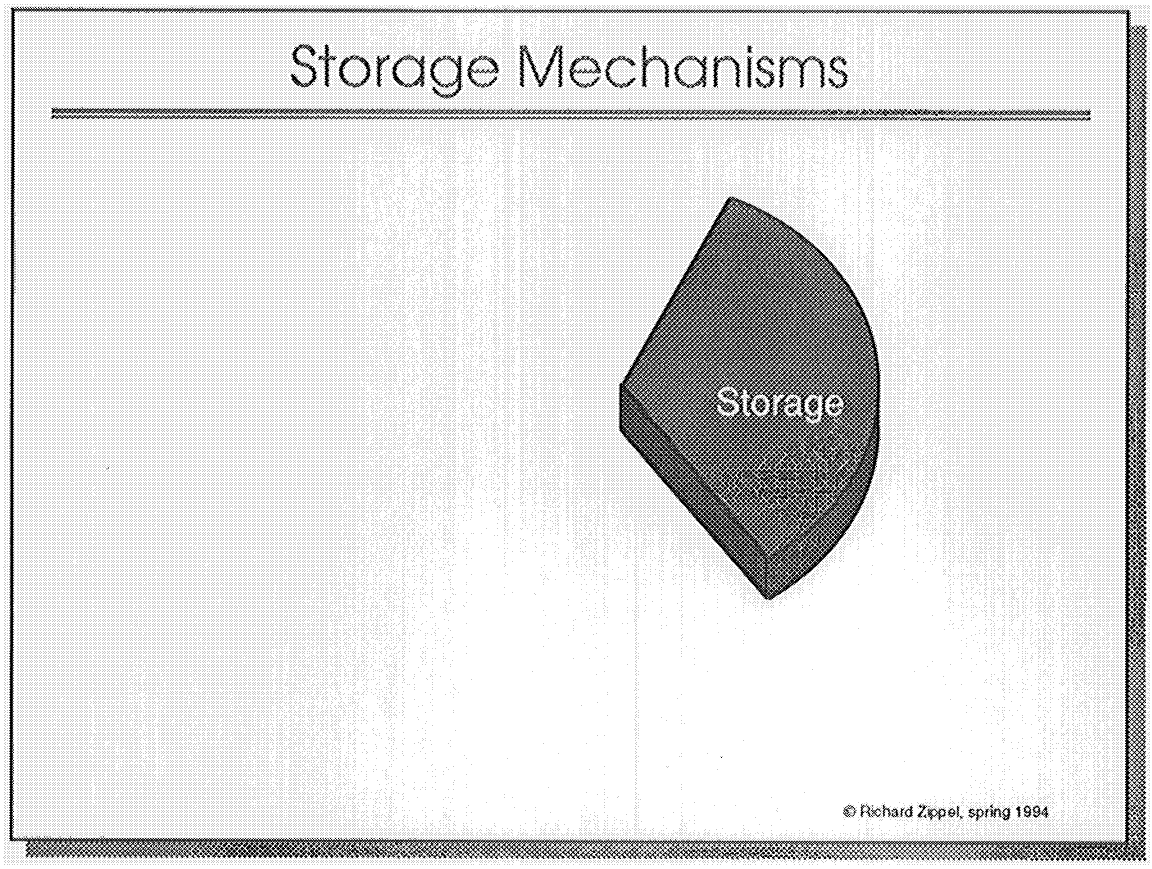
- Does model interaction change the state of a model?

The previous discussion has pointed out how the communications based on models can be used to facilitate future applications. The ability to create and use these models will dramatically increase the flexibility of the applications we will create in the future.

A number of research questions need to be addressed though. How can we effectively create electronic models? What sort of language will be used for describing physical objects and decision making procedures? Can systems be created to simplify the creation of models sufficiently that their creation can be a part of everyday life?

Once models are created how do they intereact? What are the set of operations that negotiating agents use? How does one create an efficient, large scale simulator that combines aspects from several different models?

And finally, how does one store models? This must be done so that the models are accessible by machines of widely varying capabilities and in ways that can protect the private data inside the models.



We think that an important, but often overlooked problem in systems is flexible storage management. Systems that deal with decision making models, and especially physical models, which may contain sound and video, will benefit from flexible storage systems.

Storage Demands

Variety of different data types

- text files
- program images
- pictures, video, sound

Variety of different interface models

- Unix style text files
- Object oriented data bases
- Single level stores

Problems for a single system:

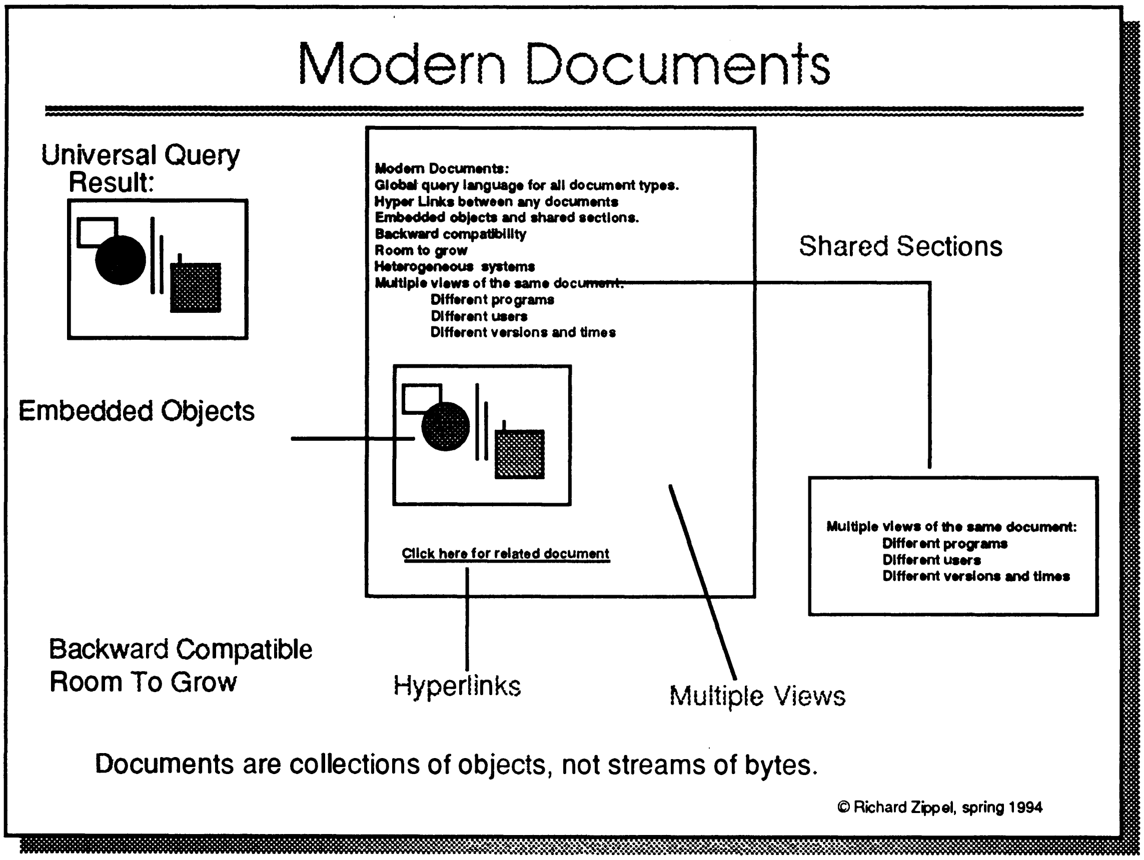
- Difficult to support different data rates
- Difficult for different interface models to coexist
- Need framework for distributed architecture

Modern applications place increasingly stringent demands on storage systems. First, storage systems are expected to deal with a wide variety of different data types. While in the early days of computers text files or linear arrays of numbers sufficed, now program images (which have a richer structure than text files), large collections of database records and continuous media need to be stored.

Second, we have a variety of different interface models for this storage, ranging from the simple file model of Unix, through relational and object oriented database structures. We are just starting to learn about the application program interfaces (API's) that are appropriate for continuous media.

As we build more complex systems in the future, addition storage models will arise that we cannot now foresee. It is incumbent on us to develop systems that can deal with all of these different structures as efficiently and easily.

Thus far this has not been the case. Rarely do object oriented systems coexist with conventional file systems.

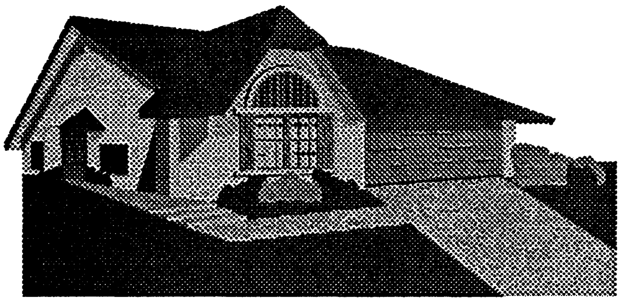


The documents manipulated by modern word processors exhibit this complexity. They are much more than a sequences of characters. Now they can contain embedded objects of varying types including tables and graphics. Some of these shared objects may be shared with other documents, or may be accessible to other applications, e.g. tables may actually be spreadsheets. Finally, the document may not even have a linear structure, but may contain *hyperlinks* to other points in other documents.

The desirability of this compound document structure can be seen from the rapid acceptance of standards like Microsoft's OLE 2, Apple's AOCE and the World Wide Web on the Internet.

But even these documents do not exhaust the possible components.

Documents with Models



Design of building
- drawings, textual descriptions, simulation results
- schedules
- negotiating positions on marble façades, appliances

Knowledge is a web of connected models

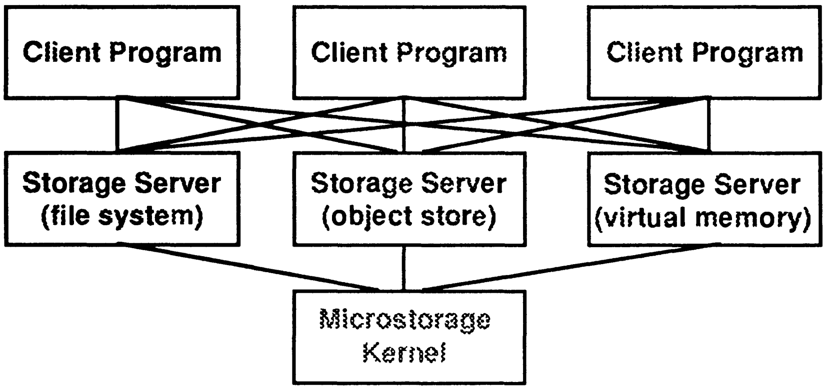
Knowledge is what needs to be stored

Consider a "document" that captures the design of a building. Such a document contains construction plans of the building, floor plans, wiring and plumbing schematics, textual specifications of the image the building should project and three dimensional renderings of the building. In addition, for certain types of buildings, simulation and analysis results (e.g., to make sure that windows will not fall out) are included.

Added to this are schedules for delivery of building materials, availability of different craftsmen and financing, and negotiating positions for the purchase of all the exotic materials used. For instance, in a residential house the architect might specify a certain type of marble to be used in the entrance way if it is available at a "good" price, otherwise a cheaper granite might be a better choice.

Documents are in the business of representing knowledge. In the past books were the approach we most often used for representing knowledge. But today we have found that more complex structures can be useful. In principal, we feel that knowledge should be viewed as a web of connected models. The models can be as simple as a text file, or as complex as the three dimensional description of the shape of a building. In the end this web of models is what storage systems must be able to deal with. But storage systems must also deal with existing forms of data and allow for a smooth evolution from our current storage models to new models.

Microstorage Architecture



- Each storage server implements a different storage model
- Different storage models are different interfaces to the same data
- Microstorage kernel deals with disk, caching, faulting, etc

Supporting these varied documents and the applications that will use them would be very difficult for a single storage system, especially given the changing requirements of new applications. To address this problem we propose a *microstorage architecture* that separates kernel aspects of the storage system from the storage server models that provide the different application program interfaces. This approach of providing multiple "personalities" for the storage system that are implemented on top of the single, thin kernel is similar to the microkernel approaches used in current operating systems.

The microkernel is responsible for moving data between memory and the disk, caching data operations, organizing the data on the disk, and a number of other issues that will be discussed later. It presents to the different *storage servers* a single, relatively simple model of storage. The storage servers are responsible for implementing different interfaces to storage. For instance, file systems and object oriented stores can be easily implemented as storage servers. Somewhat surprisingly perhaps, one could also implement the virtual memory system of operating system as separate storage server.

The client programs are then free to access as many or as few of the storage servers as desired. As new storage needs arise new storage servers can be implemented, experimented with and deployed without the need to remove or replace the existing storage servers. The new servers can co-exist with the old servers without compromising performance. In addition, using some of the techniques to be discussed, data that is managed by one storage server can also be made available to other servers. This allows continuous data, like sound and video, to be incorporated into "textual" files in a properly designed file system.

Microstorage Kernel

Role of Microstorage Kernel:

- Provide building blocks for files, objects, models, etc.
- Independent of any particular storage policy
- Language independent—OS extension, not language extension.

Segments—basic building block

- Variable sized block of data managed by kernel
- Properties—invisible to user, but provide storage for kernel and server usage.
- Links—persistent pointers between two segments
- Reference counts and garbage collection (for cycles)

The microstorage kernel, or *microstore* for short, provides the basic building blocks for constructing higher level storage systems. The microstore does not specify any particular storage policy. Rather, it is provides the mechanisms for implementing the policy specified by a storage server. The facilities provided by the microstore are intended to be langauge independent. They should be viewed as an operating system extension that is available to all programming languages. This precludes using linguistic mechanisms that are not universally available, unfortunately.

The basic building block provided to the storage servers by the microstore are *segments*, which should be viewed as variable sized blocks of data that are maintained contiguously on the disk. Although there are certain differences that we will not go into here, the storage server developer can view these segments as the disk blocks of the storage system. Thus a Unix style file system might choose to only use 4 kilobyte segments, while a multimedia storage system would use larger segments and an object oriented server might use segments of varying sizes.

Attached to each segment, but invisible to the client programs, is a set of properties that are used by the microstore for bookkeeping and are available to the storage servers for their own purposes.

The microstore provides a globally unique identifier for each segment. Combined with a segment offset, this permits the creation of *links*, or pointers, between segments. The global nature of the segment identifier means that links can point across machine boundaries. The microstore provides a reference count mechanism for dealing with most of the reclamation issues and general garbage collection for dealing with cycles that cannot be dealt with using reference counts.

Links

When segment is in memory, links become pointers

Links to non-memory resident segments are trapped

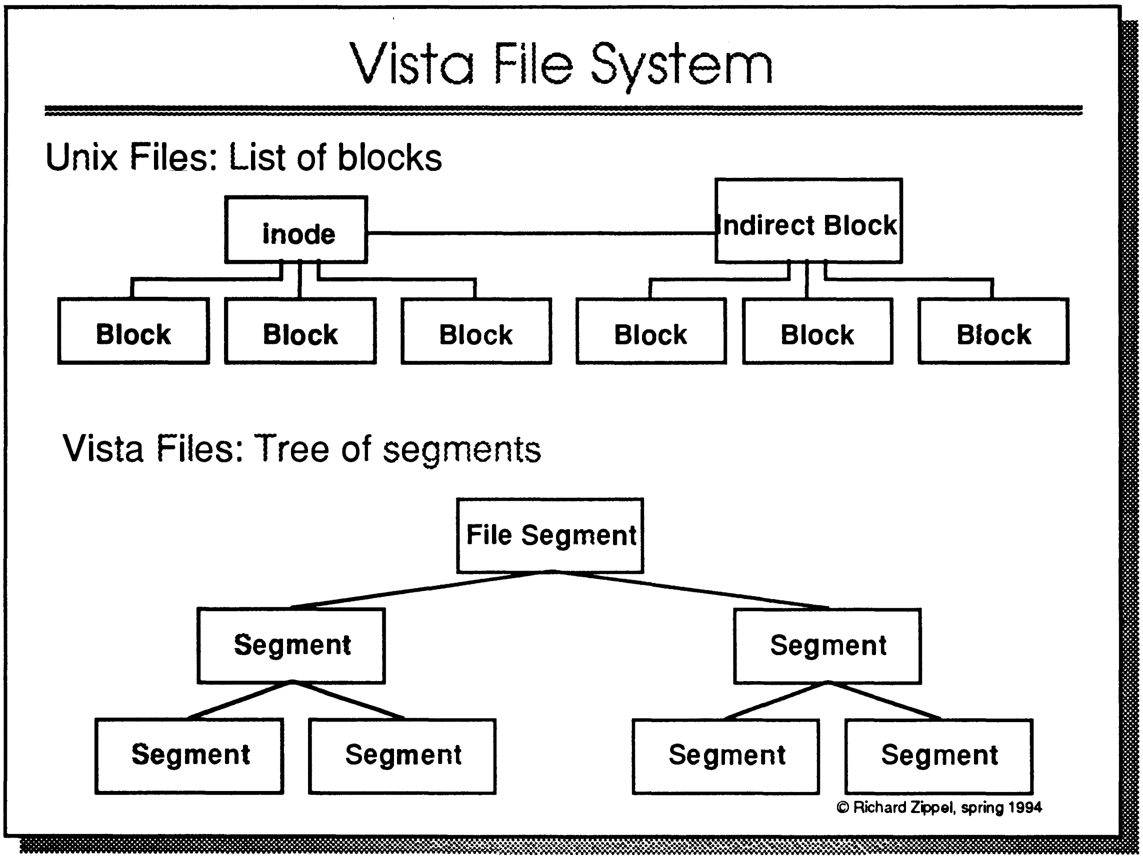
Trapping mechanisms used for a variety of applications

- Object oriented storage systems
- Split transaction remote procedure call
- Multiprocessing "futures"
- Virtual memory

Links are a mechanism that is usually not provided in a storage system. At a some performance penalty they could be implemented in the different storage servers. However, we feel that a pointer mechanism that was uniform across all storage servers would be much more powerful. This requires standardization, which is most easily accomplished by providing the link mechanism in the microstorage kernel. Furthermore, in order to efficiently implement garbage collection mechanisms, it was best that the microstorage kernel be aware of the structure of links. Finally, experience has shown that unless garbage collection mechanisms are provided at the initial design of storage system, they are rarely successfully implemented. All of these issues strongly supported our decision to provide the link mechanism in the microstore.

The full form a link is a rather large structure, since it must uniquely identify every machine on the network (at least the Internet) and all the segments each machine is able manage. Thus they are typically about 128 bits (2^{64} different machines, 2^{24} disks per machine and 2^{40} bytes per disk). When a segment containing a link is loaded into memory, however, the link is represented by the machine's standard pointer. If the link points to data that is not in memory then a fault will result when the link is de-referenced. This simple approach directly implements virtual memory as a special case.

This mechanism can be used for other purposes as well. In an object store, inter-object pointers are naturally implemented as links. In parallel processing environments it useful to have a split-transaction type of remote procedure call. For this type of procedure call, the caller does not wait for the callee to return the value. Instead, the caller proceeds with its computation until the callee's result is needed and then checks to see if has already been provided. Using the link mechanism, the callee will immediately return a link to a block of memory that will contain the result of the computation. The caller can then anything desired with the link. If it tries to de-reference the link before the computation is complete the microstore will fault.



To demonstrate the feasibility of the microstorage architecture, Dawson Dean has implemented a prototype microstore called *Vista* and has implemented a file system on top of Vista.

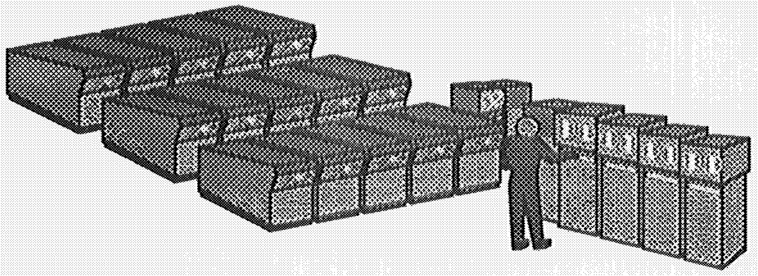
A standard Unix file system implements a file as a linear array of (typically) 4K blocks. For large files, this may involve one or more indirection blocks. Among other things, this leads to fast seeking to specific positions in the file.

Within Vista a file is organized as a tree structured set of segments rooted at a file segment. The segments in the tree can be 4K byte "text segments," as in Unix, but can also be variable sized segments as needed to represent database or multimedia objects that are embedded in the file.

A common operations used by some programs is to jump to specific byte in a file, often called an `lseek`. With a Unix file system, performing an `lseek` is quite easy because of the linear structure of files. This operation can usually preformed with a single disk seek. In Vista, this is more difficult. The size of each subtree in a Vista file can vary because the segements can cary in size. Thus, in the worst possible case the each segment must be examined to determine the size of a file. Because the segments are organized as a tree, we can usually improve this to log n seeks by caching in each segment the size of the sub-tree of segments below it. However, a problem can arise if one of the segments in the sub-tree is managed by a different storage server. In this case, the a segment can change size without the the Vista storage server being informed. Thus it may be necessary to to examine each of the segments in a sub-tree to determine the size of the sub-tree. A full traversal of the tree is sometimes necessary to determine the size of the file.

However, we feel that most applications that make heavy use of `lseek` typically are representing more complex structures using text files and are better implemented using a more sophisticated storage format.

Consistent Backup



Problem: How to perform a consistent backup of a 100Gbyte, continuously changing storage system?

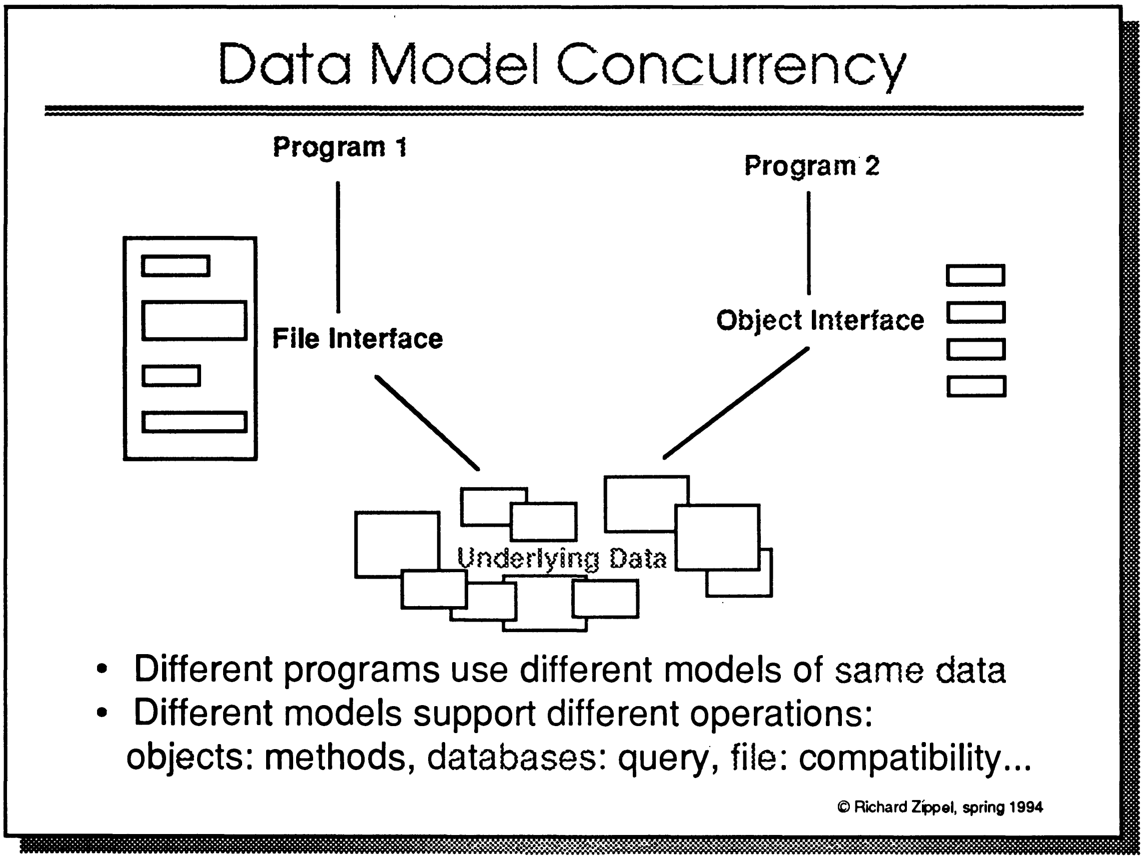
- Distinguish backup process from user process and provide each different versions of segments

© Richard Zippel, spring 1994

One example of where the microstorage architecture does provide added performance performing a consistent backup of a large disk farm. On many systems this is quite difficult, because the data on these disks is always changing. The only way to provide a consistent backup is to prohibit writes on the disks during the backup. Usually this means shutting own the disks for the several hours require to perform the backup. This is not feasible for most large storage systems.

Using the microstorage architecture, one can easily implement a scheme that was originally patented by IBM for use in the I/O channel based disk systems. At the microstore level we distinguish two different types of processes—backup processes and all others. At the beginning of a backup each machine is asked to mark all segments in memory as read-only. From then on, if a user process modifies the segment, the original value of the segment is preserved and a new read/write copy is provided for the user process. Similar actions are provided for all segments that are brought into memory. However, when backup processes request a pointer to a segment they will be given the original segments. This process is requires only a minimal pause in the operation of the system when the backup is initiated. Note that this technique permits consistent backup of all storage servers concurrently. This is typically difficult in other implementations.

Caching can also be improved by having the microstore optimize the use of cache memory across all the storage servers, rather than optimize each independently. This will give better overall system performance for a given amount of cache memory than independently optimized systems.



In the microstorage architecture segments can be used by more than one storage server. This is an important ability that allows legacy software based on text files and programs that use more sophisticated data structures to coexist. A simple example of this is provided by a bibliography database where each bibliographic reference is represented by an object in an object oriented database. At the same time, this collection of objects must also be viewed as a text file for older software like Bibtex or Ref. In this second case, the file system storage server will synthesize the textual format required by the legacy programs from the objects found in the database.

Since the segments are being concurrently used by more than on storage server, we call this the *data concurrency* problem.

Data concurrency introduces a new set of problems not normally found in standard storage systems. It is possible that operations by one storage server will confuse other storage servers. An example of this is was mentioned previously when discussing the length of Vista files. In this case the problem is easily dealt with by paying a performance penalty in the Vista file system version of `lseek`. For more complicated operations no such simple solution is possible.

Mechanisms to indicate ownership of segments and inheritance of properties are provided in Vista to solve the data model concurrency problem.

Virtual Memory Storage Server

Virtual memory can be implemented as another storage server

- Paging faults are handled by link trapping mechanism
- Page faults may be resolved across network

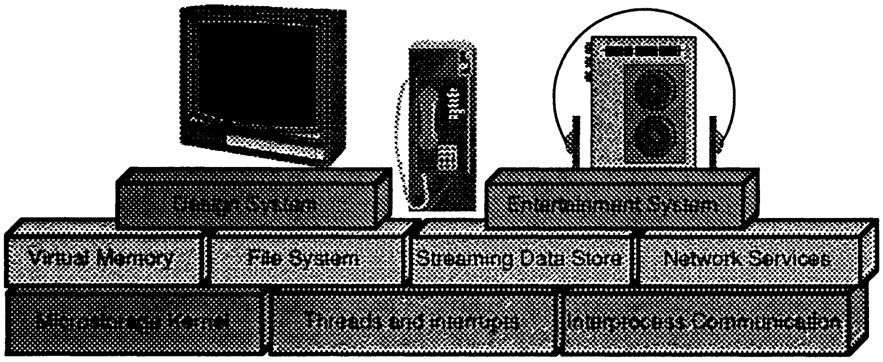
New model of operating system

- Microkernel — thread handling and interrupts
- Microstorage — foundation for all storage

As pointed out earlier, virtual memory is easy to implement using the microstorage kernel. Page faults are handled by the same mechanism used to deal with link trapping. Coupled with the globally unique segment ID's this approach allows page faults to be resolved across the network, if one should so want.

Perhaps the most interesting aspect of implementing virtual memory over a microstorage architecture is that it teases apart a conventional microkernel operating system into two pieces—a microstore for dealing with all of the storage issues, and what's left. What's left of the microkernel which is thread handling, interrupts and device drivers. Virtual memory is implemented on top of the microkernel/microstore.

System Organization



Build address spaces out of segments

- Map files into virtual memory
- Share segments between address spaces
- Access controls for shared memory

Segments provide access to all data in a system: memory and disk

We believe that the microstorage architecture just described lays the foundation to a new type of operating system that is more modular and flexible than existing operating systems. The storage model with which applications are built will not be the segment model provided by the microstore, but rather the model provided by the different storage servers. If additional performance is required, different storage servers can be constructed, but the semantic level at which the application builder will work will be higher. For instance, it is important that the application developer not specify that a large block of data be stored on disk in sequential blocks, but rather indicate the throughput and latency requirements the application places on the data. The responsibility of the microstore is to provide the data within those parameters.

While microkernelized operating systems were great conceptual simplifications of the previous organizations, the microstorage architecture provides additional simplifications. As shown in the figure, the lowest layer of the operating system is divided into three simple components: the microstore, the interrupt and thread manager and the interprocess communications manager. Notice the parallel in this division and the general partitioning of systems' mechanisms discussed earlier:

microstore—storage

threads—computation

communications—interprocess mechanism

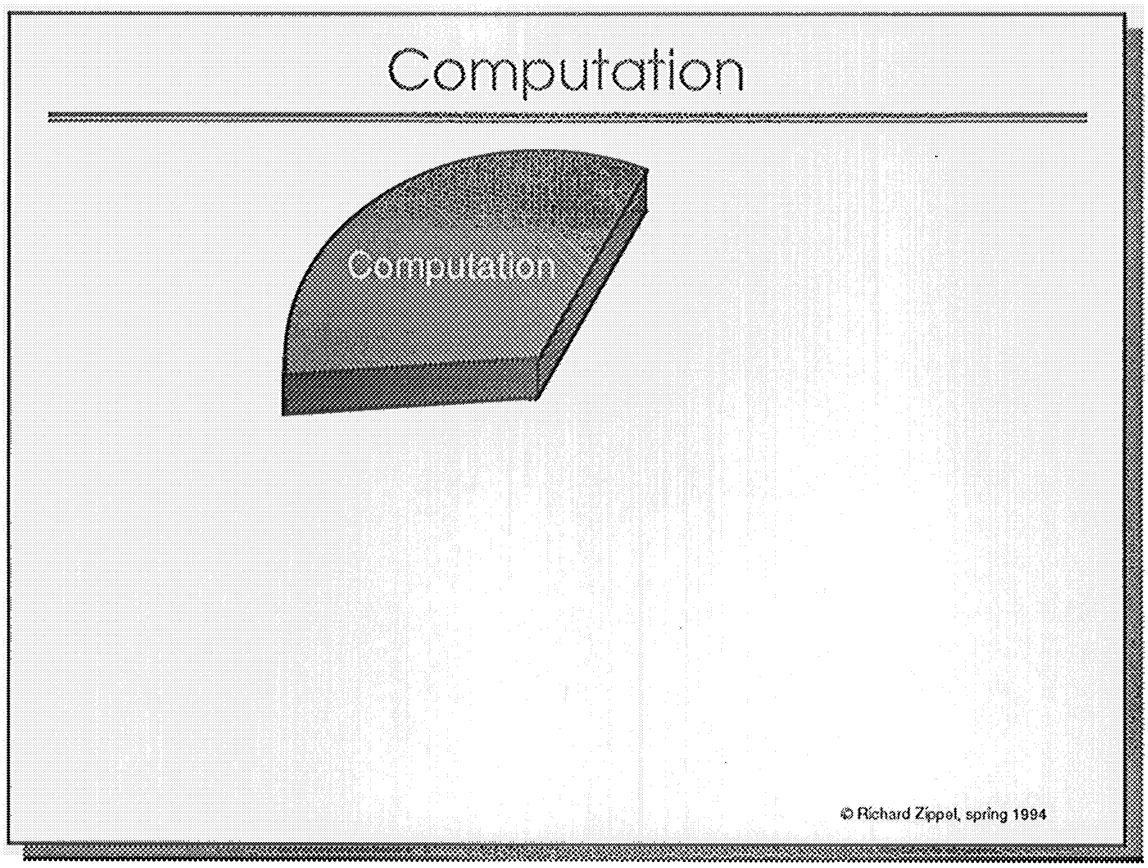
Research Questions

- Software for easy creation of storage servers
- How will multiple storage servers change our approach to systems and applications building?
  - What is a file system when we have a microstore?
  - What role should each data model have in a microstore?
- Negotiation between storage servers and microstore
- How to tune storage performance to applications?
  - No modification of kernel
- How to provide consistent concurrent *multiple* data models?

A variety of research questions arise when developing microstorage architectures. First, while many applications will benefit from specialized storage servers, and the microstorage architecture simplifies the creation of the multiple storage servers, it is not easy to create a storage server. Better tools and abstractions need to be developed to alleviate this problem.

How will multiple storage servers change the organization of systems? We have already seen changes arising from component technologies like OLE and OpenDoc. How will a microstorage architecture play with the object models they provide, and will the ability to create new storage servers introduce new application modularity structures?

A crucial aspect of the performance of a microstorage architecture is ensuring that microstore performs precisely the operations required by the storage server with a minimal amount of overhead. How much influence can a storage server have on the caching policies of the microstore? More generally, what other aspects of the storage servers' policies must be conveyed to the microstore and how do the storage servers negotiate with the microstore?



We now move on to the computational aspects. Here we discuss some ideas that lead to higher level descriptions of computational processes and tools that can convert these descriptions to executable code.

SPL/Weyl Philosophy

Weyl
Extensible
Symbolic Computing
Substrate

SPL
Very High Level
Mathematical
Programming Language

- Raise the semantic level at which mathematical computations are described
  - All mathematical concepts should be expressible in language, even if not effective
- Language is a specification of what to compute
  - Transformations indicate how to compute
- Correctness of program includes many mathematical issues

These tools include a new programming/specification language for mathematical computation called SPL and a library of program transformation that convert these specifications into executable code. SPL is a more or less conventional language that has been extended to include constraints on variables. Uniquely to SPL, these constraints can be coupled algebraic equations, differential equations and even minimization principles.

The transformations that convert SPL programs into directly executable code are implemented using a symbolic computing substrate called Weyl, which extends a conventional programming language (Lisp) to manipulate algebraic expressions. Within this extended Lisp algebraic numbers, polynomials, rational functions and matrices of these objects can be manipulated with the same operators as integers and floating point numbers. In addition, the algebraic domains of which these objects are elements are also first class citizens and can be used.

Both SPL and Weyl illustrate our general approach of raising the semantic level at which systems are described—in this case for mathematical computations.

Weyl's Approach

Extends an existing programming language (Lisp) to include symbolic mathematical capabilities

- Permits use of existing software: window systems, networking tool-kits, debuggers, etc.
- Obviates the need to learn a new programming paradigm

Endeavors to capture mathematical semantics and idiom

Domains as well as their elements are first class objects

Functorial approach simplifies extensibility

Where appropriate new control structures are introduced

Based on a simplification and reorganization of ideas in IBM's Axiom/Scratchpad

One of the features of Weyl that distinguishes it from all other symbolic computing systems, is that it is not a self contained programming environment. It does not have its own programming language or unique user interface. Instead it extends a conventional language with symbolic computing facilities. To program in Weyl you must know Lisp, but you need not learn a new Weyl language. In addition all the debugging and program development facilities of Lisp are available to the Weyl programmer. While Lisp was chosen for the initial implementation of Weyl, the basic ideas could be ported to other object oriented programming languages. In addition, we have been actively exploring the issues involved in hosting Weyl in C++ for portability.

Symbolic algorithms can be coded in Weyl in a functorial fashion. For instance, the basic matrix package is defined over an arbitrary field. By combining the matrix domain creation routine with different domains, one can create matrices whose elements are integers mod p, rational numbers, rational functions, etc. and all matrix operations will continue work. This functorial approach again exemplifies the philosophy of raising the semantic level at which programs are developed.

In many ways, Weyl is a reorganization and simplification of the ideas developed by the Axiom/Scratchpad project at IBM Research at Yorktown Heights.

Domains in Weyl

Domain Elements	Domains
1.23	$\mathbb{R}$
$x^2 + 1.23$	$\mathbb{R}[x, y, z]$
$\langle x^2 + 1.23, xy, yz \rangle$	$\mathbb{R}[x, y, z]^3$

Domains are first class objects

- Need to compute with domains: algebraic extensions, tangent spaces, homology groups
- Information can be attached to them (e.g., dimension, Euclidean, Hausdorff)

Weyl provides the usual types of elements for arithmetic operation, real numbers (of arbitrary precision), polynomials, vectors, matrices etc. In addition, the domains of which these objects are elements are also first class objects in Weyl. In fact one creates a ring of polynomials by specifying a list of variables and the domain from which the coefficients will come.

These domains are very useful attachment points for information. For instance, the characteristic of of ring, the dimension of a vector space, etc.

Algebraic Example

$$f_n = -\mu f_{n-1} - \sigma(\mu + 2\varepsilon)\frac{\partial f_{n-1}}{\partial \varepsilon} + (\varepsilon - 2\sigma^2)\frac{\partial f_{n-1}}{\partial \sigma} - 3\mu\sigma\frac{\partial f_{n-1}}{\partial \mu}$$

```
(setq R (get-polynomial-ring (get-rational-integers) '(m e s)))
 → Z[m, e, s]

(setq mu (coerce 'm R))
(setq eps (coerce 'e R))
(setq sigma (coerce 's R))

(defun f (n)
 (if (= n 0) (coerce 1 R)
     (+ (* (- mu) (partial-derivative (f (- n 1)) eps))
        (* (- sigma) (+ mu (* 2 eps)) (partial-deriv (f (- n 1)) eps))
        (* (- eps (* 2 sigma sigma)) (partial-deriv (f (- n 1)) sigma))
        (* -3 mu sigma (partial-deriv (f (- n 1)) mu)))))
```

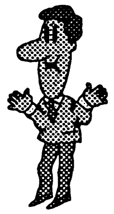

This example uses Weyl to perform a simple computation. The functions f_n satisfy the recurrence relation shown at the top. To begin the computation, we must first create the algebraic domain in which the f_n lie. This is done by the assignment statement.

Next the symbolic variables are mu, eps and sigma are created as polynomials in the ring to represent the expressions μ, ε and σ respectively. Finally the program is written as a recurrence in the standard fashion. Notice that standard Lisp programming is used, with the usual control structures and operators.

Program Transformations

Compilers

- Strength reduction
- Loop unrolling



Mathematical Analysis

- Horner's rule

$$a x^3 + a_1x^2 + a_2x + a_3 \Longrightarrow ((a_0x + a_1)x + a_2)x + a_3$$

- Discretization techniques

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} \Longrightarrow \begin{array}{l} u_i^{(n+1)} \leftarrow u_i^{(n)} + \frac{\Delta t}{h^2}\left(u_{i+1}^{(n)} - 2u_i^{(n)} + u_{i-1}^{(n)}\right) \\ \Delta t \leftarrow ??? \end{array}$$

The SPL programming environment is based on two innovations. First, the SPL language is used to describe mathematical computations at an exceptionally high level. Second, SPL programs are converted to executable programs using a rich library of powerful program transformations. We begin by briefly discussing transformations in scientific program.

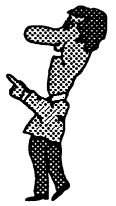
Two research communities are actively developing transformations of computational structures. The most familiar community, to computer scientists, is the programming language and compiler communities whose transformations include loop unrolling, strength reduction and other forms of program restructuring.

We claim that the computational/applied mathematicians are also developing program transformations. A simple example is Horner's rule, which converts a polynomial written as a sum of products into a recursive form that can be evaluated more efficiently. Similarly, we claim that the discretization of a differential equation is also a program transformation. At first glance these transformations may appear to be transformations of equations, but on closer examination this is not the case. For instance, the discretization of a differential equation also includes, in this example, the a time advancement loop and a strategy for choosing the next time step.

The SPL programming language is designed to support both types of transformations. Unique to our environment has been our ability to capture many of the mathematical transformations like discretization of initial value problems.

SPL Language

```
(defprogram Square ((P R))
  (bind ((w nil R) (h nil R))
    (constrain (w h) (h + w = P/ 2)
      (maximize (w h ) (hw )
        (print (list w h))))))
```



- Variables can have the types of Weyl expressions, e.g. differentiable functions
- Constrain: Variables within a given scope satisfy certain equations or inequalities
- Minimize: Variables within a given scope minimize/ maximize a given expression
- Invariant: Additional relationships that hold among variables, but not part of the state equations.

This simple program, which computes the height and width of a rectangle of a given perimeter with maximum area, contains many of the novel features of SPL. The program takes the perimeter of the rectangle as its argument. This argument will represent an element of the field of real numbers, as indicated. Next two variables are created, w and h, which also represent real numbers, but which are given no initial values. Since these variables will correspond to the height and width of a rectangle of perimeter P, their sum must be $P/2$. This is indicated by the constrain statement. The following statement indicates that the product of h and w should be maximized. Finally, we print out the value of h and w.

Notice that we have not indicated how the computation is to performed, we have merely indicated what the computation is supposed to produce. At this point SPL is more of a specification language than programming language.

Lagrange Multipliers

```
(defprogram Square ((P R))
  (bind ((w nil R) (h nil R) (λ nil R))
    (constrain (w h) (h + w = P/ 2)
      (constrain (w h λ) (h + λ= 0, w + λ = 0)
        (print (list w h))))))
```

Lagrange multiplier transform

- introduces new variables
- converts maximize/minimize form to constrain

Similar transforms can be developed for calculus of variations

By using a transformation that implements Lagrangian multipliers we can produce the program shown. Notice that a new variable has been introduced, λ, which is the multiplier. the maximize constraint has also be replaced by a pair of linear equations which arise from differentiating the Lagrangian. At this point it is only necessary to convert the linear equations into a program that solve them.

Similar transformations can be developed for minimizing functionals using the calculus of variations.

Program Transformations

```
(defprogram LinearODE (end)
 (bind ((x nil C∞(R→R)) (y nil C∞(R→R)))
  (constrain (x y) (∀ t ε [0, ∞] .
        dx/dt(t) = x(t) + y(t), dy/dt(t) = x(t) − 2y(t)
        x(0) = 1, y(0) = 1)
   (bind ((s 0))
   Loop:   (if s ≥ end (goto End))
           (print x(s)² + y(s)²)
           (s ← 0.1 + s)
           (goto Loop)
   End:    ))))
```

Forward Euler Program Transformation ⟹

```
(defprogram LinearODE (end)
 (bind ((x nil C∞(Z→R)) (y nil C∞(Z→R)))
  (x[0] ← 1) (y[0] ← 1)
  (bind ((n 0))
   Loop:   (if s ≥ end (goto End))
           x[n+1] ← (1+h)x[n] + h·y[n]
           y[n+1] ← 2h·x[n] + (1−h)·y[n]
           (print x[n]² + y[n]²)
           (n ← n + 1)
           (goto Loop)
   End:    ))))
```

SPL language

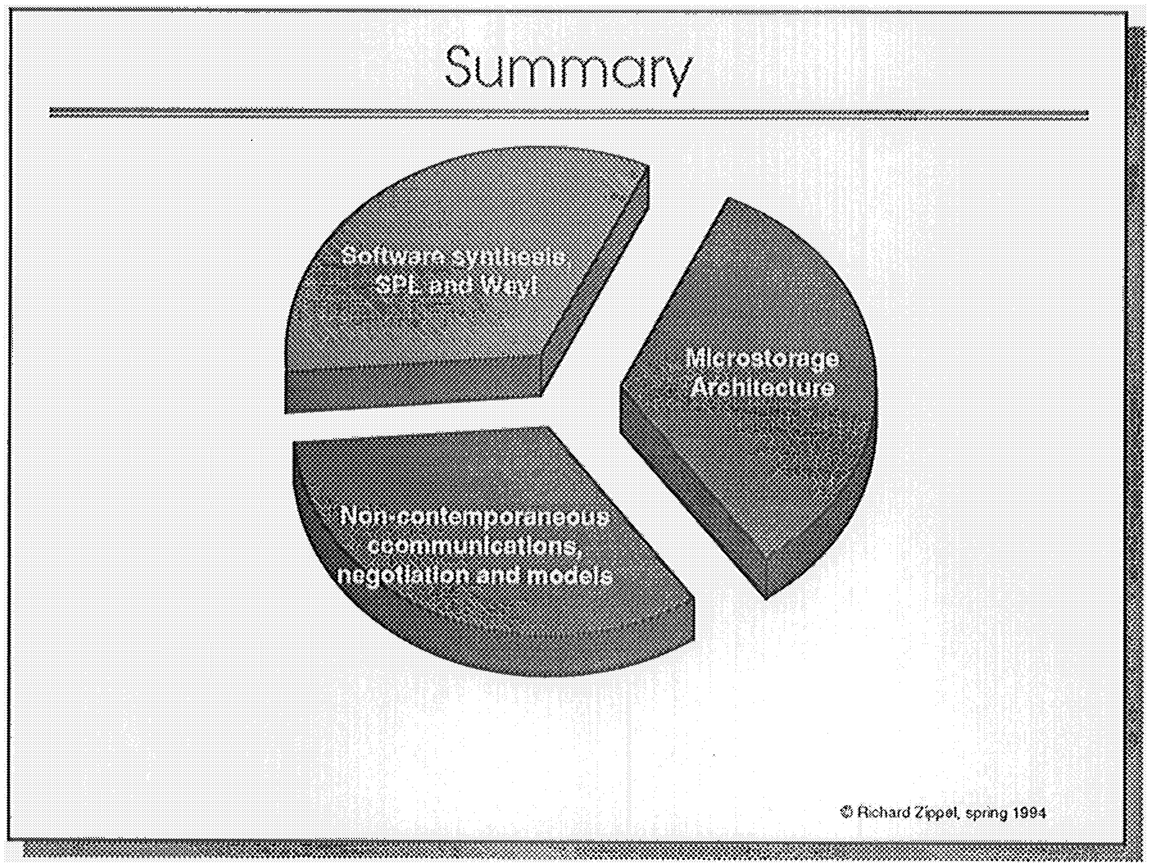
- Variables can be continuous functions
- Constraint language with differential equation constraints—natural way of describing physical situations

Program Transformations

- Capture mathematical techniques
- Blends compiler optimizations and mathematical analysis

On the left we have a more complex SPL program. The variables constrained are infinitely differentiable, real valued functions one variable. These functions are constrained to satisfy the the two differential equations indicated.

Our library includes transformations like the Forward Euler transform, which converts the program on the left to the one on the right. Prior to the use of x and y in the print statement, their values are updated based on the Forward Euler update formula. In fact, the real transformation introduces an additional adaptive timestepping loop, so that new values of x and y are computed sufficiently accurately.



We have outlined three different mechanisms that enter into the creation of intelligent systems—computation, storage and communications. Advanced information systems will integrate these systems mechanisms to a greater degree than is common today. For each of these three mechanisms we have presented some new ideas that acheive greater modularity and extensibility than previous approaches.